Probabilistic Programming with Programmable Divide-Conquer-Combine Inference on Modern Hardware

MARKUS BÖCK, TU Wien, Austria
JÜRGEN CITO, TU Wien, Austria

Universal probabilistic programming languages (PPLs) enable the specification of models with stochastic support structure. Posterior inference is notoriously hard and remains difficult to accelerate on modern hardware for this class of models. In response to these challenges, we introduce Upix – the first probabilistic programming system that realises the divide-conquer-combine (DCC) inference algorithm as a framework. In Upix, a model expressed in a universal PPL is automatically split into multiple sub-models with static support structure, which are then compiled with JAX for execution on accelerator hardware. The system allows extensive customisation of inference algorithms by incorporating numerous established concepts from programmable inference. To evaluate our system, we implemented two existing DCC algorithms in Upix and instantiated three novel algorithms. We show that our implementation achieves better approximation quality compared to existing approaches by achieving between 7 and 720 times more computation within the same time budget. On machines with up to 64 CPU cores and 8 GPU devices, we demonstrated that Upix enables the scaling of inference algorithms to workloads that are impractically slow for CPUs and prior methods.

CCS Concepts: • **Mathematics of computing** → **Bayesian computation**; **Statistical software**; • **Software and its engineering** → *Just-in-time compilers.*

Additional Key Words and Phrases: probabilistic programming, stochastic support, divide-conquer-combine, programmable inference, GPU-acceleration

1 Introduction

Probabilistic programming languages (PPLs) enable the specification of probabilistic models as programs and automate Bayesian inference. In their implementations, they have to trade off expressivity and inference efficiency.

On one side, there are PPLs like PyMC [48] or Stan [8] which restrict the class of programs to models with fixed and finite support structure. This allows efficient model representation and optimised implementations of inference algorithms like HMC [5], ADVI [23], or NUTS [20]. Even more, some probabilistic systems in this category like TensorflowProbability [24], BlackJAX [7], or NumPyro [41] build on the JAX numerical computing library to run inference on CPUs, GPUs, and TPUs via just-in-time (JIT) compilation.

On the other side, there are so-called *universal* PPLs like Gen [12], Turing [14], or Anglican [50], which embed their probabilistic constructs in a Turing-complete programming language. To support this large class of models they have to rely on general-purpose inference algorithms which often can be inefficient. To overcome this problem, these PPLs typically allow the user to customise the inference algorithms for the specific model at hand. This type of customisation, which came to be known as *programmable inference*, is actively researched and developed [4, 11, 12, 25, 30].

Universal PPLs gained popularity and research interest, because models with stochastic support structure can be easily expressed in them. Such models include mixture models with an unknown number of components [40, 44], kernel and program induction models [46, 47], "open-universe" models [33, 54], statistical phylogenetics models [45], models based on physical simulations [3], and many Bayesian non-parametric models [18, 28, 52].

However, to the best of our knowledge, there exists no system that 1) makes it easy to parallelise inference on GPUs or TPUs, 2) enables the specification of models in a universal PPL, and 3)

Authors' Contact Information: Markus Böck, markus.h.boeck@tuwien.ac.at, TU Wien, Vienna, Austria; Jürgen Cito, juergen.cito@tuwien.ac.at, TU Wien, Vienna, Austria.

implements a programmable inference machinery. In this work, we propose to bridge this gap by building a system following the *divide-conquer-combine* (DCC) approach introduced in 2020 by Zhou et al. [55].

Conceptionally, the DCC approach is rather simple: In the *divide step*, a model specified in a universal PPL is split up into multiple sub-models with fixed and finite support structure. In the *conquer step*, inference is performed on the sub-models by leveraging optimised algorithms. Lastly, in the *combine step*, all results are combined in an unbiased way to approximate the posterior distribution of the full model. However, in practice, each step comes with challenges:

- **Divide.** It is difficult to automatically split up a probabilistic program into sub-models. Existing approaches either rely on user-annotation or consider only the simpler case in which the program is split up by conditioning on the values of discrete variables.
- **Conquer.** The derived sub-models may still pose challenging inference problem rendering black-box approaches infeasible.
- **Combine.** To combine the results, we have to weigh the inference result of each sub-model proportional to its contribution to the full posterior. This amounts to computing the *normalisation constant* accurately for each sub-model – a notoriously difficult problem.

This work. We present Upix – a probabilistic programming system, which solves the above challenges by implementing the DCC approach for a universal PPL as a framework with programmable inference in JAX: In short, a JAX program transformation was developed to split up the probabilistic programs into sub-models automatically. Next, the system provides constructs to customise *both* the inference routine and the normalisation constant computation for each sub-model making the conquer and divide step fully programmable. Further, as a product of implementing the system in JAX, inference in Upix can greatly leverage vectorisation on CPUs, GPUs, and TPUs, as well as parallelisation across multiple devices. Lastly, through its abstractions Upix enables the development of new DCC-based inference algorithms.

Contributions. This paper contributes:

- Upix [1] – the first probabilistic programming system that realises the DCC approach as a framework and implements a programmable inference machinery to run inference on CPUs, GPUs, or TPUs for models specified in a universal PPL.
- Five instantiations of the framework yielding *three novel DCC-based algorithms* (two have been adapted from prior work): a Markov chain Monte Carlo approach as in the original DCC publication [55], Support Decomposition Variational Inference (SDVI) [42], a Reversible Jump / Involutive MCMC inspired approach, a Sequential Monte Carlo based approach, and a Variable Elimination based approach for discrete models.
- An empirical evaluation which compares Upix to existing work on challenging models showing that Upix achieves 7 to 720 more computation within the same time budget on consumer-grade CPUs, which substantially improves approximation quality.
- An investigation of the scaling properties of Upix on modern hardware (up to 64 CPU cores and 8 GPU devices), demonstrating that inference algorithms can be scaled to workloads that are impractically slow for CPUs and existing approaches.

2 Overview

Probabilistic programming languages (PPLs) are equipped with constructs for declaring random variables and conditioning on observed data. In this work, we focus on *universal PPLs* which are PPLs embedded in a Turing-complete language that support stochastic control flow and recursion [15]. In particular, we consider a language that allows the user to declare a random variable by linking a

dynamically computed label – the *address* – to a dynamically computed distribution in a *sample statement*. Conditioning of the model is achieved by fixing the values of selected addresses to observed data.

In Figure 1 on the left, we show how the mixture model

$$B \sim \texttt{Bernoulli}(0.5), \quad z \sim \begin{cases} \texttt{Normal}(-3,1) & \text{if } B = 1, \\ \texttt{Uniform}(1,4) & \text{otherwise.} \end{cases}, \quad y \sim \texttt{Normal}(z,2)$$

may be implemented with stochastic branching in a universal PPL.

```
def disc_mixture():
  B = sample("B",Bernoulli(0.5),
             branching=True)
  if B == 1:
    z = sample("z",Normal(-3,1))
  else:
    z = sample("z",Uniform(1,4))
  sample("y",Normal(z,2),observed=2)
```

```
def model2():
  U = sample("U",Uniform(0,1))
  if U > 0.5:
    z = sample("z1",Normal(-3,1))
  else:
    z = sample("z2",Uniform(1,4))
  sample("y",Normal(z,2),observed=2)
```

Fig. 1. Two mixture models implemented in a universal PPL with stochastic branching. We highlight the annotations required by *prior* DCC implementations to identify SLPs in yellow.

In this work, the domain of probabilistic programs is defined with *traces* – mappings from addresses of random variables to their values. We interpret a probabilistic program as a function p mapping traces to their probability density. The function p is only well-defined for traces that are compatible with executions of the program. The *support* of a probabilistic program is the set of traces which map to a positive density. For instance, for the considered mixture model, we have $p(\{\texttt{B} \mapsto 0, \texttt{z} \mapsto 2.3\}) = \text{pdf}_{\texttt{Bernoulli}(0.5)}(0) \cdot \text{pdf}_{\texttt{Uniform}(1.4)}(2.3) \cdot \text{pdf}_{\texttt{Normal}(2.3,2)}(2) \approx 0.0329$.

The great expressivity of universal PPLs comes at the cost of making automated inference more challenging. The Divide-Conquer-Combine (DCC) approach [55] proposes to solve this problem by splitting up the model support, given by a set of traces $\mathcal{T}$, into an (typically) infinite number of disjoint subsets $\mathcal{T}_k$ such that the support structure on $\mathcal{T}_k$ is static and finite dimensional (essentially isomorphic to some $\mathbb{R}^n$ measurable space). This *divide step* splits the model into multiple sub-models $p_k(\texttt{tr}) = [\texttt{tr} \in \mathcal{T}_k] \cdot p(\texttt{tr})$ such that $p(\texttt{tr}) = \sum_k p_k(\texttt{tr})$. In the context of probabilistic programming, these sub-models are referred to as *straight-line programs* (SLPs) as they are free from universal language features like stochastic control flow and recursion.

