# Augur: a Modeling Language for Data-Parallel Probabilistic Inference

**Jean-Baptiste Tristan** JEAN.BAPTISTE.TRISTAN@ORACLE.COM
Oracle Labs

**Daniel Huang** DEHUANG@FAS.HARVARD.EDU
Harvard University

**Joseph Tassarotti** JTASSARO@CS.CMU.EDU
Carnegie Mellon University

**Adam Pocock** ADAM.POCOCK@ORACLE.COM
Oracle Labs

**Stephen J. Green** STEPHEN.X.GREEN@ORACLE.COM
Oracle Labs

**Guy L. Steele, Jr** GUY.STEELE@ORACLE.COM
Oracle Labs



## Abstract

It is time-consuming and error-prone to implement inference procedures for each new probabilistic model. Probabilistic programming addresses this problem by allowing a user to specify the model and having a compiler automatically generate an inference procedure for it. For this approach to be practical, it is important to generate inference code that has reasonable performance. In this paper, we present a probabilistic programming language and compiler for Bayesian networks designed to make effective use of data-parallel architectures such as GPUs. Our language is fully integrated within the Scala programming language and benefits from tools such as IDE support, type-checking, and code completion. We show that the compiler can generate data-parallel inference code scalable to thousands of GPU cores by making use of the conditional independence relationships in the Bayesian network.

## 1. Introduction

Machine learning, and especially probabilistic modeling, can be difficult to apply. A user needs not just to design the model, but also to implement the right inference procedure. There are many different inference algorithms, most of which are conceptually complicated and difficult to implement at scale. Despite the enthusiasm that many engineers who practice data analysis have for machine learning, this complexity can be a barrier to deployment. Any effort to simplify the use of machine learning would thus be very useful.

Probabilistic programming (Goodman, 2013), as introduced in the BUGS project (Thomas et al., 1992), is a way to simplify the application of machine learning based on Bayesian inference. The key feature of probabilistic programming is separation of concerns: the user specifies *what* needs to be learned by describing a probabilistic model, while the compiler automatically generates the *how*, that is, the inference procedure. In probabilistic programming, the programmer writes a program whose semantics is a probability distribution. Using a compiler-generated inference algorithm, the programmer can then sample from this distribution.

However, doing inference on probabilistic programs is computationally intensive and challenging. As a result, developing algorithms to perform inference is an active area of research. These include deterministic approximations (such as variational methods) and Monte Carlo approximations (such as MCMC algorithms). The problem is that most of these algorithms are conceptually complicated, and it is not clear, especially for non-experts, which one would work best for a given model.

To address the performance issues, our work has been driven by two observations. The first observation is that

good performance starts with the appropriate inference algorithms, and selecting the right inference procedure is often the hardest problem. For example, if our compiler emits only Metropolis-Hastings inference with generic proposals, there are models for which our programming language will be of no use, even given large amounts of computational power. We must design the compiler in such a way that we can include the latest inference research while reusing pre-existing analyses and optimizations, or even mix inference techniques. Consequently, we have designed our compiler as a modular framework where one can add a new inference algorithm while reusing already implemented analysis and optimizations. For that purpose, our compiler uses an intermediate representation (IR) for probability distributions that serves as a target for modeling languages and as a basis for inference algorithms.

The second observation is if we wish to continue to benefit from advances in hardware we *must* focus on producing highly *parallel* inference algorithms. We claim that many MCMC inference algorithms are highly data-parallel (Hillis & Steele, 1986; Blelloch, 1996) if we take advantage of the conditional independence relationships of the input model (*e.g.* the assumption of *i.i.d.* data makes the likelihood independent across data points). Moreover, we can automatically generate good data-parallel inference with a compiler. Such inference will run very efficiently on highly parallel architectures such as Graphics Processing Units (GPUs). It is important to note that parallelism brings an interesting trade-off for performance since some inference techniques (such as collapsing for Gibbs sampling) can result in less parallelism and will not scale as well.

In this paper, we present our compilation framework, named *Augur*. To start (section 2), we demonstrate how to specify the latent Dirichlet allocation (LDA) model (Blei et al., 2003) in our language and use it to learn the topics from a corpus of text. We then review the two contributions of our work, with a focus on our LDA example. First (section 3), we describe how we support different modeling languages by embedding them into Scala using Scala's macro system, which provides type checking and IDE support, and we describe the probabilistic IR. Second, we describe data-parallel versions of Metropolis-Hastings and Gibbs sampling that scale on the GPU and speed up sampling (section 4). Next, we present the results of our preliminary benchmarks, which include comparisons against other implementations of inference for LDA but also against our own hand-written data-parallel and optimized implementation of a Gibbs sampler for LDA (section 5). The paper uses LDA as a running example, but many more models can be expressed in Augur, and we present examples of regression, clustering, and classification (section 6). Finally, we compare our work with other research in probabilistic programming (section 7).

Our main result is that not only are some inference algorithms highly data-parallel and amenable to GPU execution, but a compiler can *automatically* generate such GPU implementations effectively.

## 2. Example: Latent Dirichlet Allocation

As an example, we first present the specification of the latent Dirichlet allocation model (Blei et al., 2003) in Augur and show how to use it to learn the topics present in a set of documents. The supplementary material contains other examples of probabilistic models in Augur.

### 2.1. Specification of LDA

The specification of the LDA model in Augur is presented in figure 1. The probability distribution is defined as a Scala object (`object LDA`) and is composed of two declarations. First, we declare the support of the probability distribution as a class that must be named `sig`. Here the support is composed of four arrays, with one each for the distribution of topics per document (`theta`), the distribution of words per topic (`phi`), the topics assigned to the words (`z`), and the words in the corpus (`w`). The support is used to store the inferred model parameters. These last two arrays are flat representations of ragged arrays, and so we do not require the documents to be of equal length.

