Efficiently Vectorized MCMC on Modern Accelerators

Hugh Dance[1] Pierre Glaser[1] Peter Orbanz[1] Ryan Adams[2]

March 3, 2025

Abstract

With the advent of automatic vectorization tools (e.g., JAX's `vmap`), writing multi-chain MCMC algorithms is often now as simple as invoking those tools on single-chain code. Whilst convenient, for various MCMC algorithms this results in a synchronization problem—loosely speaking, at each iteration all chains running in parallel must wait until the last chain has finished drawing its sample. In this work, we show how to design single-chain MCMC algorithms in a way that avoids synchronization overheads when vectorizing with tools like `vmap`, by using the framework of finite state machines (FSMs). Using a simplified model, we derive an exact theoretical form of the obtainable speed-ups using our approach, and use it to make principled recommendations for optimal algorithm design. We implement several popular MCMC algorithms as FSMs, including Elliptical Slice Sampling, HMC-NUTS, and Delayed Rejection, demonstrating speed-ups of up to an order of magnitude in experiments.

1 Introduction

Automatic vectorization is the act of transforming one function into another that can handle batches of inputs without user intervention. Implementations of automatic vectorization algorithms—such as JAX's `vmap`—are now available in many mainstream scientific computing libraries, and have dramatically simplified the task of running multiple instances of a single algorithm concurrently. They are routinely used to train neural networks (Flax, 2023) and in other scientific applications, e.g., Schoenholz and Cubuk (2021); Oktay et al. (2023); Pfau et al. (2020).

This paper focuses on the use of automatic vectorization for Markov chain Monte Carlo (MCMC) algorithms. Tools like `vmap` provide a convenient way to run multiple MCMC chains in parallel: one can simply write single-chain MCMC code, and call `vmap` to turn

[1]Gatsby Unit, University College London
[2]Princeton University

it into vectorized, multi-chain code that can run in parallel on the same processor. Many state-of-the-art MCMC libraries have consequently adopted machine learning frameworks with automatic vectorizaton tools—such as JAX or TensorFlow—as their backend.

One limitation with automatic vectorization tools in modern frameworks is how they handle control flow. Since all instructions must be executed in lock-step, if the algorithm has a while loop all chains must wait until the last chain has finished its iterations. This can lead to serious inefficiencies for MCMC algorithms that generate each sample using variable-length while loops. Roughly speaking, if vectorization executes, say, 100 chains in parallel, all but one finish after at most 10 steps, and the remaining chain runs for 1000 steps, then about 99% of the GPU capacity assigned to `vmap` is wasted (and our simulations show that the effect can indeed be this drastic). For the No-U-Turn Sampler (HMC-NUTS) (Hoffman et al., 2014), this problem is well-documented (BlackJax, 2019; Sountsov et al., 2024; Radul et al., 2020). However this also affects various other algorithms, such as variants of slice sampling (Neal, 2003; Murray et al., 2010; Cabezas and Nemeth, 2023), delayed rejection methods (Mira et al., 2001; Modi et al., 2024) and unbiased Gibbs sampling (Qiu et al., 2019).

In this work, we show how to transform MCMC algorithms into an equivalent sampling algorithm that avoids these synchronization barriers when using `vmap`-style vectorization. In particular,

1. We develop a novel approach to transform MCMC algorithms into finite state machines (FSMs), that can avoid synchronisation barriers when vectorizing with tools like `vmap`.

2. We analyze the time complexity of our FSMs against standard MCMC implementations and derive a theoretical bound on the speed-up under a simplified model.

3. We use our analysis to develop principled recommendations for optimal FSM design, which enable us to nearly obtain the theoretical bound in speed-ups for certain MCMC algorithms.

4. We implement several popular MCMC algorithms as FSMs, including Elliptical Slice Sampling, HMC-NUTS, and Delayed-Rejection MH - demonstrating speed-ups of up to an order of magnitude in experiments.

2 Background and Problem Setup

In this section, we briefly review how MCMC algorithms are vectorized, explain the synchronization problems that can arise, and formalize the problem mathematically.

2.1 MCMC Algorithms and Vectorization

MCMC methods aim to draw samples from a target distribution π (typically on a subset of $\mathbb{R}^d$) which is challenging to sample from directly. To do so, they generate samples $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\} \in \mathbb{R}^{d \times n}$ from a Markov chain with transition kernel P and invariant distribution π, by starting from an initial state $\boldsymbol{x}_0 \in \mathbb{R}^d$ and iteratively sampling $\boldsymbol{x}_{i+1} \sim P(\cdot|\boldsymbol{x}_i)$. For aperiodic and irreducible Markov chains, as $n \to \infty$ the samples will converge in distribution to π (Brooks et al., 2011). In practice, P is implemented by a deterministic function `sample`, which takes in the current state $\boldsymbol{x}_i$ and a pseudo-random state $r_i \in \mathbb{N}$, and returns new states $\boldsymbol{x}_{i+1}$ and r_{i+1}. This procedure is given in Algorithm 1.

Algorithm 1 MCMC algorithm with `sample` function

1: **Inputs**: sample $\boldsymbol{x}_0$, seed r_0
2: **for** $i \in \{1, ..., n\}$ **do**
3: generate $\boldsymbol{x}_i, r_i \leftarrow$ `sample`$(\boldsymbol{x}_{i-1}, r_{i-1})$
4: **end for**
5: **return** $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$

Practitioners commonly run Algorithm 1 on m different initializations, producing m chains of samples. Given an implementation of `sample`: $\mathbb{R}^d \times \mathbb{N} \to \mathbb{R}^d \times \mathbb{N}$, one way to do this is to use an automatic vectorization tool like JAX's `vmap`[1]. `vmap` takes `sample` as an input and returns a new program, `vmap(sample)`: $\mathbb{R}^{d \times m} \times \mathbb{N}^m \to \mathbb{R}^{d \times m} \times \mathbb{N}^m$, which operates on a batch of inputs collected into tensors

$$\begin{aligned} \tilde{\boldsymbol{x}}_{i-1} &:= (\boldsymbol{x}_{i-1,1}, \ldots, \boldsymbol{x}_{i-1,m}) \in \mathbb{R}^{d \times m} \\ \tilde{r}_{i-1} &:= (r_{i-1,1}, \ldots, r_{i-1,m}) \in \mathbb{N}^m \end{aligned}$$

and returns the corresponding outputs from `sample`. One can therefore turn Algorithm 1 into a multi-chain algorithm by simply replacing `sample` with `vmap(sample)` and replacing $(\boldsymbol{x}_0, r_0)$ by $(\tilde{\boldsymbol{x}}_0, \tilde{r}_0)$ (see Algorithm 6 in Appendix B). Under the hood, `vmap` transforms every instruction in `sample` (e.g., a dot product) into a corresponding instruction operating on a batch of inputs (e.g., matrix-vector multiplication); that is, it 'vectorizes' `sample`. These instructions are executed in lock-step across all chains. Using `vmap` usually yields code that performs as well as manually-batched code. For this reason, as well as its simplicity and composability with other transformations like `grad` (for automatic differentiation) and `jit` (for Just-In-Time compilation), `vmap` has been adopted by major MCMC libraries such as NumPyro and BlackJAX.