For the example mixture model, the two disjoint sub-models $p_1(\texttt{tr}) = [\texttt{tr(B)} = 1] \cdot p(\texttt{tr})$ and $p_2(\texttt{tr}) = [\texttt{tr(B)} \neq 1] \cdot p(\texttt{tr})$ are given by the SLPs below, which return the log-density along with a boolean value indicating whether the trace belongs to the sub-model.

```
def disc_mixture_SLP1(tr):
  lp = 0.0
  lp += Bernoulli(0.5).log_prob(tr["B"])
  lp += Normal(-3,1)).log_prob(tr["z"])
  lp += Normal(z,2).log_prob(2)
  return lp, B == 1
```

```
def disc_mixture_SLP2(tr):
  lp = 0.0
  lp += Bernoulli(0.5).log_prob(tr["B"])
  lp += Uniform(1,4).log_prob(tr["z"])
  lp += Normal(z,2).log_prob(2)
  return lp, B != 1
```

It is desirable for systems implementing the DCC approach to be capable of automatically finding and compiling the associated SLP functions. However, the only existing implementations [42, 55] rely on user-annotations to mark discrete branching variables or require the model to be rewritten

such that the set of sample addresses encountered in a program run uniquely identifies the SLPs when branching depends on continuous variables, see Figure 1. By building Upix on JAX [13], a *tracing just-in-time* (JIT) compiler for Python, we are able to fully automate SLP generation *without* user-annotation, see Section 3.3.

```
def gmm(ys: jax.Array):
  N = ys.shape[0]
  K = sample("K",Poisson(lam-1)) + 1 # influences the shape of w, mus, and vars
  w = sample("w",Dirichlet(jnp.full((K,),delta)))
  mus = sample("mus",Normal(jnp.full((K,),xi),jnp.full((K,),1/jnp.sqrt(kappa))))
  vars = sample("vars",InverseGamma(jnp.full((K,),alpha),jnp.full((K,), beta)))
  zs = sample("zs",Categorical(jax.lax.broadcast(w,(N,))))
  sample("ys", Normal(mus[zs],jnp.sqrt(vars[zs])),observed=ys)
```

Fig. 2. Gaussian Mixture Model implemented in Upix making use of dynamic array shapes.

Furthermore, we consider a previously unrecognised source for stochastic model support in probabilistic programming. To illustrate, we show a Gaussian Mixture Model (GMM) with an unknown number of components `K` in Figure 2. In this program, the shapes of the arrays passed as arguments to the distribution objects of `w`, `mus`, and `vars` are dynamically determined by the value of `K`, thereby inducing a stochastic support whose structure varies with `K`. Our JAX implementation is able to identify the SLPs fully automatically even in this case . In contrast, prior implementations require the annotation of `K` with `branching=True` or modifiying the addresses, e.g. `"w"+str(K)`.

The GMM also demonstrates the benefits and challenges of the conquer and combine steps of DCC. In the *conquer step*, efficient inference algorithms for the static and finite dimensional sub-models are used to approximate $\hat{p}_k \approx p_k = [\mathtt{tr}(\mathtt{K} = k)] \cdot p(\mathtt{tr})$. While easier than inference for the full model, inference for the sub-models is already challenging and black-box approaches as realised in existing DCC implementations perform poorly for complex models. Instead, we propose to use established programmable inference constructs [4, 11, 12, 25] for these sub-problems. For instance, for the GMM, an MCMC approach based on well-known Gibbs kernels [44] can be readily programmed as proposal distributions for a Metropolis-Hastings algorithm. Notably, these Gibbs kernels only exist for the finite sub-models with fixed `K`, not for the full model.

A drawback of splitting the model into multiple sub-problems is the need to allocate computational resources effectively among high-probability SLPs. While Zhou et al. [55] and Reichelt et al. [42] have proposed general resource allocation techniques for DCC, we leave room in Upix to customise resource allocation programmatically.

Finally, in the *combine step* of DCC, the estimates of the sub-models have to be combined in an unbiased manner with $\hat{p} \approx \sum_k w_k \hat{p}_k$. In the original formulation of DCC, the weight $w_k = \hat{Z}_k / \sum_j \hat{Z}_j$ for $\hat{p}_k$ is computed by estimating the *normalisation constants* or *marginal likelihood* $\hat{Z}_k = \int_{\mathcal{T}_k} p(\mathtt{tr}) d\mathtt{tr}$ with a variant of importance sampling. Unfortunately, this integral becomes very challenging for complex models, and we found that general-purpose estimation techniques like importance sampling are insufficient. Again, we propose to customise this estimation programmatically in Upix for which we provide programmable inference constructs. For instance, as we will elaborate in Section 4.3, we compute the SLP weight w_k for the GMM by implementing a *reversible-jump* kernel that allows us to estimate the probability of "switching" between sub-models in the full posterior.

3 The Upix System

3.1 Language

Upix allows the specification of probabilistic models in a universal PPL embedded in Python. Our syntax is adapted from Pyro, with sample statements invoking the abstract function

```
def sample(address:str, distribution:Distribution, [observed:Array]) -> Array
```

where `address` is a unique identifier for a random variable that is distributed according to argument `distribution`, and the `observed` argument is optional for incorporating observed data. A model is given by a Python function annotated with the decorator `@model` and includes sample statements mixed with arbitrary Python code that modifies JAX arrays. Importantly, Python control flow, like if statements and while loops, and array shapes that depend on concrete JAX array values are allowed. Normally, this is disallowed when JIT-compiling a JAX function and leads to a `ConcretizationTypeError`, because array shapes have to be statically known in compilation. For this reason, NumPyro, the JAX implementation of Pyro, also disallows these language features and only supports models with fixed and finite structure.[1] In Section 3.3, we describe how we enable these language constructs with our custom JAX interpreter.

3.2 Abstract Divide-Conquer-Combine

At the highest level, we developed a novel abstraction of the DCC approach shown in Listing 1. The `initialise_active_slps` and `update_active_slps` methods implement the divide step. The former finds SLPs (sub-models) in the initialisation phase and in the update phase the latter retains promising SLPs while discarding less promising SLPs based on the results collected so far. New SLPs may also be instantiated and made active in the update phase. The `run_inference` method implements the conquer step and performs inference for all active SLPs and stores the result. The `estimate_log_weight` is responsible for computing the contribution weight of the SLP posterior to the full model posterior. Together with the `combine` method the combine step of DCC is implemented. In this last step, all results are aggregated with respect to the estimated contribution weights. In Section 4, we will instantiate this abstract routine based on several Bayesian inference approaches to implement existing DCC methods and to formulate novel DCC algorithms.

Listing 1. The divide-conquer-combine approach as abstracted in Upix.

```
class DCC:
  def run(self, model):
    active_slps, inactive_slps, results = [], [], []
    self.initialise_active_slps(model, active_slps, inactive_slps)
    while len(active_slps) > 0:
      for slp in active_slps:
        self.run_inference(slp, results)
        self.estimate_log_weight(slp, results)
      self.update_active_slps(model, results, active_slps, inactive_slps)
    return self.combine(results)
```

3.3 Compiling SLPs

Building on JAX enables us to formulate a simple program transformation that, for the first time, *automatically* identifies and compiles straight-line programs (SLPs) from a probabilistic program. Thus, without the need for annotations, we allow programs to contain control flow that depends on random variables and allow programs to contain distribution objects whose support structure

[1] Note that Reichelt contributed a rudimentary implementation of DCC [55] and SDVI [42] to NumPyro which relies on user annotations for discrete branching variables.

depends on other random variables. First, we transform a model definition according to standard density-based semantics, where sample method calls are replaced with

```
value = observed if observed is not None else tr[address]
lp += distribution.log_prob(value)
return value
```

where `tr:Dict[str,Array]` is our trace data structure whose values are injected at the corresponding addresses if there is no observed data. In addition, we accumulate the probability density in log-space with the variable `lp`. The decorator `@model` transforms a Python function `def f(args:S) -> T` to `model(f): S -> (Dict[str,Array] -> (Float,T))` by replacing sample statements as described above and adding the trace input argument. That is, `model(f)(args)(tr)` runs the program with respect to input trace `tr` and outputs the return value together with its probability density. Typically, the arguments to a probabilistic model `args:S` pass hyper-parameters or observed data and are constant throughout inference. Further, the posterior distribution over the latent random variables is of more interest than the distribution of some return value. For these reasons, we often omit arguments and return values in model definitions in the rest of the paper to ease presentation, but they are supported in Upix.

Note that `model(f)(args)` cannot be JIT-compiled in general as this function may contain control flow and array shapes that depend on `tr`. Transforming the function into multiple straight-line programs by constraining the branching and shape decisions both allows us to split the underling probabilistic model into multiple sub-models with disjoint support as required by DCC and allows us to JIT-compile the SLPs. We achieve this with a tracing-based program transformation, which takes arbitrary functions `def g(args:S) -> T` and produces `trace_decisions(g): S -> (T,Bool)`. This transformation is implemented as a custom JAX interpreter that executes the program with the input arguments `args`, circumvents `ConcretizationTypeError`s by making branching and shape decisions according to the concrete values of `args`, and traces all encountered array operations for subsequent compilation. Thus, a JIT-compilation of `trace_decisions(g)` with respect to `args` produces a straight-line program where control flow and array shapes are fixed by `args` even if the executable is invoked with inputs that would result in different control flow or array shapes in `g`. To keep track of this mismatch, the transformation also adds a boolean return value which

```
@model
def simple():
  U = sample("U",Uniform(0,1))
  if U > 0.5:
    N = 1
  else:
    N = sample("N",Poisson(3))
  for i in range(N):
    sample(f"x_{i}",
      Normal(zeros((N,)),1,))
```

```
def SLP1(tr):
  lp = 0
  lp += Uniform(0,1).log_prob(tr["U"])
  lp += Normal([0],1)).log_prob(tr["x_0"])
  return lp, (tr["U"] > 0.5)

def SLP2(tr):
  lp = 0
  lp += Uniform(0,1).log_prob(tr["U"])
  lp += Poisson(3).log_prob(tr["N"])
  lp += Normal([0,0,0],1).log_prob(tr["x_0"])
  lp += Normal([0,0,0],1).log_prob(tr["x_1"])
  lp += Normal([0,0,0],1).log_prob(tr["x_2"])
  return lp, (tr["U"] <= 0.5) and (tr["N"] == 3)
```

Fig. 3. SLP generation process exemplified: Left: a probabilistic model defined in Upix that exhibits stochastic support. Right: Two example straight-line programs which are equivalent to programs obtained with the JAX transformations described in Section 3.3.

indicates whether or not the input matches the decisions of the original `args`. We call `args` the *decision representative* of the transformed program.