The second declaration specifies the Bayesian network associated with LDA and makes use of our domain specific language for Bayesian networks. The DSL is marked by the `bayes` keyword and delimited by the following enclosing brackets. The model first declares the parameters of the model: `K` for the number of topics, `V` for the vocabulary size, `M` for the number of documents, and `N` for the array that associates each document with its size.

In the model itself, we define the hyper-parameters (values `alpha` and `beta`) for the Dirichlet distributions and draw `K` Dirichlet samples of dimension `V` for the distribution of words per topic (`phi`) and `M` Dirichlet samples of dimension `K` for the distribution of topics per document (`theta`). Then, for each word in each document, we draw a topic `z` from `theta`, and finally a word from `phi` based on the topic we drew for `z`.

### 2.2. Using the model

Once a model is specified, it can be used as any other Scala object by writing standard Scala code. For instance, one may want to use the LDA model with a training corpus to learn a distribution of words per topic and then use it to learn the per-document topic distribution of a test corpus. An implementation is presented in Figure 2. First the programmer must allocate the parameter arrays which contain the inferred values. Then the signature of the model

```
1  object LDA {
2  class sig(var phi : Array[Double], var theta : Array[Double], var z : Array[Int],
       var w : Array[Int])
3
4  val model = bayes {
5   (K: Int,V: Int,M: Int,N: Array[Int]) => {
6
7     val alpha = vector(K,.1)
8     val beta = vector(V,.1)
9
10    val phi = Dirichlet(V,beta).sample(K)
11    val theta = Dirichlet(K,alpha).sample(M)
12    val w =
13      for(i <- 1 to M) {
14        for(j <- 1 to N(i)) {
15          val z: Int = Categorical(K,theta(i)).sample()
16          Categorical(V,phi(z)).sample()
17        }
18      }
19    observe(w)
20   }
21 }}
```

*Figure 1.* Specification of the latent Dirichlet allocation model in Augur. The model specifies the probability distribution $p(\phi, \theta, z \mid w)$. The keyword `bayes` introduces the modeling language for Bayesian networks.

is constructed which encapsulates the parameters. The `LDA.model.map` command returns the MAP estimate of the parameters given the observed words.

To test the model, a new signature is constructed containing the test documents, and the previously inferred `phi` values. Then `LDA.model.map` is called again, but with both the phis and the words observed (by supplying `Set("phi")`). The inferred thetas for the test documents are stored in `s_test.theta`, for display or use as features in another system.

## 3. A Modular Compilation Framework

Before we detail the architecture of our compiler, it is useful to understand how our LDA example goes from a specification down to CUDA code running on the GPU. There are two distinct compilation phases. The first happens when the programmer compiles his program with a command such as (assuming that the code from Figure 1 is in a file named `LDA.scala`)

```
scalac -classpath augur.jar LDA.scala
```

The file `augur.jar` is the package containing our compiler. The first phase of compilation happens statically, during normal `scalac` compilation. In this phase, the block of code following the `bayes` keyword is transformed into our intermediate representation for probability distributions. The second compilation phase happens at runtime, when the programmer calls the `LDA.model.map` method. At that point, the IR is transformed, analyzed, and optimized, and finally, CUDA code is emitted and run.

Our framework is therefore composed of two distinct components that communicate through the IR: the front end, where domain specific languages are converted into the IR, and the back end, where the IR can be compiled down to various inference algorithms (currently Metropolis-Hastings, and Gibbs sampling). To define a modeling language in the front end, we make use of the Scala macro system. The macro system allows us to define a set of functions (called "macros") that will be executed by the Scala compiler on the code enclosed by the macro. We are currently focusing on Bayesian networks (as presented in our example for LDA), but other DSLs (*e.g.* , Markov random fields) could be added without modifications to the back end. The implementation of the macros to define the Bayesian network language is conceptually uninteresting so we will not give further details. Our Bayesian network language is fairly standard, with the notable exception that it is implicitly parallel.

Separating the compilation into two distinct phases gives us many advantages. As our language is implemented using Scala's macro system it provides automatic syntax highlighting, method name completion and code refactoring in any IDE which supports Scala. This greatly improves the usability of the DSL as no special tools need to be developed to support it. This macro system allows Augur to use Scala's parser, semantic analyzer (*e.g.* , to check that variables have been defined), and type checker. Also we bene-

```
1  val phi = new Array[Double](k * v)
2  val theta_train = new Array[Double](doc_num_train * k)
3  val z_train = new Array(num_tokens_train)
4  val s_train = new LDA.sig(phi, theta_train, z_train, w_train)
5  LDA.model.map(Set(), (k, v, doc_num_train, docs_length_train), s_train, s_train,
       samples_num, Infer.GIBBS)
6
7  val z_test = new Array(num_tokens_test)
8  val theta_test = new Array[Double](doc_num_test * k)
9  val s_test = new LDA.sig(phi, theta_test, z_test, w_test)
10 LDA.model.map(Set("phi"), (k, v, doc_num_test, docs_length_test), s_test, s_test,
        samples_num, Infer.GIBBS)
```

*Figure 2.* Example use of the LDA specification. Function `LDA.model.map` returns a maximum a posteriori estimation. It takes as arguments, in order, the set of variables to observe (on top of the ones declared as observed in the model specification), the hyperparameters, the initial parameters, the output parameters, the number of iterations and the inference to use. The parameters are stored in `LDA.sig`.

fit from the Scala compiler's optimizations such as constant folding and dead code elimination.

Then, because the IR is compiled to CUDA code at runtime, we know the values of all the hyper-parameters, and the size of the dataset. This enables better optimization strategies, and also gives us key insights into how to extract parallelism (see section 4.2). For example, when compiling LDA, we know that the number of topics is much smaller than the number of documents and thus parallelizing over documents will produce more parallelism than parallelizing over topics.

Finally, we also provide a library which defines standard distributions such as Gaussian, Dirichlet, etc. In addition to these standard distributions, each model denotes its own user-defined distribution. All of these distributions are subtypes of the `Dist` supertype. Currently, the `Dist` interface provides two methods: `map`, which implements maximum a posteriori estimation; `sample`, which returns a sequence of samples.