[1]We use JAX and its vectorization map `vmap` throughout, since this framework is widely adopted. Similar constructs exist in TensorFlow (`vectorized_map`) and PyTorch (`vmap`).

2.2 Synchronization Problems with While Loops

Control flow (i.e., `if/else`, `while`, `for`, etc.) poses a challenge for vectorization, because different batch members may require a different sequence of instructions. `vmap` solves this by executing all instructions for all batch members, and masking out the irrelevant computations. A consequence of this is that if `sample` contains a `while` loop, then `vmap(sample)` will execute the body of this loop for all chains until all termination conditions are met. Until then, no further instruction can be executed. As a result, if there is high variation in the number of loop iterations across chains, running Algorithm 1 with `vmap(sample)` introduces a synchronization barrier across all chains: *at every iteration, each chain has to wait for the slowest* `sample` *call.*

This issue arises in practice, because a number of important MCMC algorithms have while loops in their `sample` implementations: such as variants of slice sampling (Neal, 2003; Murray et al., 2010; Cabezas and Nemeth, 2023), delayed rejection methods (Mira et al., 2001; Modi et al., 2024), the No-U-Turn sampler (Hoffman et al., 2014), and unbiased Gibbs sampling (Qiu et al., 2019).

Formalizing The Problem. Here we formalize the problem through a series of short derivations. These will be made precise in Section 4. Suppose we run a `vmap`'ed version of Algorithm 1 using a `sample` function that has a while loop. Let $N_{i,j}$ denote the number of iterations required by the jth chain to obtain its ith sample. If the while loop has a variable length, $N_{i,j}$ is a random number. Due to the synchronization problem described above, the time taken to run `vmap(sample)` at iteration i is approximately proportional to the largest $N_{i,j}$ out of the m chains, $\max_{j \le m} N_{i,j}$. The total runtime after n samples, $C_0(n)$, is then approximately proportional to,

$$C_0(n) \;\overset{\propto}{\approx}\; \sum_{i=1}^{n} \max_{j \le m} N_{i,j} \tag{1}$$

By contrast, if the chains could be run without any synchronization barriers, the time taken would instead be

$$C_*(n) \;\overset{\propto}{\approx}\; \max_{j \le m} \sum_{i=1}^{n} N_{i,j} \tag{2}$$

The key difference is that the maximum is now outside the sum. This reflects the fact that when running independently, we only have to wait at the end for the slowest chain to collect its n samples, rather than waiting at every iteration. Clearly $C_*(n) \le C_0(n)$. Significant speed-ups are obtainable by de-synchronizing the chains when $C_*(n) \ll C_0(n)$. If each $N_{i,j}$ converges in distribution to some $\mathbb{P}_N$ as we draw more samples and an appropriate law of large numbers holds, we can expect for large enough n that:

$$C_0(n) \;\overset{\propto}{\approx}\; n\,\mathbb{E} \max_{j \le m} N_{\infty,j} \tag{3}$$

$$C_*(n) \;\overset{\propto}{\approx}\; \max_{j \le m} n\,\mathbb{E}N_{\infty,j} \;=\; n\,\mathbb{E}N_{\infty,1} \tag{4}$$

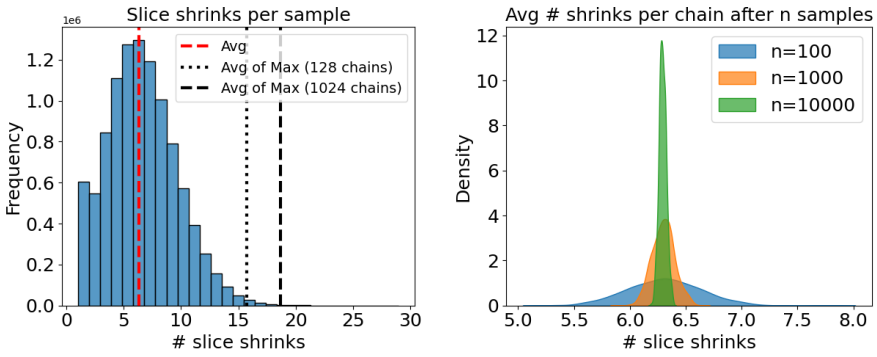

Figure 1: Statistics on the elliptical slice sampler for Gaussian Process Regression on the Real Estate Dataset (Yeh, 2018). LHS: histogram of the number of slice shrinks per sample. RHS: smoothed histogram of the average number of slice shrinks per sample across $m = 1024$ chains, after $n \in \{100, 1000, 10000\}$ samples.

where $(N_{\infty,1}, ..., N_{\infty,m}) \overset{iid}{\sim} \mathbb{P}_N$. Therefore, de-synchronizing the chains should lead to large speedups if

$$\mathbb{E} \max_{j \leq m} N_{\infty,j} \gg \mathbb{E} N_{\infty,1} \tag{5}$$

In this work, we will see that Equation (5) holds in various situations.