In the context of DCC, we have `g = model(f)(model_args): Dict[str,Array] -> Float` and a trace `tr:Dict[str,Array]` is the decision representative of an SLP. The compiled function `trace_decisions(g)` returns the log-probability density of the sub-model encoded by the SLP and a boolean value `b` which is true if the input trace would result in the same SLP as the decision representative trace `tr` that the function was compiled with respect to. Thus, the input trace lies in the support of the sub-model if and only if `(lp > -inf) and b`. In Figure 3, we illustrated this process by giving two example SLP generations for a model which exhibits stochastic support. Note that the branching condition, the set of executed sample statements, and the dimensionality of the random variables with addresses `f"{x_i}"` depend on the input traces. Unlike existing implementations of DCC, our program transformation automatically compiles SLPs without the need to annotate that the support structure depends on `U` and `N`.

Note that the described JAX transformations were implemented to be compatible with other JAX transformations. This means that, for instance, we can use `jax.vmap` to vectorise the log-probability density computation or `jax.grad` to differentiate the log-probability density with respect to the trace. Both of these transformations enable an efficient implemention of gradient-based inference algorithms like HMC [5] or ADVI [23].

3.4 Programmable Inference

In principle, every step of DCC is customisable in Upix. We provide several instantiations of the framework for the most popular inference algorithms including Markov chain Monte Carlo (MCMC), sequential Monte Carlo (SMC), and Variational Inference (VI) algorithms. These instantiations are organised hierarchically as shown in Figure 4 and provide default implementations of the individual steps of DCC as well as convenience constructors to ease programmable inference based on established principles [6, 12, 25, 38]. In Section 4, we will describe these instantiations in detail.

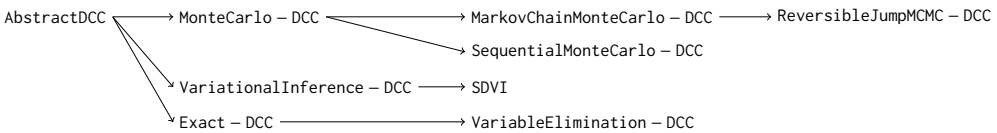

Fig. 4. Hierarchical organisation of the instantiations of DCC algorithms in Upix.

Unlike existing universal probabilistic programming systems which allow the user to customise inference only for the full model, Upix enables the user to customise inference for each sub-model encoded by an SLP. Crucially, at the SLP level, the trace data structure `tr:Dict[str,Array]` has fixed array shapes which allows the compilation of the associated (log-)probability density function, as discussed in Section 3.3. This enables us to also compile and execute full inference routines programmed for each SLP with the provided convenience constructors and run them on CPUs, GPUs, or TPUs.

Furthermore, note that at the SLP level, the log-probability density function can be "flattened" to the form `log_p: Array -> Float` by concatenating the values of the input trace in fixed order. Mathematically, the log-probability density function of an SLP is a function $\mathbb{R}^n \to \mathbb{R}$ with fixed domain dimension n. Most of the inference algorithms implemented in Upix, which form the building blocks of the programmable inference constructs, are implemented based solely on a JAX-compiled log-probability density function. This opens up opportunities for interoperability with existing inference systems. For instance, BlackJAX [7] is a Bayesian inference framework built around the

log-probability density. But also parts of the inference machinery of TensorflowProbability [24] or NumPyro [41] could be integrated in Upix as new building blocks for programmable inference.

Lastly, while we provide default implementations for the `initialise_active_slps` method and for the `update_active_slps` method, it is generally beneficial to adapt them to the model at hand. We give examples in Section 4. Effectively, this gives rise to a two-layer approach to programmable inference. First, inference routines may be customised at the SLP level in the conquer step as described. Second, on the meta-inference level, users may customise how the space of SLPs is explored and how computational resources are allocated to them.

3.5 Parallelisation

The DCC approach together with JAX compilation allow for many opportunities to parallelise inference across computational devices.

First, Upix implements parallelisation techniques for the inference run and the estimation of the weight of a each SLP. In this parallelisation approach, computations are scheduled sequentially in lines 6-8 of Listing 1.

Vectorisation. The vectorisation transform of JAX called `jax.vmap` was used to implement inference machinery as numerical computations on multi-dimensional arrays. Such computations can be efficiently run on accelerator hardware designed for single-instruction multiple-data (SIMD) use cases like GPUs and TPUs. This parallelisation technique is recommended for machines with a single GPU.

Sharded Arrays. On top of vectorisation, JAX allows the distribution of computations on multi-dimensional arrays among several accelerators with `jax.pmap` and `jax.shard_map`. Thus, we can run inference and weight estimation for each SLP on multiple GPUs and TPUs. This is recommended for inference and weight estimation workloads which are embarrassingly parallel. In Section 5, we use this technique to scale MCMC to thousands of chains, VI to thousands of gradient estimates per step, and SMC to thousands of particles.

Second, Upix implements parallelisation techniques to run inference and weight estimation for multiple SLPs at the same time.

Multiple accelerator devices. If the inference and weight estimation workloads are not embarrassingly parallel, we may distribute the full workloads among GPUs and TPUs which work independently in a multi-processing fashion.

Multiple CPU cores. By default, JAX does not allow one to dispatch multiple computations to specific CPU cores in parallel. As some users of Upix may not have access to machines with multiple accelerators, we decided to implement a workaround to run inference and weight estimation of multiple SLPs with multi-processing. To achieve this, we launch multiple JAX processes pinned to single CPU cores and send them workloads via the serialisation functionality built into JAX.

4 DCC Algorithms Instantiated with Upix

4.1 Markov Chain Monte Carlo DCC

As the first application of Upix, we adapt the DCC algorithm of Zhou et al. [55], which is built around MCMC inference. MCMC algorithms build a chain of samples x_i which are perturbed by a kernel $x_{i+1} \sim k(.|x_i)$ which leaves the posterior distribution invariant. This property implies that with a sufficiently long chain, a sample from the posterior distribution is approximated. Such kernels can be constructed even if the posterior density function is only known up to a multiplicative

constant, e.g. if the joint probability density over latent and observed variables can be computed. Notable examples are the Metropolis-Hastings [34], the HMC [5], or the NUTS kernel [20].

The MCMC DCC algorithm is implemented in Upix as sketched in Listing 2. It is realised as a sub-class of `MonteCarloDCC` our abstract implementation of DCC which approximates the posterior with a weighted set of traces. The initialisation of active SLPs is done by sampling a trace from the model prior and checking if the trace is not in the support of any SLP found so far. As proposed by Zhou et al. [55], active SLPs are updated by running a *global* random walk Metropolis-Hastings algorithm on the full model, where traces may change with respect to their set of addresses and array shapes. For this reason, this global algorithm cannot be JIT-compiled and serves solely for updating the set of active SLPs. In the `MarkovChainMonteCarloDCC` class the normalisation constant of an SLP is estimated via importance sampling as in Zhou et al. [55]. To ease programmable inference for MCMC-based DCC, we provide convenience constructs adapted from prior work [12, 14] to define MCMC kernels for each SLP in the `get_MCMC_inference_regime` method, which are then used in the MCMC routine launched in the `run_inference` method.

Listing 2. Implementation sketch of Markov Chain Monte Carlo DCC in Upix.

```
class MonteCarloDCC(AbstractDCC):
  def initialise_active_slps(self, model, active_slps, inactive_slps):
    # draw traces from prior
  def update_active_slps(self, model, results, active_slps, inactive_slps):
    # global random walk updates
  def combine(self, results):
    # combine inference results as set of weighted traces
class MarkovChainMonteCarloDCC(MonteCarloDCC):
  def estimate_log_weight(self, slp, results):
    # estimate log-weight with importance sampling
  def get_MCMC_inference_regime(self, slp):
    # provided by the user
  def run_inference(self, slp, results):
    # run MCMC routine based on self.get_MCMC_inference_regime(slp)
```

Our programmable inference approach to MCMC-based DCC is best illustrated with an example. Mak et al. [28] introduced the Pedestrian model shown in Figure 5 to motivate NP-DHMC their novel non-parametric variant of discontinuous HMC [39] which can handle both discontinuities in the density function and stochastic support. The Pedestrian model is challenging, because the while loop condition and the observation depend on all random variables sampled so far. In the spirit of DCC, we approximate the posterior of this model by running discontinuous HMC individually for each SLP and combine the results later. Upix automatically splits up this program into SLP_k, $k \in \mathbb{N} \setminus \{0\}$, where SLP_k is the program that executes the while loop for k iterations.

On the right in Figure 5, we show how users may customise MCMC kernels for the Pedestrian model with the provided constructs. They can choose from a set of inference algorithms like Metropolis-Hastings `MH` or discontinuous HMC `DHMC`, customise the proposal distributions and hyper-parameters, and apply them to a subset of variables which may be specified with regular expressions.

To compare our approach, we ran the NP-DHMC PyTorch implementation of Mak et al. [28] with 8 chains of 1 000 samples configured with their best scoring hyper-parameter values: 50 HMC steps with step-size 0.1. Running each chain in multiple processes in parallel takes 121 seconds on a 10 core M2 Pro machine. In less than 25 seconds (our reported runtimes always include JAX compile

```
@model
def pedestrian():
  position = sample("start",Uniform(0,3))
  distance = 0.
  t = 0
  while (position > 0) & (distance < 10):
    t += 1
    step = sample(f"step_{t}",Uniform(-1,1))
    position += step
    distance += jax.lax.abs(step)
  sample("obs",
    Normal(distance,0.1),observed=1.1)
```

```
regime = MCMCSteps(
  MCMCStep(
    Variables("start"),
    MH(lambda _: Uniform(0,3))
  ),
  MCMCStep(
    Variables(r"step_\d+"),
    DHMC(50,0.05,0.15)
  )
)
```

Fig. 5. Left: Pedestrian model [28] in UPIX. Right: MCMC routine specified with provided constructs.

time), we can run 8 chains of 25 000 samples in UPIX-MCMC-DCC with the same configuration for DHMC. This runs inference and combines results for SLP_1 to SLP_6. It was determined that all other SLPs do not contribute enough to the posterior to warrant an inference run. For this, we customised `initialise_active_slps` to iterate over the SLPs ordered by number of steps and stop adding SLP_k to `active_slps` once the normalisation constant of SLP_k is estimated to be a factor of 10^3 smaller than the highest weighted SLP seen so far.