Note that even though our main interest is in GPU computation, it is possible to re-target the backend for other architectures, such as multiprocessors (using OpenMP) or distributed systems (using MPI). For example, our compiler has an option to emit standard sequential C code, though we mainly use this feature for debugging purposes.

## 4. Generation of Data-Parallel Inference

When an inference procedure is invoked on a model (e.g. `LDA.model.map`), the IR is compiled down to CUDA inference code for that model. Informally, our IR expressions are generated from the Backus-Naur form grammar below

$$P ::= p(\vec{X}) \;\Big|\; p(\vec{X} \;\Big|\; \vec{X}) \;\Big|\; PP \;\Big|\; \frac{1}{P}$$
$$\Big|\; \prod_i^N P \;\Big|\; \int_X P\, dx \;\Big|\; \{P\}_c$$

The goal of the IR is to make the sources of parallelism in the model more explicit and to support analysis of the probability distributions present in the model. For example, a $\prod$ indicates that each sub-term can be evaluated in parallel.

In the rest of this section, we explain how the compiler generates data-parallel samplers that exploit the conditional independence structure of the model. We will use the LDA example to explain how the compiler analyzes the model and generates the inference.

### 4.1. Generation of a data-parallel Metropolis sampler

In the case where the user wants to use Metropolis-Hastings inference on the LDA model, the compiler needs to emit code for a function $f$ that is proportional to the distribution the user wants to sample from. This function is then linked with our library implementation of Metropolis-Hastings. The function $f$ is composed of the product of the prior and the likelihood of the model and is extracted automatically from the model specification. In the case of LDA, $f$ is defined as

$$f(\theta, \phi, z, w) = p(w|\phi, z)p(z|\theta)p(\theta)p(\phi)$$

which is equal to (and represented in our IR as)

$$\left(\prod_i^M p(\theta_i) \prod_j^{N(i)} p(w_{ij}|\phi_{z_{ij}})p(z_{ij}|\theta_i)\right)\left(\prod_k^K p(\phi_k)\right)$$

In this form, the compiler knows that the distribution factorizes into a large number of terms that can be evaluated in

parallel and then efficiently multiplied together, and more specifically, in particular it knows that the data is *i.i.d.* and that it can optimize accordingly. In this case, each document contributes to the likelihood independently, and they can be evaluated in parallel. In practice, we work in log-space, so we perform summations. The compiler can then generate the CUDA code for the evaluation of $f$ from the IR representation. This code generation step is conceptually simple and we will not explain it further.

It is interesting to note that despite the simplicity of this parallelization the code scales reasonably well: there is a large amount of parallelism because it is roughly proportional to the number of documents; uncovering the parallelism in the code does not increase the overall quantity of computation that has to be performed; and the ratio of computation to global memory accesses is high enough to hide memory latency bottlenecks.

### 4.2. Generation of a data-parallel standard Gibbs sampler for LDA

Alternatively, the compiler can generate a Gibbs sampler for LDA. Currently we cannot generate a collapsed or blocked sampler, but there is interesting work related to dynamically collapsing or blocking variables (Venugopal & Gogate, 2013) and we leave it to future work to extend our compiler with this capability.

To generate a Gibbs sampler, the compiler needs to figure out how to sample from each univariate distribution. As an example, to draw $\theta_m$ as part of the $(\tau+1)$th sample, the compiler needs to generate code that samples from the following distribution

$$p(\theta_m^{\tau+1}|w^{\tau+1}, z^{\tau+1}, \theta_1^{\tau+1}, ..., \theta_{m-1}^{\tau+1}, \theta_{m+1}^{\tau}, ..., \theta_M^{\tau})$$

To accomplish this, the compiler implements an algebraic rewrite system that attempts to rewrite the above expression in terms of expressions it knows (i.e., the joint distribution of the entire model). We show a few selected rules below to give a flavor of the rewrite system.

(a) $\frac{P}{P} \Rightarrow 1$

(b) $\int P(x)\, Q\, dx \Rightarrow Q \int P(x) dx$

(c) $\prod_i^N P(x_i) \Rightarrow \prod_i^N \{P(x_i)\}_{q(i)=true} \prod_i^N \{P(x_i)\}_{q(i)=false}$

(d) $P(x) \Rightarrow \frac{P(x,y)}{\int P(x,y)\, dy}$

Rule (a) states that like terms can be canceled. Rule (b) says that terms not dependent on the variable of integration can be pulled out of the integral. Rule (c) says that we can partition a product over N-terms into two products, one where a predicate $q$ is satisfied on the indexing variable and one where it is not. Rule (d) is a combination of the product and sum rule. Currently, the rewrite system is just comprised of rules we found useful in practice, and it easy to extend the system to add more rewrite rules. We will review properties of this rewrite system toward the end of this section.

**Algorithm 1** Sampling from Dirichlet($\alpha$) M times

**Input:** array $\alpha$ of size $n$
**for** M documents in parallel **do**
  **for** $i = 0$ **to** $n-1$ **do**
    $v[i] \sim \texttt{Gamma}(a[i])$
  **end for**
  $s = \sum_0^{n-1} a[i]$ in parallel
  **for** $i = 0$ **to** $n-1$ in parallel **do**
    $v[i] = \frac{v[i]}{s}$
  **end for**
**end for**
**Output:** array $v$

Going back to our example, the compiler might rewrite the desired expression into the one below:

$$\frac{p(\theta_m^{\tau+1}) \prod_j^{N(m)} p(z_{mj}|\theta_m^{\tau+1})}{\int p(\theta_m^{\tau+1}) \prod_j^{N(m)} p(z_{mj}|\theta_m^{\tau+1}) d\theta_m^{\tau+1}}$$

In this form, it is clear that each of the $\theta_1, \ldots, \theta_m$ is independent of the others after conditioning on the other random variables. As a result, they may all be sampled in parallel.