Example. Consider the elliptical slice sampling algorithm (Murray et al., 2010) which samples from distributions which admit a density with Gaussian components, $p(\boldsymbol{x}) \propto f(\boldsymbol{x})\mathcal{N}(\boldsymbol{x}|0, \Sigma)$. Its transition kernel (see Algorithm 8 in Appendix B) draws each sample by (i) generating a random ellipse of permitted moves and an initial proposal, and (ii) iteratively shrinking the set of permitted moves and resampling the proposal from this set until it exceeds a log-likelihood threshold. The second stage uses a while loop which requires a random number of iterations.
On Figure 1 we display results when implementing this algorithm in JAX to sample from the hyperparameter posterior of a Gaussian process implemented on a regression task using a real dataset from the UCI repository (details are in Section 7). On the LHS we can see the distribution of the number of while loop iterations (i.e. slice shrinks) needed to generate a sample. While the average is $\sim$6, the average of the *maximum* across 1024 chains is >18, which implies $\mathbb{E} \max_{j \leq 1024} N_{n,j} \approx 3\mathbb{E} N_{n,1}$. On the RHS, we can see that the differences across the chains do balance out as more samples are drawn, which suggests a certain law of large numbers does hold here. In particular, after just 100 samples the distribution of the *average* number of iterations per chain is contained in the interval $[5, 8]$, and after 10,000 samples this shrinks to $[6.2, 6.4]$. This implies that if we could vectorize the algorithm without incurring synchronization barriers, we could improve the Effective Sample Size per Second (ESS/Sec) by up to 3-fold. We will see that for other algorithms (e.g., HMC-NUTS and delayed rejection) the potential speed-ups are much larger than this.

3 Finite State Machines for MCMC

In this section we present an implementation approach for MCMC algorithms with a given `sample` function, which avoids the above synchronization problems when vectorizing with `vmap`. The basic idea is to break down `sample` into a series of smaller 'steps' which avoid using iterative control flow like while loops and have minimal variance in execution time. We will then define a runtime procedure that allows chains to progress through their own step sequences in de-synchronized fashion. To do this in a principled manner, we use the framework of finite state machines (FSMs). Formally, an FSM is a **5-tuple** $(\mathcal{S}, \mathcal{Z}, \delta, \mathcal{B}, \mathcal{F})$, where $\mathcal{S}$ is a finite set of states, $\mathcal{Z}$ is a (finite) input set, $\delta : S \times \mathcal{Z} \to S$ is a transition function, $\mathcal{B} \in S$ is an initial state, and $\mathcal{F}$ is the set of 'final' states (Hopcroft et al., 2001). FSMs are useful tools which are typically used to represent algorithms that run on machines with finite storage. Below we show how to represent different `sample` functions as an FSM.

3.1 `sample`-to-FSM conversion

At a high-level, we construct the FSM of an MCMC algorithm as follows. The states $\mathcal{S} = (S_1, ..., S_K)$ are chosen as functions which execute contiguous code blocks of the algorithm. The boundaries of each code block are delineated by the start and end of any while loops. For example, S_1 executes all lines of code before the first while loop, S_2 executes all lines of code from the beginning of the first while loop body to either the start of the next while loop or the end of the current while loop, and so on. The inputs $z \in \mathcal{Z}$ are all variables taken as input to each code block. (e.g. current sample $\boldsymbol{x}$, proposal $\boldsymbol{x}'$, log-likelihood $\log p(\boldsymbol{x})$)[2], whilst the transition function δ selects the next code block to run according to the relevant loop condition and the received output z' after executing the current block (e.g. if a while loop starts after the block S_1 executes, δ will check this loop's condition using the output $z' = S_1(z)$). Below we make this construction procedure more precise for three kinds of `sample` functions which cover all MCMC algorithms considered in the present work: (i) functions with a single while loop, (ii) functions with two sequential while loops, and (iii) functions with two nested while loops. Additional details on the construction process are in Appendix B.

The single while loop case. The simplest case is a `sample` method with a single while loop. This covers (for example) elliptical slice sampling (Murray et al., 2010) and symmetric delayed rejection Metropolis-Hastings (Mira et al., 2001). In this case, we break `sample` down into three code blocks B_1, B_2, B_3 (one before the while loop, one for the body of the while loop, and one after the while loop) and the termination condition of the loop. This is shown on the LHS of Figure 2. Using these blocks, we define the FSM as $(\mathcal{S}, \mathcal{Z}, \delta, \mathcal{B}, \mathcal{F})$, where (1) $\mathcal{Z}$ is the set of possible values for the local variables of `sample` (e.g., the current sample $\boldsymbol{x}$, seed r, and proposal etc.), (2) $\mathcal{S} = \{S_1, S_2, S_3\}$

[2]Whilst we may have that (e.g.) the sample $\boldsymbol{x} \in \mathbb{R}^d$, which violates finiteness of $\mathcal{Z}$, in practice an MCMC algorithm works using a finite subset of $\mathbb{R}^d$ (e.g. 32-bit floating point precision).

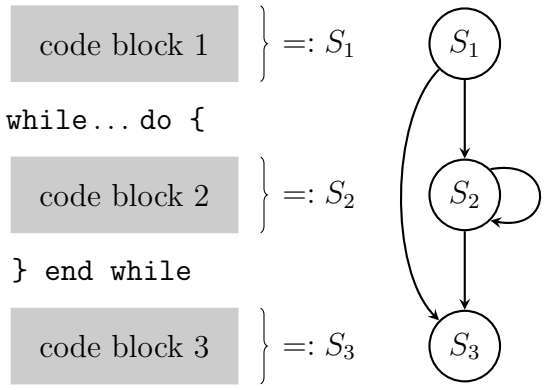

Figure 2: FSM of an MCMC algorithm with a single while loop.

is a set of three functions $\mathcal{Z} \to \mathcal{Z}$, where for each $k \in \{1, 2, 3\}$, $S_k(z)$ runs B_k on local variables z and returns their updated value, (3) for each $k \in \{0, 1\}$ and $z \in \mathcal{Z}$, the transition function $\delta(S_k, z)$ checks the while loop termination condition using z, and returns S_2 if False and S_3 if True, (4) $\mathcal{B} = S_1$ and, finally, (5) $\mathcal{F} = S_3$. The RHS of Figure 2 illustrates the resulting FSM diagram. Note there is an edge $S_k \to S_{\tilde{k}}$ between states if and only if $\delta(S_k, z) = S_{\tilde{k}}$ for some z.