Thus, with inference routines JIT-compiled for each SLP we are able to take 720 times more samples per second than the NP-DHMC algorithm which has to move between SLPs in a non-compiled program. At this point, we emphasise that the implementation of Mak et al. [28] was not optimised for speed. Nevertheless, in Figure 6 we show the qualitative difference between the achieved approximations to the true posterior. The approximation quality is quantified with $L_\infty(\hat{F}, F) = \max_x |\hat{F}(x) - F(x)|$, where F is the ground truth posterior cumulative distribution function over the "start" variable x estimated with 10^{12} importance samples and $\hat{F}$ is the approximation obtained with the MCMC approaches. NP-DHMC achieves $L_\infty \approx 0.02372$ while our UPIX implementation achieves $L_\infty \approx 0.00133$.

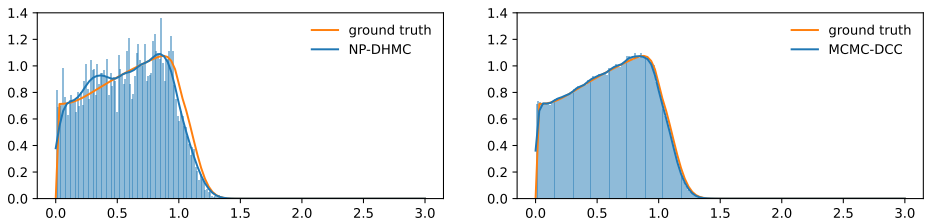

Fig. 6. Approximations to the true posterior of the Pedestrian model (estimated with 10^{12} importance samples) with $8 \cdot 1\,000$ samples from NP-DHMC versus $6 \cdot 8 \cdot 25\,000$ samples from MCMC-DCC as 100 bin histograms.

4.2 Variational Inference DCC: SDVI

In Variational Inference the posterior distribution is approximated by optimising the parameters ϕ of a *variational distribution* q_ϕ to minimise the Kullback–Leibler divergence to the posterior. In practice, the so-called *evidence-lower-bound* (ELBO) $\mathcal{L}(\phi) = \mathbb{E}_{z\sim q_\phi}\left[\log p(z) - \log q_\phi(z)\right]$ is maximised which constitutes an equivalent but more practical optimisation objective. This optimisation is achieved

via gradient ascent and typically implemented on top of automatic differentiation machinery in probabilistic programming systems. In this case, the approach is called *Automatic Differentiation Variational Inference* (ADVI) [23].

Reichelt et al. [42] proved that when we fit a local variational distribution $q^k_{\phi_k}$ to the posterior of each individual sub-model encoded by an SLP, then the mixture of the local variational distributions, where each $q^k_{\phi_k}$ is weighted proportional to their local ELBO, $w_k \propto \exp(\mathcal{L}_k(\phi_k))$, is the optimal approximation to the full model. This enabled their variational inference based DCC approach.

This approach is implemented in Upix as sketched in Listing 3. To ease programmable inference we allow the user to specify a variational distribution as a so-called *guide program* following Pyro's approach [6]. A guide program in Upix is a Python function made up from sample statements `sample(...)` and parameter declarations `param(...)`. The distributions in sample statements are parameterised with respect to the declared parameters and we can compute the ELBO gradient via JAX's automated differentiation machinery. In addition, we also make auto-guides available to the user that approximate the posterior with a parameterised multivariate Gaussian distribution.

Listing 3. Implementation sketch of Variational Inference DCC and SDVI in Upix.

```
class VariationalInferenceDCC(AbstractDCC):
  def initialise_active_slps(self, model, active_slps, inactive_slps):
    # draw traces from prior
  def get_guide(self, slp):
    # provided by the user
  def run_inference(self, slp, results):
    # run ADVI routine based on self.get_guide(slp)
  def estimate_log_weight(self, slp, results):
    # estimate log-weight with local ELBO
  def combine(self, results):
    # combine inference results as a mixture of variational distributions
class SDVI(VariationalInferenceDCC):
  def update_active_slps(self, model, results, active_slps, inactive_slps):
    # use SuccessiveHalving strategy by Reichelt et al. 2022
```

Reichelt et al. [42] also proposed a *Successive Halving* strategy to allocate computational resources to the most promising SLPs. In this strategy, a fixed computational budget is divided among SLPs for a number of phases. After each phase, half of the SLPs are discarded, while the other half is optimised with double the number of ADVI iterations. Their full method is called *Support Decomposition Variational Inference* (SDVI) which we also implemented in Upix via the `update_active_slps` method, see Listing 3.

We compare the runtime of their Pyro implementation of SDVI against our implementation in Upix.[2] As a benchmark, we implement a probabilistic program to infer the kernel structure of a Gaussian Process model for airline passenger data. The kernel structure is given by the probabilistic context-free grammar $\mathsf{K} ::= \mathsf{SE}|\mathsf{RQ}|\mathsf{PER}|\mathsf{LIN}|\mathsf{K}+\mathsf{K}|\mathsf{K}\times\mathsf{K}$, where we consider the squared-exponential, rational-quadratic, periodic, and linear primitive kernels. This model exhibits stochastic support as the number of base kernels is not fixed and unbounded. We have a unique SLP for each possible kernel structure. The prior over the kernel grammar is implemented as a recursive Python function. Like Reichel et al., we also use an auto-guide for each SLP. Implementation details can be found in the replication package [1].

[2]Note that Reichelt also contributed a NumPyro version of SDVI, which presumably would run faster. However, this implementation is rudimentary and does not fully replicate their SDVI approach [42].

The Pyro version with 10 parallel processes takes 18 minutes and 20 seconds on a M2 Pro to run ADVI on initially 110 random SLPs reduced to the 10 best SLPs over 5 phases with a budget of 10^6 total ADVI steps. Our equivalent Upix implementation takes only 2 minutes and 24 seconds to go from 98 random SLPs to the 10 best, again over 5 phases. Here, we use the *Mutliple CPU cores* parallelisation strategy with 10 parallel JAX processes, see Section 3.5. As computing a ground truth posterior for the Gaussian process model is infeasible, we cannot compare the approximation quality of both approaches. However, as we reproduced the implementation of SDVI in Upix exactly, a runtime comparison between the approaches is sufficient.

4.3 Reversible Jump / Involutive MCMC DCC

Reversible Jump MCMC (RJMCMC) algorithms [17], which belong to the class of *involutive MCMC* algorithms [38], were developed for models that can be decomposed into enumerable sub-models of varying dimension. Metropolis-Hastings moves are designed to switch between these sub-models during inference. As suggested by the name, these jumps have to be designed in a reversible manner: for each jump move there has to be a move which reverses it, establishing a bijection. In this section, we describe our novel DCC algorithm based on the RJMCMC principle.

This approach is best explained by example. In one of the earliest formulations Richardson and Green [44] designed reversible jumps for a Gaussian mixture model (GMM) with unknown number of components. Mathematically, we model the number of components $K \sim \texttt{Poisson}(2) + 1$ and the membership probability as $w \sim \texttt{Dirichlet}(\delta)$. Data is modelled with $z_i \sim \texttt{Categorical}(w)$ and $x_i \sim \texttt{Normal}(\mu_{z_i}, \sigma^2_{z_i})$, where $\mu_k \sim \texttt{Normal}(0, \kappa^{-2})$ and $\sigma_k \sim \texttt{InverseGamma}(\alpha, \beta)$ are the center and deviation around the center for component k. We have implemented this model in Upix as shown in Figure 2. For this model, the jumps are given by 1) splitting one component into two components and 2) merging two components to one component. These jumps are designed to be inverses of each another on an extended variable space. The RJMCMC inference algorithm for the GMM alternates between the reversible jumps, changing the number of components, and well-known Gibbs moves to draw samples from the posterior of each sub-model. These Gibbs moves are readily implemented with the MCMC programmable inference constructs provided by Upix.

At first, we implemented inference for the GMM following the MCMC-DCC approach as described in Section 4.1 which eliminates the need to jump between sub-models as every SLP corresponds to a sub-model with a certain fixed number of components. However, we were not able to accurately estimate the normalisation constants needed to combine the results with importance sampling methods as proposed by Zhou et al. [55] or with other common approaches for marginal likelihood estimation like Chib's method or sequential Monte Carlo methods [26].