At each step, the compiler can test for a conjugacy relation. In the above form, the compiler recognizes that the $z_{mj}$ are drawn from a categorical distribution and $\theta_m$ is drawn from a Dirichlet, and can exploit the fact that these are conjugate distributions. The posterior distribution for $\theta_m$ is:

$$\texttt{Dirichlet}(\alpha + c_m)$$

where $c_m$ is a vector whose $k$th entry is the number of $z$ associated with document $m$ that were assigned topic $k$. Drawing for each $\theta_m$ can be done in parallel with a simple algorithm (algorithm 1) that samples from a Dirichlet by normalizing Gamma variates (Marsaglia & Tsang, 2000). Also, we now know that the drawing of each $z$ must include a counting phase.

The case of the $\phi$ variables is slightly more interesting. In this case, we want to sample from

$$p(\phi_k^{\tau+1}|w^{\tau+1}, z^{\tau+1}, \theta^{\tau+1}, \phi_1^{\tau+1}, ..., \phi_{k-1}^{\tau+1}, \phi_{k+1}^{\tau}, ..., \phi_K^{\tau})$$

After the application of the rewrite system to this expres-

**Algorithm 2** Sampling from K Dirichlet

**Input:** matrix $a$ of size $k$ by $n$
**for** $i = 0$ **to** $n - 1$ in parallel **do**
  **for** $j = 0$ to $k - 1$ **do**
    $v[i, j] \sim \texttt{Gamma}(a[i, j])$
  **end for**
  $\overrightarrow{v} \times 1$
**end for**
**Output:** matrix $v$

sion, the compiler discovers that this is equal to

$$\frac{p(\phi_k)\prod_i^M \prod_j^{N(i)} \{p(w_i|\phi_{z_{ij}})\}_{k=z_{ij}}}{\int p(\phi_k)\prod_i^M \prod_j^{N(i)} \{p(w_i|\phi_{z_{ij}})\}_{k=z_{ij}} d\phi_k}$$

The key observation that the compiler takes advantage of to reach this conclusion is the fact that the $z$ are distributed according to a categorical distribution and are used to index into the $\phi$ array. Therefore, they partition the set of words $w$ into $K$ disjoint sets $w_1 \uplus ... \uplus w_k$, one for each topic. More concretely, the probability of the words that belong to topic $k$ can be rewritten in partitioned form using rule (c) as

$$\prod_i^M \prod_j^{N(i)} \{p(w_{ij}|\phi_{z_{ij}})\}_{k=z_{ij}}$$

This expresses the intuition that once a word's topic is fixed, the word depends on only one of the $\phi_k$ distributions. In this form, the compiler recognizes that it should draw from

$$\texttt{Dirichlet}(\beta + c_k)$$

where $c_k$ is the count of words assigned to topic $k$. In general, the compiler detects patterns like the above when it notices that samples drawn from categorical distributions are being used to index into arrays.

In the previous case, sampling from a Dirichlet $M$ times in parallel was straightforward and effective. In this case, it is not, because parallelizing over the number of topics $K$ will typically not provide sufficient parallelism for a scalable GPU execution, as we need an order of magnitude more threads to use the GPU effectively. Thus, while the compiler can exploit the conditional independence present in the model, a smarter compiler can discover other sources of parallelism as well. Since drawings from the `Gamma` distributions are independent from each other, it is possible to implement the sampling as presented in algorithm 2. In this version, for each vocabulary word in parallel, we draw the gammas for all of the topics. Then to compute the normalizing constants, we multiply the resulting matrix by a unit vector using CUBLAS. Finally, we normalize the matrix. This is an instance where the two-stage compilation procedure described in section 3 is useful, because the compiler is able to use information about the relative sizes of $K$ and $V$ to decide that algorithm 2 may be more efficient.

Finally, the compiler turns to analyzing the $z_{ij}$. In this case, it will again detect that they can be sampled in parallel but it will not be able to detect a conjugacy relationship. It will then detect that the $z_{ij}$ are drawn from discrete distributions, so that the univariate distribution can be calculated exactly and sampled from. In cases where the distributions are continuous, it can try to use another approximate sampling method as a subroutine for drawing that variable.

The rules in the rewrite system can be roughly divided into two groups. The first group contains simplification and normalization rules such as $P(X) \times 1 = P(X)$. These rules are repeatedly applied until we reach a normal form. The second group of rules contains more advanced rules that could potentially lead to cycles. Examples include the sum and product rules, the partitioning rule presented in this section, or rules related to integrals. An application of a rule from the second group can be followed by applications of rules from the first group to renormalize the expression. The compiler uses predefined pipelines of these rewrites to uncover conjugacy relationships.

The rewrite system is not specific to LDA. A rule such as the partitioning presented above corresponds to a common pattern in probabilistic modeling where a categorical random variable is used to index a vector of random variables (e.g. mixture model). More generally, the rewriting process always terminates and it is deterministic. In the case where a conjugacy relation cannot be found for one of the univariate conditionals, the compiler can revert to using the Metropolis-Hastings algorithm. In general, the compiler could also use methods such as Rejection sampling or slice sampling. It would be useful to be able to characterize the class of models for which Augur can be expected to discover the intended conjugacy relationships of a model. It is a difficult problem since in all generality, the model could contain complicated arithmetic expressions (for instance, a polynomial regression with order 5 or higher). Note that in many simple practical cases, the compiler will discover conjugacy relationships despite the use of arithmetic operations because the optimizer will simplify the model enough.

It is interesting to note that many of the optimizations used in the literature to improve the mixing time of a Gibbs sampler, such as blocking or collapsing can reduce the amount of available parallelism by introducing dependencies between previously independent variables. The use of such techniques in a system like Augur introduces interesting tradeoffs: it is not always beneficial to "eliminate" some variables (for example by collapsing) if it results in more

dependencies for the remaining variables. The amount of parallelism in the inference produced by the compiler depends directly on the amount of conditional independence in the model. Consequently, in its current state, the compiler would not produce good parallel code for a model like a hidden Markov model because of the dependencies between observations.