Two sequential while loops. We break down `sample` into two blocks: B_1 contains all the code up to the second while-loop. B_2 contains the remaining code. In this case, B_1 and B_2 are now single while-loop programs, and thus can both be represented by FSMs F_1 and F_2 using the above rule. The FSM representation of `sample` can then be obtained by "stitching together" F_1 and F_2. The full construction process is in Appendix B. The resulting FSM is provided for the case of the Slice Sampler (Figure 3, top-right panel), which contains two[3] while loops—one for expanding the slice, and one for contracting the slice.

Two nested while-loops In the case of two nested while loops, we break down `sample` into B_1, B_2, B_3, where B_1 (resp. B_3) is the code before (resp. after) the outer while loop and B_2 is the outer while-loop body. As B_2 is a single-while loop program, it admits its own FSM F_i. Building the final FSM of `sample` then informally consists in obtaining a first, "coarse" FSM by treating B_2 as opaque, and then refining it by replacing B_2 with its own FSM. The full construction process is given in Appendix B. The resulting FSM is provided for the case of NUTS (Figure 3, bottom-right panel), which—in its iterative form—uses the outer while loop to determine whether to keep expanding a Hamiltonian trajectory, and the inner while loop to determine whether to keep integrating along the current trajectory.

[3]We note that for 1D problems, the slice expansion loop can be broken into two loops—one for the upper bound of the interval, and one for the lower bound.

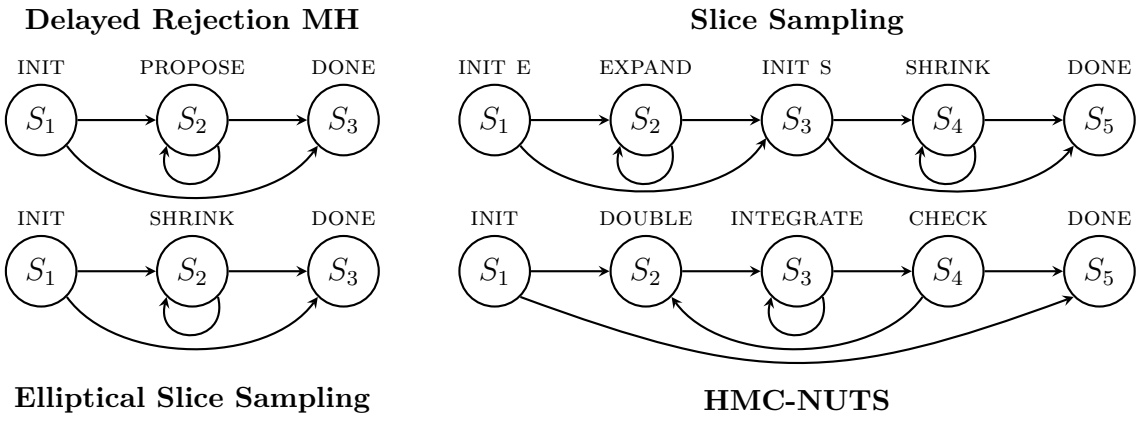

Figure 3: Finite state machines for the `sample` function of different MCMC algorithms: The symmetric delayed-rejection Metropolis-Hastings algorithm (Mira et al., 2001), elliptical slice sampling (Murray et al., 2010), (vanilla) slice sampling (with single slice expansion loop) (Neal, 2003), the No-U-Turn sampler for Hamiltonian Monte Carlo (Hoffman et al., 2014).

3.2 Defining the FSM runtime

Going forward, for convenience we assume the transition function δ takes in and returns the *label* k of each block S_k, rather than S_k itself. Now, given a constructed FSM, in Algorithm 2 we define a function `step(k,z)` which when executed performs a single transition along an edge in the FSM graph. For reasons that will be clear shortly, we augment this function with a flag that indicates when the final block is run.

Algorithm 2 `step` function for FSM

Inputs: algorithm state k, variables z

1: set `isSample` $\leftarrow 1\{k = K\}$
2: $z \leftarrow$ `switch`$(k, [\{\text{run } S_1(z)\}, ..., \{\text{run } S_K(z)\}])$
3: $k \leftarrow$ `switch`$(k, [\{\text{run } \delta(1, z)\}, ..., \{\text{run } \delta(K, z)\}])$
4: **return** $(k, z,$ `isSample`$)$

Note that if we start from some input $(k, z) = (0, z_0)$, by calling `step` repeatedly we will transition through a sequence of blocks of `sample`, until we eventually reach the terminal state, at which point a sample is obtained (as indicated by `isSample=True`). We can use this function to draw n samples, by (1) adding a transition from the terminal state $\mathcal{F} = S_K$ back to the initial state $\mathcal{B} = S_0$, and (2) defining a wrapper function which iteratively calls `step` until `isSample=True` is obtained n-times (see Algorithm 3). Both Algorithm 3 and Algorithm 1 draw n samples from `sample` and can be easily vectorized with `vmap`. In the case of Algorithm 3, we just call `vmap` on `step` and modify the outer loop to terminate when all chains have collected n samples each (see Algorithm 7 in Appendix B). The crucial difference is that: (i) by changing the definition

Algorithm 3 FSM MCMC algorithm

input: initial value $\boldsymbol{x}_0$, # samples n
initialize: $z = \text{init}(\boldsymbol{x}_0), X = \text{list}(), B = \text{list}()$
Set $t = 0$ and $k = 0$
while $t < n$ **do**
    $(k, z, \texttt{isSample}) \leftarrow \texttt{step}(k, z)$
    append current sample value $\boldsymbol{x}$ stored in z to X
    append `isSample` to B
    update sample counter $t \leftarrow t + \texttt{isSample}$
end while
return $X[B]$

of a 'step' to allow different chains to execute different code blocks, Algorithm 3 enables them to progress their own independent block sequences, and (ii) by essentially moving the while loop to the outer layer, Algorithm 3 only requires the chains to re-synchronize after n samples.

4 Time Complexity Analysis of FSM-MCMC

One limitation with our FSM design is the `step` function relies on a switch to determine which block to run. When vectorized with `vmap` or an equivalent transformation, all branches are evaluated for all chains, with irrelevant results discarded. This means the cost of a single call of `step` is the cost of running all state functions $S_1, ..., S_K$. To obtain a speedup, the FSM must therefore sufficiently decrease the expected number of steps to obtain n samples from each chain. In this section we quantitatively derive conditions under which this occurs in the simplified setting of an MCMC algorithm with a single while loop. This enables us to subequently optimize the design of the FSM in Section 5.