Subsequently, we realised that for the combine step to be sound in DCC, the absolute normalisation constants of each SLP are not required. Instead, it is sufficient to weigh an SLP only *relative* to the other SLPs. Even more, this relative estimation can be built around reversible jumps. To see this, let $\mathcal{D}$ be the observed data and let $p(.|i, \mathcal{D})$ denote the d_i dimensional posterior of sub-model i and $k(j, y|i, x)$ a reversible jump kernel that moves from sub-model i to sub-model j. We observe following relationship between normalisation constants $p(i|\mathcal{D})$ and $p(j|\mathcal{D})$:

$$
\begin{aligned}
Z_{j \leftarrow i} &\coloneqq \int_{\mathbb{R}^{d_j}} \int_{\mathbb{R}^{d_i}} k(j, y|i, x)\, p(x|i, \mathcal{D})\, dxdy = \\
&= \frac{1}{p(i|\mathcal{D})} \int_{\mathbb{R}^{d_j}} \int_{\mathbb{R}^{d_i}} k(j, y|i, x)\, p(i, x|\mathcal{D})\, dxdy = \\
&= \frac{1}{p(i|\mathcal{D})} \int_{\mathbb{R}^{d_i}} \int_{\mathbb{R}^{d_j}} k(i, x|j, y)\, p(j, y|\mathcal{D})\, dydx = \frac{p(j|\mathcal{D})}{p(i|\mathcal{D})} Z_{i \leftarrow j}
\end{aligned}
$$

Above, the second to last equation follows from the fact that involutive MCMC kernels such as RJMCMC kernels satisfy *detailed balance* [38]. This gives rise to an infinite system of equations

$$Z_i = p(i|\mathcal{D}) = Z_j \frac{Z_{i \leftarrow j}}{Z_{j \leftarrow i}}, \quad i \in \mathbb{N}, j \in \mathbb{N}. \tag{1}$$

This system can be approximated by only taking into account the SLPs for which inference was performed in DCC. The system is under-determined, so we may arbitrarily set $Z_0 = 1$. The quantities $Z_{j \leftarrow i}$ may be approximated through sampling as follows: First, take $x \sim p(.|i, \mathcal{D})$, where the posterior $p(.|i, \mathcal{D})$ is approximated by running inference for SLP_i. Then, we simulate reversible jumps $(j, y) \sim k(.|i, x)$ and take $\hat{Z}_{j \leftarrow i}$ as the empirical marginal distribution over j. Crucially, this estimation can often be JIT-compiled as both SLP_i and SLP_j have static support structure.

To ease programmable inference, a language for declaring involutions and computing acceptance rates of reversible jumps via automatic differentiation was implemented following Cusumano-Towner et al. [11]. The `estimate_log_weight` method builds the system of equation (1) based on the user-provided reversible jumps and solves it numerically to estimate the SLP weights.

To evaluate our approach, we compare against the described RJMCMC algorithm for the GMM implemented in Gen adapted from Matheos et al. [33]. Running 8 MCMC chains of length 25 000 in a multi-threaded fashion takes 95 seconds (with $\approx$ 13 seconds Julia JIT-compile time). This results in an approximation to the posterior with error $L_\infty(\hat{F}, F) = \max_k |\hat{F}(k) - F(k)| \approx 0.00759$, where F is the ground truth cumulative distribution function over the number of components estimated by collecting a total of 10^7 samples and $\hat{F}$ is the approximation obtained by the $8 \cdot 25\,000$ samples. In this run the highest number of components encountered was 9. Our approach takes 91 seconds ($\approx$ 47 seconds compile-time) to run 8 chains and 25 000 seconds for 11 SLPs corresponding to component numbers 1 to 11. For this, we customised `update_active_slps` to run inference on SLPs ordered by the compontent count and SLP_{k+1} is made active only if the split probability of SLP_k exceeds the threshold 0.01. In approximately the same time, Upix produces 10 times more samples compared to Gen, which results in a better estimate $L_\infty(\hat{F}, F) \approx 0.00401$.

4.4 Sequential Monte Carlo DCC

Although the sequential Monte Carlo (SMC) [10] inference algorithm class can be quite naturally instantiated as a DCC algorithm, to the best of our knowledge, we are the first to do so. Different to MCMC and VI, SMC algorithms propagate a set of samples – so-called *particles* – once through the probabilistic program. Observed data is added to the inference process iteratively. After each addition of data, the particles are re-weighted relative to their likelihood. Then, particles may be resampled according to their weights effectively discarding low-probability particles and duplicating high-probability particles. Optionally, a *rejuvenation* MCMC kernel may be applied to all particles after resampling to make them more diverse. At the end, the set of weighted particles constitutes an approximation to the posterior distribution of the model. As a convenient by-product SMC algorithms also provide an estimate of the marginal likelihood Z.

Thus, an SMC-DCC algorithm is implemented in Upix as sketched in Listing 4, where we re-use the marginal likelihood estimate from the inference run in the combine step. We enable the user to easily specify the schedule with which data is introduced to the inference process in the `get_SMC_data_schedule` method. For instance, this could be one data point at a time or a batch of data points at a time. To run SMC efficiently, many probabilistic programming systems implement machinery to incrementally execute the program and add data this way. A similar approach in Upix would lead to a lot of compilation overhead as the log-probability density function would have to be compiled for every data increment. Instead, we decided to always execute the full probabilistic program, but pass a mask as argument that determines which data points contribute

to the log-probability density. This way the density computation has to be compiled only once per SLP in SMC. In addition to making the data schedule programmable, we also provide the option to specify a rejuvenation MCMC kernel with the same constructs as in Section 4.1. We also provide many options to customise the resampling scheme in SMC like the resampling method (multinomial, stratified, and systematic) and resampling time (before or after rejuvenation).

Listing 4. Implementation sketch of Sequential Monte Carlo DCC in Upix.

```
class SequentialMonteCarloDCC(MonteCarloDCC):
  def get_SMC_data_schedule(self, slp):
    # provided by the user
  def get_SMC_rejuvenation_regime(self, slp):
    # provided by the user
  def run_inference(self, slp, results):
    # run SMC routine based on rejuvenation kernel and data schedule
  def estimate_log_weight(self, slp, results):
    # estimate log-weight with estimate from SMC inference run
```

To test this approach, we again consider the Gaussian Process model from Section 4.2, but with slighlty different kernel grammar $\mathsf{K} ::= \mathsf{GE}|\mathsf{PER}|\mathsf{LIN}|\mathsf{K}+\mathsf{K}|\mathsf{K}\times\mathsf{K}$, where GE is a gamma-exponential primitive kernel. At the start we make the three primitive kernels active SLPs. On each of four subsequent DCC iterations, we run inference for five active SLPs which are obtained by perturbing the previous set of active SLPs with global random walk updates as in Listing 2. We use 100 SMC particles, a HMC rejuvenation kernel, and we split the airline passenger data set in 10 sub-sets. On a M2 Pro this takes around 7 minutes and 35 seconds to complete.

This example is based on Saad et al. [46], who designed an SMC approach with involutive MCMC rejuvenation moves for time series structure discovery and implemented it in the Julia package `AutoGP.jl`. We configured our Upix-SMC-DCC instantiation to be as close to their implementation as possible with the difference that we do not use involutive MCMC rejuvenation moves as our approach does not move between sub-models during inference. On the same hardware, `AutoGP.jl` with 100 particles takes around 8 minutes and 50 seconds to finish. In this time, the 100 particles are propagated only once through the full model, changing the structure of the Gaussian process kernel many times. In contrast, our approach propagates 100 particles through a total of 23 SLPs, keeping the structure fixed. The former has the advantage of exploring more kernel structures, while the latter performs inference and marginal likelihood estimation more accurately for a smaller set of kernel structures.

This comparison serves only to put the runtime of our approach into perspective. Evaluating the approximation quality of both approaches is difficult due to the infeasibility of computing a ground truth and thus, beyond the scope of this paper. We refrain from comparing the approaches by assessing prediction quality on a hold-out test set as others have done, because even if an approach has better prediction quality this may not be due to better posterior approximations – the primary goal of Bayesian inference.

4.5 Variable Elimination DCC

In the conquer step of DCC, we are not limited to approximate inference algorithms which we have considered up until this point. In this section, we present a novel instantiation of the DCC framework which performs inference for discrete models with stochastic support by exactly solving the sub-models which have static structure.

It is well known that discrete probabilistic models with a fixed finite number of random variables can be solved exactly [22]. This can be achieved by naively enumerating the model support and computing the marginal likelihood. However, as this becomes infeasible for larger models, more sophisticated methods like *Variable Elimination* (VE) or Belief Propagation have to be used [22]. In variable elimination, the probability density function of a model is given in factorised form $p(X) = \prod_{k=1}^{K} f_k(X_k)$, where $X_k \subseteq X$ are subsets of variables. Typically, the factors have the form of conditional probability tables, i.e. $p(x|\text{parents}(x))$. This representation allows the efficient marginalisation of a variable x by building the factor product of all factors to which the variable x belongs, and then summing out x. This new factor $f'(X) = \sum_x \prod_{k:\, x \in X_k} f_k(X_k)$ replaces all the factors used in the product and gives rise to a new factorisation without x. This elimination process can be repeated until only a set of *query* variables remains or until all variables are eliminated such that the result of the computation is the normalisation constant.

We have implemented a DCC version of variable elimination in Upix as follows. In the method `get_query_variables`, the user may specify query variables. As the efficiency of variable elimination heavily depends on the elimination order, we enable the user to customise it with a `get_elimination_order` method, but we also provide a strong default ordering. We have implemented a JAX interpreter which automatically constructs the conditional probability table for each variable in the sub-model, thus setting up the factorisation. By default, this is done one factor at a time, which may cause a lot of compilation overhead. To remedy this, we give the user the option to specify sets of variables for which the factors can be computed in one pass in a `get_factors` method. Finally, variable elimination is performed in `run_inference` for each SLP and we can use the exact normalisation constant computed as a by-product as weight in `estimate_log_weight`.

To test the variable elimination DCC approach, we consider the urn model of Milch et al. [36]. In this model, we have an urn with an unknown number of balls $N \sim \texttt{Poisson}(6)$ each of which has equal prior probability of being black or white. We repeatedly draw a ball, observe its color with 80% accuracy, and put it back. With posterior inference we aim to answer: Given that we observed 5 white and 5 black balls, how many balls are inside the urn? As the number of balls is unknown, the support structure of this model is stochastic and cannot be directly solved with exact inference. With our Upix implementation it takes 13 seconds to compute the factorisation and the exact solution for 1 to 20 balls (64 seconds with unoptimised factor computation). In this implementation, we customised SLP initialisation to be performed in order of the number of balls. This gives a maximum absolute error of $1.315 \cdot 10^{-6}$ to the true posterior probabilities mass function over N computed analytically. For reference, producing 10^7 importance samples in BLOG [35, 54], a system designed for such models, takes 27 seconds and yields an error of $5.260 \cdot 10^{-4}$.