## 5. Experimental Study

To evaluate the code generated by our compiler, we compare its performance with that of Factorie (McCallum et al., 2009) on the LDA example (specifically the Factorie 1.0.0-M6 release). Like Augur, Factorie is a Scala library that tries to make it easier to use machine learning algorithms. However, in contrast to Augur's language-based approach, Factorie allows users to express graphical models by constructing and composing Scala objects. These objects are then passed as input to several inference algorithms written in Scala.

Factorie features two implementations of LDA. First, it has a collapsed Gibbs sampler (Griffiths & Steyvers, 2004) defined using its primitives for constructing graphical models. In addition, it includes an alternate collapsed Gibbs sampler specific to LDA, known as SparseLDA (Yao et al., 2009) (not to be confused with Sparse Stochastic LDA); SparseLDA is a specialized and highly optimized sampler for LDA and is implemented in Scala. The former represents what a user of Factorie could write, while the latter represents what an expert familiar with the recent literature on inference algorithms tuned for LDA could do.

Like the collapsed Gibbs variant in Factorie, Augur requires only a model specification; the inference is supplied by the Augur compiler. Our aim is to see if exploiting the parallelism provided by the GPU can lead to runtime performance equivalent to a specialized model-specific inference method, whilst using general-purpose inference methods and compile-time optimizations.

We tried to compare Augur against other probabilistic programming languages, but some were unable to handle the experimental data set. For instance, the Stan (Hoffman & Gelman, In press) system has a prohibitive memory requirement (more than 40 GB) for our data set[1]. Many other probabilistic programming systems discussed in Section 7 focus on language design and expressiveness rather than on scalability and performance and we do not consider them for this reason.

We also compare the performance of the code generated by our compiler against an optimized hand-written CUDA implementation of a Gibbs sampler for LDA. The hand-written implementation is optimized for the particular system we use for these experiments, and carefully manages data locality and data transfers between the different memories of the GPU. We expect the hand-written implementation to form an upper-bound on inference speed.

[1]Stan uses Hamiltonian Monte Carlo as its inference method, which does not perform well on LDA and other topic models.

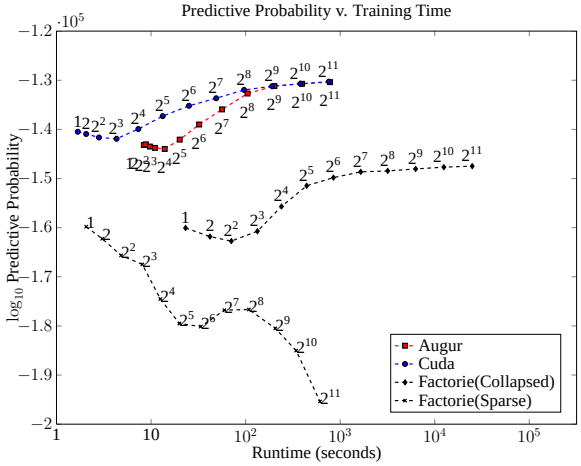


*Figure 3.* Evolution of the predictive probability over time for up to 2048 samples and for four implementations of LDA inference: Augur, hand-written CUDA, Factorie's Collapsed Gibbs and SparseLDA.

### 5.1. Experimental Setup

We used a corpus extracted from the simple English variant of Wikipedia, with standard stopwords removed. This gives a corpus with 48556 documents, a vocabulary size of 37276 words, and approximately 3.3 million tokens. From that we sampled 1000 documents to use as a test set, removing words which only appear in the test set. To evaluate the model fit we use the log predictive probability measure (Hoffman et al., 2013) on the test set.

All experiments were run on a single workstation with an Intel Core i7 3770 @3.4GHz CPU, 32 GB RAM, a Samsung 840 SSD, and an NVIDIA Geforce Titan (@837MHz). The Titan runs on the Kepler architecture. All probability values are calculated in double-precision. The CPU performance results using Factorie are calculated using a single thread, as the multi-threaded samplers are neither stable or performant in the tested release. The GPU results use all 896 double precision ALU cores available in the Titan[2].

### 5.2. Results

Our first experiment (figure 3) compares the convergence of the predictive probability over time for the four systems. We compute the predictive probability and record the time after drawing $2^i$ samples, for $i$ ranging from 0 to 11 inclusive. The time is reported in natural logarithm of runtime in milliseconds. It takes Augur 8.1 seconds to draw its first sample, whereas it takes less than a second for the sparse implementation. The reason Augur starts so slowly is that we need to compile the model to CUDA code, call `nvcc` to compile the result, and copy the data to the graphics card. Augur's predictive probability for 0 samples (not shown above) is much lower than that of the two Factorie versions because they incorporate the prior when no samples have been drawn, whereas currently Augur does not. Notice that the sparse implementation and Augur draw 2048 samples in about the same amount of time. However, the predictive probability of the sparse implementation is much lower and decreasing. It takes 6.7 more hours for the collapsed LDA implementation to draw 2048 samples than it does for the other systems. Finally, even though the hand-written CUDA implementation starts much faster than Augur's, they have similar run time characteristics as the number of samples increases. Augur's good performance compared to the hand-written CUDA implementation (that carefully optimize data transfers and locality) is due to the high ratio of computations to global memory accesses in inference code.

We have two unanswered questions about this experiment. First, we do not understand why the predictive probability of Factorie's collapsed Gibbs sampler is lower than that of the hand-written CUDA code or Augur. We tried different random number generators, but that did not significantly change the performance. Second, we do not understand why SparseLDA performance decreases over time. However, the resulting topics are reasonable so it could be that predictive probability applies poorly to SparseLDA. In any case, we hope this still demonstrates the time per sample performance.