To this end, consider a `sample` function with a single while loop, and associated FSM with K states, where S_k (for some $k \in [K]$) executes the body of this loop. Note our approach yields $K \leq 3$ when there is one while loop, but we relax this here to analyze the effect of the number of blocks on performance. We assume the cost of executing $S_1, .., S_K$ is $c_1(m), ...c_K(m)$ respectively when using vectorized Algorithm 1 and $\alpha \sum_{k=1}^{K} c_k(m)$ when using vectorized Algorithm 3, where $\alpha \in [\max_{k\in[K]} c_k(m) / \sum_{k=1}^{K} c_k(m), 1]$. The variable α reflects the cost of using multiple branches in `step`, which is 1 using Algorithm 2, but can be decreased by refining `step`, as we show later. The dependence of each cost $c_i(m)$ on m reflects the ability of the GPU to efficiently parallelize a call to code-block S_i for m chains. Under these assumptions, the average cost per sample for m chains after n samples each, is expressed for the standard design $C_0(n, m)$ and FSM design

$C_F(n,m)$ as

$$C_0(m,n) \;=\; A_0(m) + B_0(m)\frac{1}{n}\sum_{i=1}^{n}\max_{j\in[m]}(N_{i,j}) \tag{6}$$

$$C_F(m,n) \;=\; A_F(m) + B_F(m)\max_{j\in[m]}\left(\frac{1}{n}\sum_{i=1}^{n}N_{i,j}\right) \tag{7}$$

Here $A_0(m) = c_{\neg k}(m)$ and $A_F(m) = \alpha(K-1)(c_{\neg k}(m) + c_k(m))$ are the costs of calling all blocks except S_k (and $c_{\neg k}(m) = \sum_{j\neq k} c_j(m)$); $B_0(m) = c_k(m)$ and $B_F(m) = \alpha(c_k(m) + c_{\neg k}(m))$ are the costs of running the iterative block S_k; and $N_{i,j}$ is now the number of calls of S_k needed to produce sample i for chain j (i.e. $\boldsymbol{X}_{i,j}$)[4]. If the joint sequence $(\boldsymbol{X}_{i,j}, N_{i,j})_{i\geq 1}$ is a convergent Markov chain, we can state the following concentration result, which formally justifies our derivations leading to (3) and (4).

Theorem 4.1. *Let $N_{i,j} \in [0,B]$, $\boldsymbol{X}_{i,j} \in \mathcal{X} \subset \mathbb{R}^d$, $(\boldsymbol{X}_{i,j}, N_{i,j})_{i\geq 1}$ be a Markov Chain with stationary distribution π. Then with probability $1-\delta$ we have the inequalities*

$$|C_0(m,n) - A_0(m) - B_0(m)\mathbb{E}_\pi \max_{j\in[m]} N_j| \;\leq\; MB_0(m)n^{-\frac{1}{2}}\ln(2/\delta) \tag{8}$$

$$|C_F(m,n) - A_F(m) - B_F(m)\mathbb{E}_\pi N_1| \;\leq\; MB_F(m)n^{-\frac{1}{2}}\ln(2m/\delta) \tag{9}$$

Where $(N_1,..,N_m) \overset{iid}{\sim} \pi_N$ and $M > 0$.

The result essentially says that as $n \to \infty$, the cost per sample of the standard MCMC design will converge to the expected time for the *slowest* chain to draw a sample (i.e. $A_0(m) - B_0(m)\mathbb{E}_\pi \max_{j\in[m]} N_j$), whilst the FSM design will converge to the expected time for a *single* chain to draw a sample, scaled by an additional cost of control-flow in `step` (i.e. $A_F(m) - B_F(m)\mathbb{E}_\pi N_1|$). Both convergence rates are $\mathcal{O}(n^{-\frac{1}{2}})$.

Remark 4.2. The Markov chain assumption is satisfied whenever $N_{i,j}$ depends (only) on the geometry of the distribution at $\boldsymbol{X}_{i-1,j}$. Its convergence can be induced by irreducibility and aperiodicity, which we expect will typically hold under irreducibility and aperiodicity of the marginal Markov chain $(\boldsymbol{X}_{i,j})_{i>1}$.

4.1 The long-run relative cost of the FSM

Substituting the form of the constants into Theorem 4.1, the ratio of the long-run expected runtimes, $E(m)$, is given by the ratio

$$E(m) \;=\; \frac{c_{\neg k}(m) + c_k(m)\mathbb{E}\max_{j\in[m]} N_j}{\alpha(c_{\neg k}(m) + c_k(m))(K - 1 + \mathbb{E}N_1)} \tag{10}$$

[4]We use capital letters here as our analysis in this section treats the samples as random variables.

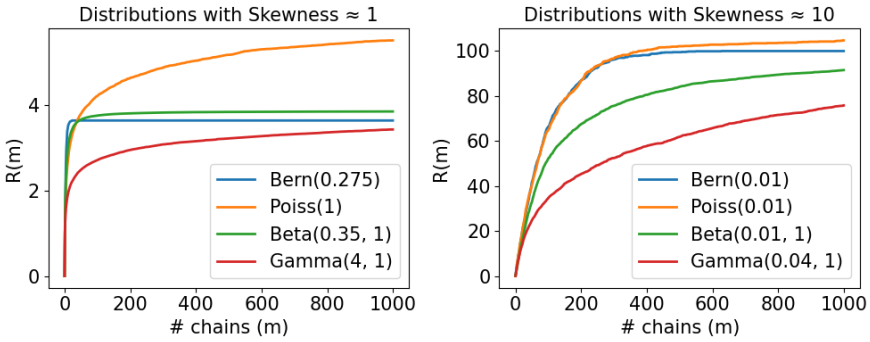

Figure 4: $R(m) = \frac{\mathbb{E}\max_{j\in[m]} N_j}{\mathbb{E}M_1}$ for different distributions with skewness $\gamma_1 \approx 1$ (LHS) and $\gamma_1 \approx 10$ (RHS).

In Proposition A.2 in Appendix A we prove that:

$$E(m) \;\leq\; R(m) \;:=\; \frac{\mathbb{E}\max_{j\in[m]} N_j}{\mathbb{E}N_1} \tag{11}$$

and that this bound is tight. We refer to $R(m)$ as the 'theoretical efficiency bound' for the FSM. Note from Equation (10) that minimizing α and K improves the efficiency $E(m)$ of the FSM. In Section 5 we will introduce two techniques to minimize α and K in practice, which enable us to nearly obtain the efficiency bound $R(m)$ for certain MCMC algorithms.