5 Scaling Experiments

So far, we have demonstrated the capabilities of Upix only on consumer-grade hardware like the M2 pro CPU. In comparison to existing probabilistic inference implementations, we observed that Upix enables faster inference which results in better approximations as summarised in Table 1. In this section, we shift our attention to large-scale computational hardware and evaluate the capabilities of Upix on machines with up to 64 CPU cores and 8 GPU devices.

First, we note that DCC algorithms can be straightforwardly parallelised by running inference for multiple SLPs in a multi-processing fashion on multiple CPU cores or multiple accelerator devices. This parallelisation method is expected to scale perfectly for models which require the exploration of many SLPs. Although this parallelisation method is supported in Upix, conducting an experiment to evaluate it would offer limited value. Instead, we examine how inference algorithms executed on individual SLPs may benefit from the use of multiple accelerators. Note that since models with

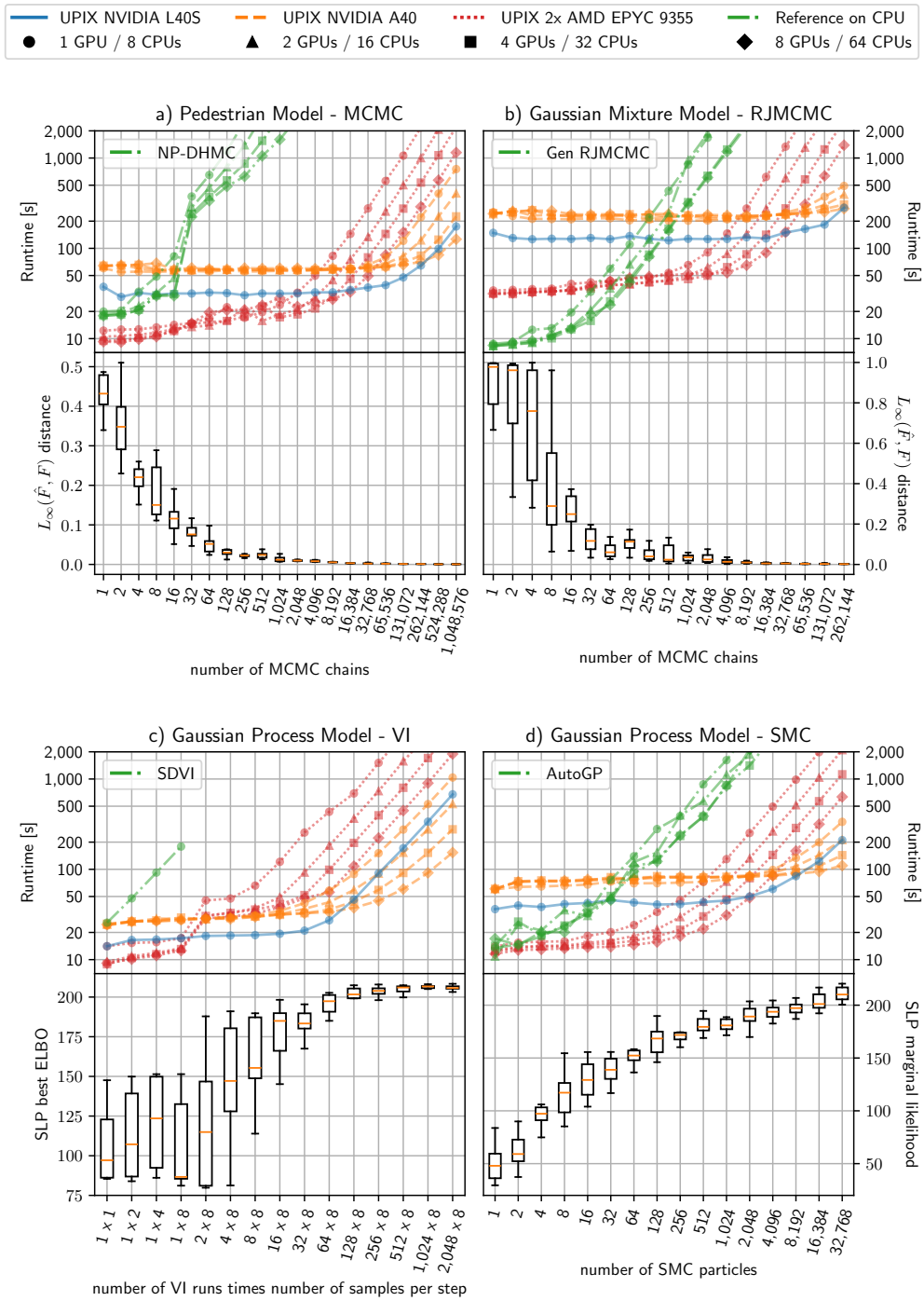

Fig. 7. Scaling experiments. Top plots: runtime of Upix DCC algorithms and reference implementations on various hardware. Bottom plots: quality of posterior approximation. For each model, the same set of SLPs was used for all measurements to ensure a fair comparison.

Table 1. Comparison of DCC algorithms implemented in Upix modelled after baseline algorithms and executed on consumer hardware (M2 Pro CPU).

Model	Method	Details	Runtime	L_∞
Pedestrian	NP-DHMC [28]	$8 \cdot 1\,000$ samples	2m 01s	0.02372
	Upix MCMC-DCC	$6 \cdot 8 \cdot 25\,000$ samples	**25s**	**0.00133**
GP	SDVI [42]	10^6 step budget, 5 phases, 110 SLPs	18m 20s	-
	Upix VI-DCC	10^6 step budget, 5 phases, 98 SLPs	**2m 24s**	-
GMM	Gen [33]	$8 \cdot 25\,000$ samples	1m 35s	0.00759
	Upix RJMCMC-DCC	$11 \cdot 8 \cdot 25\,000$ samples	1m 31s	**0.00401**
GP	AutoGP [46]	100 particles, propagated once	7m 35s	-
	Upix SMC-DCC	100 particles, 23 SLPs	8m 50s	-
Urn	BLOG (Swift) [54]	10^7 importance samples	27s	$5.3 \cdot 10^{-4}$
	Upix VE-DCC	20 SLPs solved exactly	**13s**	$\mathbf{1.3 \cdot 10^{-6}}$

static support structure are also expressible in Upix and yield a single SLP, the experimental results are likewise applicable to this class of models. Figure 7 shows the complete experiment results which we discuss in detail in the following sections.

5.1 Scaling MCMC

In MCMC algorithms, each sample depends on the predecessor in the chain making them inherently sequential. Parallelisation can only be achieved when running multiple, typically 2-4, long chains at once. Recently, the many-short-chain approach has gained more attention [24, 49], where instead of approximating the posterior with long auto-correlated chains, it is approximated with thousands of short chains, sometimes using only their independent end-points. As MCMC algorithms are only asymptotically correct, it remains an open question how short the chains may be [31, 32].

We test this approach in Upix by running MCMC-DCC for the Pedestrian model with up to $2^{20} = 1\,048\,576$ chains of 256 samples each and RJMCMC-DCC for the GMM model with up to $2^{18} = 262\,144$ chains of 2028 samples each, using only their endpoints in the final approximation. We ran DCC ten times with different RNG seeds and make the runs comparable by using a fixed set of 8 SLPs. We show the L_∞ distance to the ground truth posterior in Figure 7 a) and b) on the bottom. As expected, the approximation quality improves with increasing number of chains.

On the top plots, we also report runtime measurements on various hardware. We observe close to perfect scaling: doubling the number of chains doubles the runtime, doubling the number of CPU cores / GPU devices halves the runtime. Due to the longer compile time, GPUs only outperform CPUs when the number of chains exceeds $2^{16} = 65\,536$. Only these large workloads benefit form using multiple GPUs. We note that the newer generation NVIDIA L40s GPUs compile JAX programs faster than the A40 GPUs, but unfortunately, we ran into device communication issues for multiple L40s devices and could not complete the experiments. We also report runtimes on CPU for the NP-DHMC and RJMCMC Gen implementation after which our DCC algorithms were modelled.

5.2 Scaling Variational Inference

Although Variational Inference has been scaled up to very large datasets [19], running VI on accelerators in an embarrassingly parallel way for smaller datasets has been less explored. At first glance, the parameter L, which controls the number samples taken to estimate the ELBO gradient in each step, seems like a good candidate for scaling: more accurate gradient estimates equals faster convergence. While this is true, fast convergence to a particular local maximum is often

less desirable than having higher variance gradient estimates which help escape bad local maxima. We demonstrate this in Figure 8, where it can be seen that running VI eight times with $L = 8$ and picking the best run finds a better maxima compared to running VI once with $L = 64$.

Motivated by this finding, we examine this *multi-run VI* approach at large scale in Figure 7 c). In the lower plot, we show VI results for the Gaussian process SLP corresponding to kernel $(\texttt{PER} + \texttt{RQ}) \times \texttt{LIN}$, where we fix $L = 8$ and increase the number of parallel VI runs up to 2048. We performed 10 experiment repetitions and visualise the best ELBO achieved as boxplots. Note that in the VI run itself the ELBO is estimated with $L = 8$ samples, but for the plot we computed the ELBOs more accurately with 10 000 samples. It can be seen that with increasing number of runs better ELBOs are achieved more consistently.