In an attempt to confirm this result, our second experiment (figure 4) reports again on the predictive probability over time for the collapsed LDA implementation and Augur. In this experiment, we have made 10 runs with different train/test splits and present the timings with error bars. We also made an additional 10 runs while changing the random seed for the random number generators and the results are similar. We reduced the maximum number of samples to 512 as generating results for the Collapsed Gibbs sampler was proving prohibitive in terms of runtime for repeated

[2]The Titan has 2688 single precision ALU cores, but single precision resulted in poor quality inference results, though the speed was greatly improved.

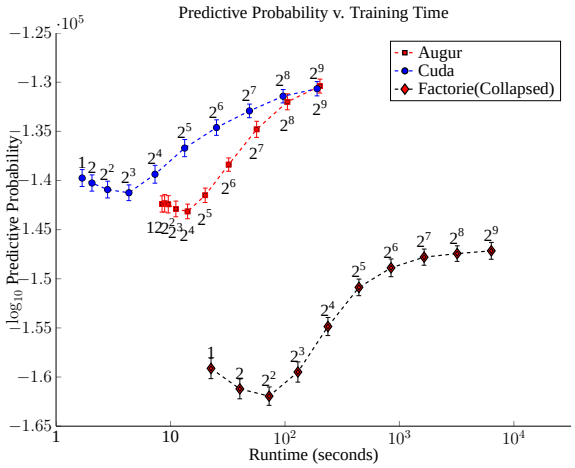


*Figure 4.* Average over 10 runs of the evolution of the predictive probability over time.

experiments.

Our third experiment (figure 5) reports on the natural logarithm of run time in milliseconds to draw 512 samples as the number of topics varies. The sparse implementation's running time does not increase as quickly as Augur's as the number of topics increases. As a result, it runs faster when the number of topics is large. This is because Augur's Gibbs sampler is linear in the number of topics during the step of sampling each of the $z_{ij}$. In contrast, the SparseLDA sampler was found to be sublinear in the number of topics (Yao et al., 2009). This is due to the way the topics are represented, where each word can appear in $n$ topics where $n$ is the number of occurrences of that word. The collapsed Gibbs sampler's performance blows up when the number of topics is increased, as seen in our results and in the experiments in (Yao et al., 2009). Again, Augur's generated code is on par with the hand-written CUDA implementation.

In general, the reason Augur is much faster than the collapsed Gibbs sampler is the sheer number of parallel threads, even though Augur requires more FLOPS to draw each sample (as it is not a collapsed sampler). This highlights the importance of good parallelization strategies for the inference code, as the performance benefit of GPUs can be extracted only by generating large amounts of parallelism, in our case at least 896 simultaneous threads.

## 6. Other Examples

We present a few examples of model specifications in Augur, covering three important topics in machine learning: regression (6.1, 6.2), clustering (6.3, 6.4), and classification (6.5). Our goal is to show how a few popular models

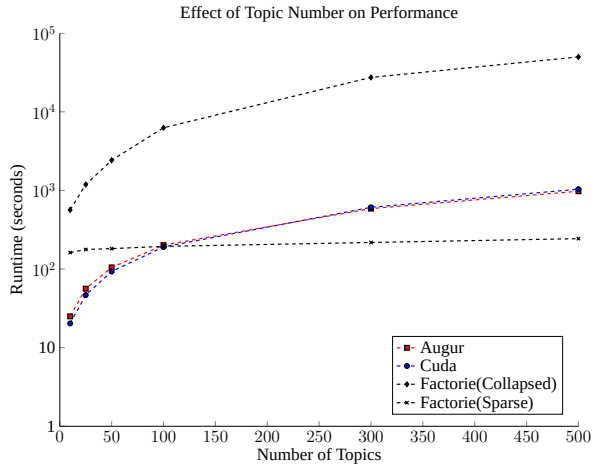


*Figure 5.* Comparison of scalability of Augur, hand-written CUDA, Factorie's collapsed Gibbs and SparseLDA *w.r.t* the number of topics.

can be programmed in Augur. For each of these examples, we first describe the support of the model, and then sketch the generative process, relating the most complex parts of the program to their usual mathematical notation.

### 6.1. Univariate polynomial regression

Our first example model is for univariate polynomial regression (Figure 6). The model's support is composed of the array `w` for the weights of each mononomial, `x` for the domain data points and `y` for their image. The parameters of the model are: `N`, the dataset size and `M`, the order of the polynomial. For simplicity, this example assumes that the domain of `x` ranges from 0 to 2.

The generative process is: We first independently draw each of the `M` weights, $w_i \sim N(0,1)$, then draw $(x, y)$ as follows:

$$x_j \sim \text{Uniform}(0, 2) \tag{1}$$

$$y_j \sim N(\sum_i^M w_i x_j^i, 1). \tag{2}$$

For simplicity, the model is presented with many "hard-wired" parameters, but it is possible to parameterize the model to control the noise level, or the domain of `x`.

### 6.2. Multivariate linear regression

The second example is a multivariate linear regression (Figure 7). The support is composed of an array `w` for the weights, a variable `bias`, and the arrays for the `x` and `y` data points. The parameters of the model are: `K` dimension, `N` data size, `xLower` and `xUpper` which define the domain of `x`. The input data for `x` is a $K \times N$ matrix flattened into an array.

The generative process is: We draw the weights $w_k \sim N(0,1)$ for each of the $K$ dimensions, a bias, and a $K$-dimensional `x` $\sim$ Uniform(`xLower`, `xUpper`), and finally $y$:

$$y \sim N(\sum_j^K w_j x_j, 1). \tag{3}$$

### 6.3. Categorical mixture

The third example is a categorical mixture model (Figure 8). The model's support is composed of an array `z` for the cluster selection, `x` for the data points that we draw, `theta` for the priors of the categorical that represents the data, and `phi` for the prior of the indicator variable. The parameters of the model are: `N` data size, `K` number of clusters, and `V` for the vocabulary size.