The scale of potential speed-ups. The size of $R(m)$ depends (only) on the underlying distribution of N_1 (since $N_i =_d N_j \; \forall i, j \in [m]$). Whenever N_1 is sub-exponential, it is known that $R(m) = \mathcal{O}(\ln(m))$ (Vershynin, 2018). Although this implies a slow rate of increase in m, $R(m)$ can be still be very large for small values of m. For example, if $N_j/B \sim Bern(p)$ (i.e., one either needs zero or B iterations to get a sample), then $R(m) = (1-(1-p)^m)/p$ and converges to $1/p$ exponentially fast as m increases. For small p this can be very large even for small m. In general $R(m)$ is sensitive to the skewness of the distribution: distributions on $[0, B]$ with zero skewness have $R(m) = 2$, whilst distributions with skew of 10 can have $R(m) \approx 100$; see Figure 4. Intuitively, these are the distributions where chains are slow only occasionally, but at least one chain is slow often. In such cases, our FSM-design can lead to enormous efficiency gains, as we show in experiments.

5 Optimal FSM design for MCMC algorithms

In this section we provide two strategies to modify the function `step` to (effectively) reduce α and K. These strategies enable us to develop MCMC implementations which nearly obtain the theoretical bound $R(m)$ in some experiments.

5.1 Step bundling to reduce K

Given an FSM with step function `step` defined by code blocks $S_1, ..., S_K$ and transition function δ, one can 'bundle' multiple FSM steps together using a modified step function, `bundled_step`, which replaces the switch over the algorithm state k in `step` with a series of separate conditional statements. This is shown in Algorithm 4 for an example with two state functions. Note that under the "run all branches and mask" behaviour of `vmap`, `vmap(step)` and `vmap(bundled_step)` have the same cost. However, whenever `bundled_step` runs S_1 and δ returns $k = 2$, it immediately also runs S_2. This essentially reduces the 'effective' number of states K and/or reduces the overall number of steps needed to recover a sample, increasing efficiency. In principle, the block ordering can be optimized for sequences that are expected to occur with higher probability.

Algorithm 4 `bundled_step` for FSM with S_1, S_2

Inputs: algorithm state k, variables z

set `isSample` $\leftarrow 1\{k = 2\}$
if $k = 1$ **then**
  run block S_1 with local variables z
  update state $k \leftarrow \delta(1, z)$
end if
if $k = 2$ **then**
  run block S_2 with local variables z
  update state $k \leftarrow \delta(2, z)$
end if
return $(k, z,$ `isSample`$)$

5.2 Cost amortization

If a function g is called on a variable $\theta \in z$ inside multiple state functions, a single call of `vmap(step)` (or `vmap(bundled_step)`) will compute $g(\theta)$ multiple times. To prevent this, we propose: (1) augmenting `step` to return another flag `doComputation` that indicates when this computation is needed in the next code block, and (2) defining a new step function `amortized_step` around `step` which calls `step` once, and executes g if `doComputation=True`. The resulting step function is shown in Algorithm 5. Note that now $g(\theta)$ is only called once per step when using `vmap` on `amortized_step`, even if it required for every code block S_k. This can extend to multiple functions $g_1, .., g_l$. In practice, we find amortization most powerful for when the log-density is expensive and needed in multiple state functions, in which case we can set $g(\cdot) = \log p(\cdot)$. The elliptical slice sampler is one such case, where the log-pdf is needed in both S_1 (i.e. the block before the while loop - INIT in Figure 3) and S_2 (i.e. the block for the body of the while loop - SHRINK in Figure 3), to check the while loop terminator.

Algorithm 5 `amortized_step` for FSM with function g

Input: Algorithm state k, variables z
$(k, z, \texttt{isSample}, \texttt{doComputation}) \leftarrow \texttt{step}(k, z)$
if `doComputation` **then**
    Unpack state $(z', \theta) = z$
    Do computation $\theta \leftarrow g(z)$
    Re-pack state $z \leftarrow (z', \theta)$
end if
Return $(k, z, \texttt{isSample})$

6 Related work

FSMs in Machine Learning. Previous work in machine learning has used the framework of FSMs to design image-based neural networks (Ardakani et al., 2020) and Bayesian non-parametric time series models (Ruiz et al., 2018), as well as extract representations from Recurrent-Neural-Networks (RNNs) (Muškardin et al., 2022; Cechin et al., 2003; Tiňo et al., 1998; Zeng et al., 1993; Koul et al., 2018; Svete and Cotterell, 2023). To our knowledge, our work is the first that use used FSMs as a framework to represent MCMC algorithms, and design novel implementations.

Efficient MCMC on Modern Hardware. Given the recursive nature of HMC-NUTS, previous work has reformulated the algorithm for compatibility with machine learning frameworks that cannot naively support recursion (Abadi et al., 2016; Phan et al., 2019; Lao et al., 2020). However, these implementations do not address synchronization inefficiencies caused by automatic vectorization tools. Our work bears some similarities with a general-purpose algorithm proposed in the High-Performance-Computing literature for executing batched recursive programs (Radul et al., 2020). However whilst both their method and ours breaks programs down into smaller blocks for efficient vectorization, there are several major differences. Our approach provides a recipe for implementing a range of single-chain iterative MCMC algorithms and is fully compatible with automatic vectorization tools like `vmap`. In contrast, their algorithm is designed to work with recursive programs and requires code which is already batched. Crucially, in order to avoid synchronization barriers due to while loops, this code cannot have been batched with a `vmap`-equivalent vectorization tool, because `vmap` converts while loops into a single batched primitive which cannot be broken down by their algorithm. Additionally, our method breaks programs down into the coarsest (while-loop-free) blocks possible and allows for control flow within each block, which we showed is crucial for optimal performance in frameworks which run and mask branches. Their algorithm does not allow blocks to contain any control-flow, yielding very small blocks. This can be detrimental to performance under the default "run and mask" assumption made in their work. By focusing on MCMC, we also obtain provable speed-ups under appropriate statistical settings and FSM design.