As before, we measured the runtime of our multi-run SDVI approach with a fixed set of 8 SLPs on various hardware, see the upper plot in Figure 7 c). The scaling is again close to perfect. Notably, it is beneficial to use GPUs already for 2 parallel runs. For reference, we also measure the runtime of the PyTorch SDVI implementation [42] on CPU. This implementation does not support multiple parallel VI runs and does not implement the gradient estimate in vectorised fashion. Thus, we only show data for single runs and 8 CPUs, because we found that increasing the number of CPUs does not lower runtime. However, we reiterate that this SDVI implementation does support running inference for multiple SLPs in parallel in multiple processes.

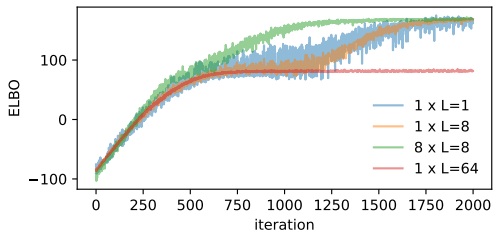

Fig. 8. ELBO curves for Variational Inference runs for Gaussian Process SLP $(\texttt{PER}+\texttt{RQ})\times\texttt{LIN}$ with various L.

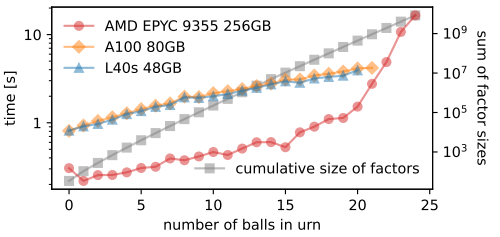

Fig. 9. Cumulative size of factors versus inference runtime of variable elimination for the Urn model.

5.3 Scaling Sequential Monte Carlo

It is a well-known fact that increasing the number of particles in SMC improves approximation quality [10]. We confirmed this by running SMC 10 times for the Gaussian process SLP corresponding to kernel $(\texttt{PER}+\texttt{RQ})\times\texttt{LIN}$ for up to $2^{15} = 32\,768$ particles. As expected and shown in Figure 7 d), the marginal likelihood (normalisation constant) estimates increase and exhibit less variance with increasing number of particles.

The runtime of SMC-DCC measured for a fixed set of 8 SLPs scales close to perfectly again, as can be seen on the top plot in Figure 7 d). GPUs are preferable to CPUs with number of particles exceeding $2^{12} = 4\,096$. The runtime of AutoGP scales similarly to SMC-DCC in Upix on CPUs. We have included these runtimes solely to have a reference for the scaling properties of Upix. Although our SMC-DCC algorithm was inspired by AutoGP's methodology, the reported runtimes should not be interpreted as directly comparable, since the two approaches differ (see Section 4.4).

5.4 Scaling Variable Elimination

Variable elimination for discrete models is performed through repeated factor products. The product of factor f_1 with variables X_1 and factor f_2 with variables X_2 is implemented in log-space as sum of their tables given by arrays, where shared variables $x \in X_1 \cap X_2$ are placed at the same axis and

axes corresponding to variables $x \notin X_1 \cap X_2$ are broadcasted. For models with high dimensional support this summation involves large arrays which can be efficiently computed on GPUs.

However, we found that variable elimination, tested on the Urn model, is memory bound. As can be seen in Figure 9, the size of the involved factors, and thus required memory, scales exponentially with N – the number of balls in the urn. This leads to out-of-memory errors at $N = 22$ for the 48GB A40 and L40s GPUs and at $N = 23$ for the 80GB A100 GPU. At these factor sizes, the computation time on GPU still exceeds the runtime of a CPU. In principle, we estimate that we could push N up to 24 or 25 by sharding the factors among multiple GPU devices or by unloading unused factors from the devices. As the accuracy of the posterior approximation is already close to numerical precision at $N = 23$, this would not significantly improve the inference quality and we leave such experiments for future work.

6 Limitations

6.1 Exploding Number of SLPs

By design, Upix inherits the general limitations of the DCC approach. The DCC approach works well when the total number of sub-models / SLPs is low or when the observed data makes it possible to guide inference towards a small set of SLPs with high posterior mass. This was the case for all models considered: for the Pedestrian model we can reject SLPs corresponding to a number of steps greater than 6; for the GMM we can reject SLPs corresponding to a number of components greater than 11; for the Gaussian Process model the number of kernel structures is large, but most of them do not fit the data well; for the Urn model, we can enumerate all 20 relevant SLPs.

We briefly discuss a model for which the DCC approach fails. In statistical phylogenetic analysis [45], birth-death models provide a means to infer properties of the evolutionary tree of a given species. e.g. the birth or death rate of lineages. Data is inherently partial, lacking information about extinct lineages. Thus, it is the job of inference to consider many *unobserved lineages* and evolutionary trees that may explain the partial observed data. In DCC, this leads to an exploding number of SLPs, none of which can be easily rejected based on the partial observations.

6.2 Compilation Time

Upix builds on the JIT-compilation machinery of JAX. Depending on model complexity, inference algorithm complexity, and hardware, this leads to substantial runtime overhead. In Table 2, we report our estimated compilation overhead for the evaluation in Section 4 on consumer hardware. In our scaling experiments of Section 5, we showed that for GPUs the computation time exceeds compile time only for large workloads for most models. Nevertheless, we demonstrated that our method, including compilation, achieves between 7 and 720 times more computation within the same time budget, compared to existing approaches. Furthermore, we showed that Upix enables scaling of inference algorithms on GPUs to workloads that are impractically slow on CPUs and existing approaches. Finally, future work on iterative compilation in JAX may further reduce compilation overhead.

Table 2. Estimated compilation overhead as fraction of runtime for all considered models.

Pedestrian	GP-VI	GMM	GP-SMC	Urn
21%	14%	39%	13%	48%

7 Related Work

7.1 Universal PPLs and Programmable Inference

With the release of Church, Goodman et al. [15] was the first to coin the term *universal* PPL. Since then, many universal PPLs were developed including WebPPL [16], Anglican [50], Pyro [6], and Turing [14]. The universal PPL Venture [30] and its successor Gen [12] are projects which pioneered programmable inference. Upix adapts many established programmable inference ideas which have been developed for MCMC [30], variational inference [4], sequential Monte Carlo [25], and involutive MCMC [11]. It also supports the specification of variational distributions as guide programs, an approach popularised by Pyro [6]. While some components of the inference machinery in the aforementioned systems can be run on GPUs, none support compiling a full programmable inference routine to accelerator hardware for models with stochastic support, such as Upix.

7.2 Inference for Models with Stochastic Support Structure

Many inference algorithms, like light-weight Metropolis Hastings algorithm [53], do not require assumptions about the support of the target model. This makes them general purpose, but also less efficient. When we can put assumptions on the model, like finite dimensionality and differentiability, general-purpose algorithms are outperformed by specialised algorithms like NUTS [20]. There are efforts to extend these specialised algorithms to stochastic support models, e.g. non-parametric HMC [28] or non-parametric involutive MCMC [29].

The Divide-Conquer-Combine (DCC) approach [55] provides a technique to run inference algorithms, specialised for static support structures, on models specified in a universal PPL. This approach is under-explored with only one extension to variational inference called SDVI [43]. As the implementation of the original publication is closed source, there is only one open source implementation of the DCC approach by Reichelt et al. [42] in Pyro [6] (which we compared against in Sections 4 and 5) with a rudimentary translation to NumPyro [41]. Upix is the first system to realise the DCC approach as a framework allowing the instantiations of many new DCC-based inference algorithms.

7.3 Accelerated Inference in Probabilistic Programming Systems

Compiling inference algorithms designed for universal PPLs to accelerator hardware like GPUs or TPUs is challenging. Upix achieves this compilation by building on the DCC approach, but there also exist other techniques. To the best of our knowledge, other than Upix, Lundén et al. [27] presents the only approach capable of compiling a full inference routine for a universal PPL to GPU. However, they solely focus on sequential Monte Carlo.

In general, GPU support is relatively limited in probabilistic programming. We highlight several systems that provide GPU support but, unlike universal PPLs, achieve this by adopting more restricted modelling languages. LibBi [37] is a language for Bayesian state-space modelling that features an OpenMP and CUDA backend for their sequential Monte Carlo implementation. Augur [21] is a probabilistic programming system that enables compilation of MCMC algorithms to CPU and GPU. An OpenCL backend was developed for Stan in 2019 [9]. Systems with a JAX backend like NumPyro [41], BlackJax [7], and TensorflowProbability [24], enable inference of finite dimensional models with static support (i.e., no stochastic control flow or dynamic array shapes) on GPUs and TPUs. Recently, ProbZelus, a reactive probabilistic programming langauge, was also equipped with a JAX backend [2].

It is more common to use GPU acceleration in *deep probabilistic programming* where models incorporate neural networks. For instance, to compute the log-probability density function, systems like Edward [51] or Pyro [6] can dispatch the neural network computations to GPUs.

8 Conclusion

We presented Upix, the first probabilistic programming system that realises the divide-conquer-combine (DCC) approach as a programmable inference framework for universal PPLs. This system enables not only the implementation of existing DCC algorithms, like the original DCC and SDVI, but also the formulation of new DCC algorithms, as was demonstrated by introducing RJMCMC-DCC, SMC-DCC, and VE-DCC. By building on the JAX infrastructure, JIT-compilation to CPUs, GPUs, and TPUs was achieved. On a consumer-grade CPU, this resulted in 7 to 720 times more computation within the same time budget, compared to prior methods, which substantially improves approximation quality for challenging probabilistic models. Lastly, in an empirically evaluation on machines up to 64 CPU cores and 8 GPU devices, we showed how inference algorithms can be scaled in Upix to workloads that are impractically slow for CPUs and existing approaches.

References

[1] Anonymous. 2025. UPIX: Universal Programmable Inference in JAX. https://anonymous.4open.science/r/UPIX-A583/.

[2] Guillaume Baudart, Louis Mandel, and Reyyan Tekin. 2022. Jax based parallel inference for reactive probabilistic programming. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 26–36.