The generative process is: For each of the `N` data points we want to draw, we select a cluster `z` according to their distribution `phi` and then draw from the categorical with distribution given by `theta(z)`.

### 6.4. Gaussian mixture

The fourth example is a univariate Gaussian mixture model (Figure 9). The model's support is composed of an array `z` for the cluster selection, `x` for the data points that we draw, `mu` for the priors of the mean of the Gaussian, `sigma` for the priors of the variance of the Gaussian, and `phi` for the prior of the indicator variable. The parameters of the model are: `N` data size, `K` number of clusters.

The generative process is: For each of the `N` data points we want to draw, we select a cluster `z` according to their distribution `pi` and then draw from the Gaussian centered at `mu(z)` and of deviation `sigma(z)`.

### 6.5. Naive bayes classifier

The fifth example is a binary naive Bayes classifier (Figure 10). The support is composed of an array `c` for the class and an array `f` for the features, `pC` the prior on the positive class, and `pFgivenC` an array for the probability of each binary feature given the class. The hyperparameters of the model are: `N` the number of data points, `K` the number of features and. The features form a 2-dimensional matrix but again the user has to "flatten" the matrix into an array.

The generative process is: First we draw the probability of an event being in one class or the other as `pC`. We use `pC` has the parameter to decide for each event in which class it falls (`c`). Then, for each feature, we draw the probability of the feature occurring, `pFgivenC`, depending on whether the event is in the class or not. Finally, we draw the features

```
1  object UnivariatePolynomialRegression {
2
3    import scala.math._
4
5    class sig(var w: Array[Double], var x: Array[Double], var y: Array[Double])
6
7    val model = bayes {
8      (N: Int, M: Int) => {
9
10     val w = Gaussian(0,1).sample(M)
11     val x = Uniform(0,2).sample(N)
12     val bias = Gaussian(0,1).sample
13     val y = for(i <- 1 to N) {
14               val monomials = for (j <- 1 to M) yield { w(j) * pow(x(i),j) }
15               Gaussian((monomials.sum) + bias, 1).sample()
16             }
17
18     observe(x, y)
19     }
20   }
21 }
```

*Figure 6.* Specification of a univariate polynomial regression

```
1  object MultivariateLinearRegression {
2
3    class sig(var w: Array[Double], var bias: Double, var x: Array[Double], var y:
         Array[Double])
4
5    val model = bayes {
6      (K: Int, N: Int, xLower: Double, xUpper: Double) => {
7
8      val w = Gaussian(0, 1).sample(K)
9      val bias = Gaussian(0, 1).sample()
10     val x = for(i <- 1 to N) yield { Uniform(xLower, xUpper).sample(K) }
11
12     val y = for(i <- 1 to N) {
13               val basisFunctions = for(j <- 1 to K) yield { w(j) * x(i)(j) }
14               Gaussian((basisFunctions.sum) + bias, 1).sample()
15             }
16
17     observe(x, y)
18     }
19   }
20 }
```

*Figure 7.* Specification of a multivariate linear regression

```
object CategoricalMixture {
  class sig(var z: Array[Int], var x: Array[Int], var theta: Array[Double], var
      phi: Array[Double])
  val model = bayes {
    (N: Int, K: Int, V: Int) => {

      val alpha = vector(V,0.5)
      val beta = vector(K,0.5)

      val theta = Dirichlet(V,alpha).sample(K)
      val phi = Dirichlet(K,beta).sample()

      val x = for(i <- 1 to N) {
           val z = Categorical(K, phi).sample()
           Categorical(N,theta(z)).sample()
         }
      observe(x)
    }
  }
}
```

*Figure 8.* Specification of a categorical mixture model

```
object GaussianMixture {

  class sig(var z: Array[Int], var x: Array[Double], var mu: Array[Double], var
      sigma: Array[Double], var phi: Array[Double])

  val model = bayes {
    (N: Int, K: Int, V: Int) => {

    val alpha = vector(V,0.1)

    val phi = Dirichlet(V,alpha).sample()
    val mu = Gaussian(0,1).sample(K)
    val sigma = InverseGamma(1,1).sample(K)

    val x = for(i <- 1 to N) {
            val z = Categorical(K, phi).sample()
            Gaussian(mu(z), sigma(z)).sample()
          }

      observe(x)
    }
  }
}
```

*Figure 9.* Specification of a Gaussian mixture model

```
1  object NaiveBayesClassifier {
2
3    class sig(var c: Array[Int], var f: Array[Int], var pC: Double, var pFgivenC:
       Array[Double])
4
5    val model = bayes {
6      (N: Int, K: Int) => {
7
8      val pC = Beta(0.5,0.5).sample()
9      val c = Bernoulli(pC).sample(N)
10
11     val pFgivenC = Beta(0.5,0.5).sample(K*2)
12
13     val f = for(i <- 0 until N) {
14             for(j <- 0 until K) {
15               Bernoulli(pFgivenC(j * 2 + c(i))).sample()
16             }
17           }
18
19     observe(f, c)
20     }
21   }
22 }
```

*Figure 10.* Specification of a naive Bayes classifier

`f` for each event.

## 7. Related Work

If we define probabilistic programming as the use of programming languages to specify probability distributions along with a compiler to automatically produce an inference implementation from this specification, it is likely that BUGS (Lunn et al., 2009) was the first probabilistic programming language. Interestingly, most of the key concepts of probabilistic programming already appeared in the first paper to introduce BUGS (Thomas et al., 1992). Since then, research in probabilistic programming languages has been focused in two directions: improving performance and scalability through better inference generation; and, increasing expressiveness and building the foundations of a universal probabilistic programming language. These two directions are useful criteria to compare probabilistic programming languages.