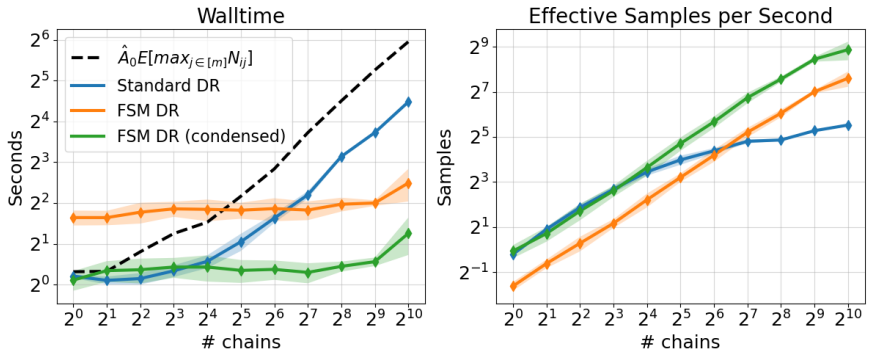

Figure 5: Mean and standard deviation walltimes (LHS) and ESS (RHS) of the symmetric Delayed-Rejection Algorithm (Mira et al., 2001) using Algorithm 1 and our FSM implementation Algorithm 3 (both with `vmap`), on a univariate Gaussian (10 random seeds). To test the effect of the number of states and step function bundling on performance, FSM uses `step` with the while loop body (PROPOSE in Figure 3) split into 4 states, whilst FSM-condensed uses `bundled_step` with the same states. By bundling the steps together, we achieve a $\sim$ 3x efficiency gain.

7 Experiments

The following experiments evaluate FSM implementations of different MCMC algorithms with while loops against their standard (non-FSM) implementations, as well as other MCMC methods in one experiment. All methods (including ours) consist of single chain MCMC algorithms written in JAX, turned into multi-chain methods with `vmap`, and compiled using `jit`. All experiments are run in JAX on a NVIDIA A100 GPU with 32GB CPU memory.

7.1 Delayed-Rejection MH on a Simple Gaussian

We first illustrate basic properties of the FSM conversion on a toy example. The MCMC algorithm used is symmetric Delayed-Rejection Metropolis Hastings (DRMH) (Mira et al., 2001), which is a simple example of a delayed rejection method. Symmetric DRMH modifies the Random-Walk Metropolis-Hastings algorithm by iteratively re-centering the proposal distribution on the rejected sample and resampling until either acceptance occurs or a maximum number of tries M is reached. To ensure detailed balance hods, the acceptance probability formula is adjusted at each step to account for past rejections.

Experimental setup. As a toy problem, we implement symmetric DRMH using (`vmap`'ed) Algorithm 1 (baseline) and Algorithm 3 (ours) to sample from a univariate Gaussian $\mathcal{N}(0, 1)$, varying the number of chains. We use a $\mathcal{N}(x, 0.1)$ proposal distribution with $M = 100$ tries per sample and draw 10,000 samples per chain. Although DRMH has 3 state functions by default (see Figure 3), the INIT and DONE states are

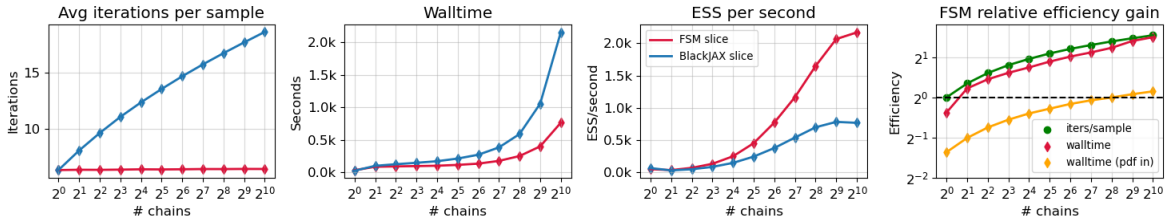

Figure 6: Average results using 10 random seeds (standard deviations too small to show) from drawing 10k posterior samples for the covariance hyperparameters (τ, θ, σ) of a Gaussian Process $Y(x) = f(x) + \epsilon$ on the Real Estate UCI dataset ($n = 411, d = 6$). Blue = BlackJAX elliptical slice, Red = FSM elliptical slice sampler (red). LHS: the average number of sub-iterations (i.e., ellipse contractions) needed to draw a single sample increases from 6 (1 chain) to 18 (1024 chains) for the standard implementation due to synchronization barriers, but remains constant for our FSM. Middle two plots: The FSM can run ∼3x faster by avoiding synchronization barriers, as shown by the Walltime (left-middle) and ESS/S (right-middle). RHS: the ratio of average iterations per sample (i.e. $R(m)$) (green) bounds the obtainable 'efficiency gain' using our FSM, but is nearly obtained in relative walltime when amortizing log-pdf calls (red).

essentially empty, and so just a single state function can be used for the while loop body. This means an appropriate FSM implementation should be able to get close to $R(m)$, up to overheads. To illustrate the importance of designing FSMs with as few (effective) state functions as possible, we implement an FSM which unrolls the while loop body into four different state functions, as well as a (condensed) FSM which uses step function bundling to effectively use a single state.

Results. As the number of chains m increases, the FSM implementations increasingly outperform the standard implementation (see Figure 5). This reflects the increasing synchronization cost of waiting for the slowest chain. The speedups are near an order of magnitude when $m = 1024$ for the condensed FSM. Bundling sees nearly a 3x efficiency gain and enables no performance loss when $m = 1$ and there is no synchronization barrier. Note Standard DR tracks the profile of the estimated $\mathbb{E}[\max_{j\in[m]} N_{1,j}]$, whilst the FSMs track the profile of $\mathbb{E}[N_{1,j}]$ (which is flat). This implies the condensed FSM (i.e. with step bundling) has been able to approximately obtain $R(m)$ up to a constant factor.

7.2 Elliptical Slice Sampling on Real Estate Data

The elliptical slice sampler (introduced in the Example in Section 2.2) has a single while loop, resulting in three state functions (see Figure 3). We compare BlackJAX's implementation to our FSM implementation.