[3] Atilim Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, et al. 2019. Etalumis: Bringing probabilistic programming to scientific simulators at scale. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–24.

[4] McCoy R Becker, Alexander K Lew, Xiaoyan Wang, Matin Ghavami, Mathieu Huot, Martin C Rinard, and Vikash K Mansinghka. 2024. Probabilistic programming with programmable variational inference. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 2123–2147.

[5] Michael Betancourt. 2017. A conceptual introduction to Hamiltonian Monte Carlo. *arXiv preprint arXiv:1701.02434* (2017).

[6] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research* 20, 1 (2019), 973–978.

[7] Alberto Cabezas, Adrien Corenflos, Junpeng Lao, and Rémi Louf. 2024. BlackJAX: Composable Bayesian inference in JAX. arXiv:2402.10797 [cs.MS]

[8] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of statistical software* 76, 1 (2017).

[9] Rok Češnovar, Steve Bronder, Davor Sluga, Jure Demšar, Tadej Ciglarič, Sean Talts, and Erik Štrumbelj. 2019. GPU-based parallel computation support for Stan. *arXiv preprint arXiv:1907.01063* (2019).

[10] Nicolas Chopin, Omiros Papaspiliopoulos, et al. 2020. *An introduction to sequential Monte Carlo*. Vol. 4. Springer.

[11] Marco Cusumano-Towner, Alexander K Lew, and Vikash K Mansinghka. 2020. Automating involutive MCMC using probabilistic and differentiable programming. *arXiv preprint arXiv:2007.09871* (2020).

[12] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*. 221–236.

[13] Roy Frostig, Matthew James Johnson, and Chris Leary. 2019. Compiling machine learning programs via high-level tracing. In *SysML conference 2018*.

[14] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: a language for flexible probabilistic inference. In *International conference on artificial intelligence and statistics*. PMLR, 1682–1690.

[15] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2012. Church: a language for generative models. *arXiv preprint arXiv:1206.3255* (2012).

[16] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2025-3-20.

[17] Peter J Green. 1995. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika* 82, 4 (1995), 711–732.

[18] Nils Lid Hjort, Chris Holmes, Peter Müller, and Stephen G Walker. 2010. *Bayesian nonparametrics*. Vol. 28. Cambridge University Press.

[19] Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. 2013. Stochastic variational inference. *the Journal of machine Learning research* 14, 1 (2013), 1303–1347.

[20] Matthew D Hoffman, Andrew Gelman, et al. 2014. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.* 15, 1 (2014), 1593–1623.

[21] Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017. Compiling Markov chain Monte Carlo algorithms for probabilistic modeling. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 111–125.

[22] Daphne Koller and Nir Friedman. 2009. *Probabilistic graphical models: principles and techniques*. MIT press.

[23] Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. 2017. Automatic differentiation variational inference. *Journal of machine learning research* 18, 14 (2017), 1–45.

[24] Junpeng Lao, Christopher Suter, Ian Langmore, Cyril Chimisov, Ashish Saxena, Pavel Sountsov, Dave Moore, Rif A Saurous, Matthew D Hoffman, and Joshua V Dillon. 2020. tfp. mcmc: Modern Markov chain Monte Carlo tools built for modern hardware. *arXiv preprint arXiv:2002.01184* (2020).

[25] Alexander K Lew, George Matheos, Tan Zhi-Xuan, Matin Ghavamizadeh, Nishad Gothoskar, Stuart Russell, and Vikash K Mansinghka. 2023. Smcp3: Sequential monte carlo with probabilistic program proposals. In *International*

conference on artificial intelligence and statistics. PMLR, 7061–7088.

[26] Fernando Llorente, Luca Martino, David Delgado, and Javier Lopez-Santiago. 2023. Marginal likelihood computation for model selection and hypothesis testing: an extensive review. *SIAM review* 65, 1 (2023), 3–58.

[27] Daniel Lundén, Joey Öhman, Jan Kudlicka, Viktor Senderov, Fredrik Ronquist, and David Broman. 2022. Compiling Universal Probabilistic Programming Languages with Efficient Parallel Sequential Monte Carlo Inference.. In *ESOP*. 29–56.

[28] Carol Mak, Fabian Zaiser, and Luke Ong. 2021. Nonparametric hamiltonian monte carlo. In *International Conference on Machine Learning*. PMLR, 7336–7347.

[29] Carol Mak, Fabian Zaiser, and Luke Ong. 2022. Nonparametric involutive markov chain monte carlo. In *International Conference on Machine Learning*. PMLR, 14802–14859.

[30] Vikash K Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 603–616.

[31] Charles C Margossian and Andrew Gelman. 2023. For how many iterations should we run Markov chain Monte Carlo? *arXiv preprint arXiv:2311.02726* (2023).

[32] Charles C Margossian, Matthew D Hoffman, Pavel Sountsov, Lionel Riou-Durand, Aki Vehtari, and Andrew Gelman. 2024. Nested ^R: Assessing the convergence of Markov chain Monte Carlo when running many short chains. *Bayesian Analysis* 1, 1 (2024), 1–28.

[33] George Matheos, Alexander K Lew, Matin Ghavamizadeh, Stuart Russell, Marco Cusumano-Towner, and Vikash Mansinghka. 2020. Transforming worlds: Automated involutive MCMC for open-universe probabilistic models. In *Third Symposium on Advances in Approximate Bayesian Inference*.

[34] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. 1953. Equation of state calculations by fast computing machines. *The journal of chemical physics* 21, 6 (1953), 1087–1092.

[35] Brian Milch, Bhaskara Marthi, Stuart Russell, David A. Sontag, Daniel L. Ong, and Andrey Kolobov. 2005. BLOG: Probabilistic Models with Unknown Objects. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, Leslie Pack Kaelbling and Alessandro Saffiotti (Eds.). Professional Book Center, 1352–1359.

[36] Brian Milch, Bhaskara Marthi, David Sontag, Stuart Russell, Daniel L Ong, and Andrey Kolobov. 2005. Approximate inference for infinite contingent Bayesian networks. In *International Workshop on Artificial Intelligence and Statistics*. PMLR, 238–245.

[37] Lawrence M Murray. 2015. Bayesian state-space modelling on high-performance hardware using LibBi. *Journal of Statistical Software* 67 (2015), 1–36.

[38] Kirill Neklyudov, Max Welling, Evgenii Egorov, and Dmitry Vetrov. 2020. Involutive MCMC: a unifying framework. In *International Conference on Machine Learning*. PMLR, 7273–7282.

[39] Akihiko Nishimura, David B Dunson, and Jianfeng Lu. 2020. Discontinuous Hamiltonian Monte Carlo for discrete parameters and discontinuous likelihoods. *Biometrika* 107, 2 (2020), 365–380.

[40] Agostino Nobile and Alastair T Fearnside. 2007. Bayesian finite mixtures with an unknown number of components: The allocation sampler. *Statistics and Computing* 17, 2 (2007), 147–162.

[41] Du Phan, Neeraj Pradhan, and Martin Jankowiak. 2019. Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro. *arXiv preprint arXiv:1912.11554* (2019).

[42] Tim Reichelt, Luke Ong, and Thomas Rainforth. 2022. Rethinking variational inference for probabilistic programs with stochastic support. *Advances in Neural Information Processing Systems* 35 (2022), 15160–15175.

[43] Tim Reichelt, Luke Ong, and Tom Rainforth. 2024. Beyond Bayesian Model Averaging over Paths in Probabilistic Programs with Stochastic Support. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 829–837.

[44] Sylvia Richardson and Peter J Green. 1997. On Bayesian analysis of mixtures with an unknown number of components (with discussion). *Journal of the Royal Statistical Society Series B: Statistical Methodology* 59, 4 (1997), 731–792.

[45] Fredrik Ronquist, Jan Kudlicka, Viktor Senderov, Johannes Borgström, Nicolas Lartillot, Daniel Lundén, Lawrence Murray, Thomas B Schön, and David Broman. 2021. Universal probabilistic programming offers a powerful approach to statistical phylogenetics. *Communications biology* 4, 1 (2021), 244.

[46] Feras Saad, Brian Patton, Matthew Douglas Hoffman, Rif A Saurous, and Vikash Mansinghka. 2023. Sequential Monte Carlo learning for time series structure discovery. In *International Conference on Machine Learning*. PMLR, 29473–29489.

[47] Feras A Saad, Marco F Cusumano-Towner, Ulrich Schaechtle, Martin C Rinard, and Vikash K Mansinghka. 2019. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32.

[48] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* 2 (2016), e55.

[49] Pavel Sountsov, Colin Carroll, and Matthew D Hoffman. 2024. Running markov chain monte carlo on modern hardware and software. *arXiv preprint arXiv:2411.04260* (2024).

[50] David Tolpin, Jan-Willem van de Meent, and Frank Wood. 2015. Probabilistic programming in Anglican. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III 15*. Springer, 308–311. https://probprog.github.io/anglican/

[51] Dustin Tran, Matthew D Hoffman, Rif A Saurous, Eugene Brevdo, Kevin Murphy, and David M Blei. 2017. Deep probabilistic programming. *arXiv preprint arXiv:1701.03757* (2017).

[52] Christopher KI Williams and Carl Edward Rasmussen. 2006. *Gaussian processes for machine learning*. Vol. 2. MIT press Cambridge, MA.

[53] David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, 770–778.

[54] Yi Wu, Lei Li, Stuart Russell, and Rastislav Bodik. 2016. Swift: Compiled inference for probabilistic programming languages. *arXiv preprint arXiv:1606.09242* (2016).

[55] Yuan Zhou, Hongseok Yang, Yee Whye Teh, and Tom Rainforth. 2020. Divide, conquer, and combine: a new inference strategy for probabilistic programs with stochastic support. In *International Conference on Machine Learning*. PMLR, 11534–11545.