In terms of language expressiveness, Augur is currently limited to the specification of Bayesian networks. It is possible to extend this language (*e.g.*, non-parametric models) or to add new modeling languages (*e.g.*, Markov random field), but our current focus is on improving the inference generation. That is in contrast with languages like Hansei (Kiselyov & Shan, 2009), Odds (Stucki et al., 2013), Stochastic Lisp (Koller et al., 1997) and Ibal (Pfeffer, 2007) which focus on increasing expressiveness, at the expense of performance. However, as Augur is embedded in the Scala programming language, we have access to the wide variety of libraries on the JVM platform and benefit from Scala tools. Augur, like Stan (Hoffman & Gelman, In press) and BUGS (Thomas et al., 1992; Lunn et al., 2009) is a domain specific probabilistic language for Bayesian networks, but it is embedded in such a way that it has a very good integration with the rest of Scala, which is crucial to software projects where data analysis is only one component of a larger artifact.

Augur is not the only system designed for scalability and performance. It is also the case of Dimple (Hershey et al., 2012), Factorie (McCallum et al., 2009), Infer.net (Minka et al., 2012) and Figaro (Pfeffer, 2012; 2009), and the latest versions of Church (Goodman et al., 2008). Dimple focuses on performance using specialized inference hardware, though it does provide an interface for CPU code. Factorie mainly focuses on undirected networks, and is a Scala library rather than a DSL (unlike all the other systems mentioned). It has multiple inference backends, and aims to be a general purpose machine learning package. Infer.net is the system most similar to Augur, in that it has a two phase compilation approach, though it is based around variational methods. A block Gibbs sampler exists but is only functional on a subset of the models. Figaro focuses on a different set of inference techniques, including techniques which use exact inference in discrete spaces (they do have a Metropolis-Hastings inference). Church provides the ability to mix different inference algorithms and has some parallel capability, but it

focused on task-parallelism for multicores rather than on data-parallelism for parallel architectures. The key difference between Augur and these other languages is the systematic generation of data-parallel algorithms for large numbers of cores (*i.e.*, thousands) on generally available GPU hardware.

## 8. Conclusion

We find that it is possible to *automatically* generate parallel MCMC-based inference algorithms, and it is also possible to extract sufficient parallelism to saturate a modern GPU with thousands of cores. Our compiler achieves this with no extra information beyond that which is normally encoded in a graphical model description. This automatically generated parallel inference is competitive with a hand-tuned inference algorithm specific to a single model, both in terms of runtime and performance. By exploiting the parallelism inherent in inference it produces code which runs many times faster than the equivalent model written to use general purpose inference on off-the-shelf multicore CPUs. Of course, it is useless to attempt to speed up inference if one is not using the appropriate algorithm, but as it is possible to test multiple inference algorithms by varying a single parameter it becomes much easier to explore the space of inference techniques. Traditionally this would require rewriting the model in another probabilistic programming system, or hand-coding a new inference method. As the system provides multiple inference engines it is possible to allow an expert in the field of inference to mix and match inference methods, using Gibbs for some variables and Metropolis-Hastings steps for others, though precisely how to expose this in a probabilistic programming language is still an area of active research.

## References


Blei, David M., Ng, Andrew Y., and Jordan, Michael I. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

Blelloch, G. E. Programming parallel algorithms. *Communications of the ACM*, 39:85–97, 1996.

Goodman, N. D. The principles and practice of probabilistic programming. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pp. 399–402. ACM, 2013.

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

Griffiths, T. L. and Steyvers, M. Finding scientific topics. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 101, 2004.

Hershey, S., Bernstein, J., Bradley, B., Schweitzer, A., Stein, N., Weber, T., and Vigoda, B. Accelerating inference: towards a full language, compiler and hardware stack. *CoRR*, abs/1212.2991, 2012.

Hillis, W. D. and Steele, Jr., G. L. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

Hoffman, M., Blei, D., Wang, C., and Paisley, J. Stochastic variational inference. *Journal of Machine Learning Research*, 14:1303–1347, 2013.

Hoffman, M. D. and Gelman, A. The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, In press.

Kiselyov, O. and Shan, C.-C. Embedded probabilistic programming. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, DSL '09, pp. 360–384. Springer-Verlag, 2009.

Koller, D., McAllester, D., and Pfeffer, A. Effective bayesian inference for stochastic programs. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI)*, pp. 740–747, 1997.

Lunn, D., Spiegelhalter, D., Thomas, A., and Best, N. The BUGS project: Evolution, critique and future directions. *Statistic in Medicine*, 2009.

Marsaglia, G. and Tsang, W. W. A simple method for generating gamma variables. *ACM Trans. Math. Softw.*, 26 (3):363–372, 2000.

McCallum, A., Schultz, K., and Singh, S. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems 22*, pp. 1249–1257, 2009.

Minka, T., Winn, J.M., Guiver, J.P., and Knowles, D.A. Infer.NET 2.5, 2012. URL `http://research.microsoft.com/infernet`. Microsoft Research Cambridge.

Pfeffer, A. The Design and Implementation of IBAL: A General-Purpose Probabilistic Language. *Introduction to statistical relational learning*, pp. 399–433, 2007.

Pfeffer, A. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.

Pfeffer, A. Creating and manipulating probabilistic programs with figaro. In *2nd International Workshop on Statistical Relational AI*, 2012.

Stucki, S., Amin, N., Jonnalageda, M., and Rompf, T. What are the Odds? Probabilistic Programming in Scala. Scala '13, pp. 360–384. Springer-Verlag, 2013.

Thomas, A., Spiegelhalter, D. J., and Gilks, W. R. BUGS: a Program to Perform Bayesian Inference using Gibbs Sampling. *Bayesian statistics*, 4:837 – 842, 1992.

Venugopal, D. and Gogate, V. Dynamic blocking and collapsing for gibbs sampling. In *29th Conference on Uncertainty in Artificial Intelligence*, UAI'13, 2013.

Yao, L., Mimno, D., and McCallum, A. Efficient methods for topic model inference on streaming document collections. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pp. 937–946. ACM, 2009.