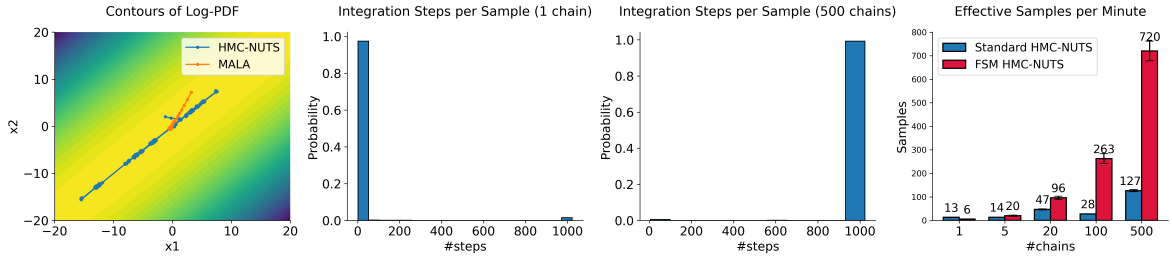

Figure 7: Left: contours of the first two dimensions of a correlated mixture of Gaussians, along with a single chain of HMC-NUTS and MALA. Middle Left: Histogram of the number of integration steps taken per sample for a single NUTS chain. Middle Right: histogram of the maximum number of integration steps taken per sample across 500 chains. Right: Effective samples per minute for the standard BlackJAX HMC-NUTS implementation, and our FSM implementation of HMC-NUTS (average and standard error bars from 5 seeds). The FSM achieves speed ups of nearly an order of magnitude for 100 chains, and or than half an order of magnitude for 500 chains.

Experimental setup. We apply the sampler to infer posteriors on covariance hyperparameters in Gaussian Process Regression, using the UCI repository Real Estate Valuation dataset (Yeh, 2018). This dataset is comprised of $n = 414$ input and output pairs $\mathcal{D}_n = \{\boldsymbol{x}_i, y_i\}_{i=1}^n$, where y_i is the house price of area i, and $\boldsymbol{x}_i \in \mathbb{R}^6$ are house price predictors including house age, spatial co-ordinates, and number of nearby convenience stores. We model $y = f(x) + \epsilon$ and assume $\epsilon \sim \mathcal{N}(0, \sigma^2)$, $f \sim \mathcal{GP}(0, k)$ with kernel $k(x, x') = \tau^2 \exp(-\lambda^2 |x - x'\|^2)$. We use Normal priors $\sigma, \tau, \lambda \sim \mathcal{N}(0, 1)$, (so the ellipse is drawn using $\mathcal{N}(0, I)$) and use the sampler to draw $10k$ samples per chain from the posterior $p(\sigma, \tau, \lambda | \mathcal{D}_n)$, for varying # chains m.

Results are shown in Figure 6. As expected, the BlackJAX implementation suffers from synchronization barriers at every iteration due to using `vmap` with while loops: its average number of iterations per sample increases roughly logarithmically from 6 (1 chain) to 18 (1024 chains), whereas the FSM implementation remains constant. As a result, the FSM significantly improves walltime and ESS/second performance (Figure 6/middle). For instance, when 1024 chains are used, the FSM reduces the time to draw 10k samples per chain from over half an hour to about 10 minutes. The efficiency gain can be measured by the ratio (BlackJAX/FSM) of wall-times (shown in Figure 6/right). As expected, FSM efficiency increases with the number of chains. The greatest efficiency gain occurs precisely where the best ESS/second can be obtained via GPU parallelism. The analysis in Section 4 shows that the efficiency gain is upper-bounded by the ratio of average # iterations per sample for both methods (i.e., $R(m)$). This bound is almost achieved here by amortizing log-pdf calls. This is because (i) roughly 80% of the time is spent in the iterative 'SHRINK' state, and (ii) log-pdf calls dominate computational cost here, so amortizing them ensures the state function cost is similar to the standard implementation. Since the log-pdf is needed in two states,

not amortizing it results in two log-pdf calls per step, and so we lose roughly a factor 2 in relative performance (orange line in Figure 6). These results change with data set size, which determines the cost of log-pdf calls (see Appendix B).

7.3 HMC-NUTS on a high-dimensional correlated MoG

The NUTS variant of Hamiltonian Monte Carlo (Hoffman et al., 2014) adaptively chooses how many steps of Hamiltonian dynamics to simulate when drawing a sample, by checking whether the trajectory has turned back on itself or has diverged due to numerical error. Its iterative implementation in BlackJAX involves two nested while loops: An outer loop that expands the proposal trajectory, and an inner loop that monitors for U-turns and divergence. Converting these while loops into an FSM using our procedure results in five states (Figure 3). We again compare BlackJAX's implementation to our own (using `vmap` for both methods).

Experimental setup. We implement NUTS on a 100-dimensional correlated mixture of Gaussians ($\rho = 0.99$), with the mixture modes placed along the principal direction at $(-5 \cdot \mathbf{1}, \mathbf{0}, 5 \cdot \mathbf{1})$. We use a pre-tuned step-size with acceptance rate ~ 0.85 and set $M = I$ for the mass matrix. We draw $n = 1000$ samples per chain and vary # chains m.

Results. The contours of the log-density and a trajectory of a single NUTS chain (1000 samples) are displayed in Figure 7 (one dot = one sample). The typical distance traveled by NUTS is small (few integration steps), with the occasional large jump (many integration steps) when the momentum sample aligns with the principal direction. The FSM yields speedups of nearly an order of magnitude for $m = 100$, and about half an order of magnitude for $m = 500$. This is reflects the fact that the marginal distribution of integration steps is very skewed (i.e. $R(m)$ is large). As Figure 7 shows, the probability a sample needs less than 20 steps is ~ 0.95 and needs > 1000 steps is ~ 0.01. However, the probability that *at least one* chain needs more than 1000 steps is ~ 0.99. Note that one can avoid sychronization barriers and obtain very high ESS/Sec using a simpler algorithm like MALA, but this fails to explore the distribution (LHS Figure 7).

7.4 Transport Elliptical Slice Sampling for distributions with challenging local geometry

Transport elliptical slice sampling (TESS), due to Cabezas and Nemeth (2023), is a variant of elliptical slice sampling designed for certain challenging local geometries. It essentially uses a normalizing flow T to 'precondition' the distribution π, and does elliptical slice sampling on the transformed distribution $T_{\#}\pi$. Since T is learned to approximately map π to $\mathcal{N}(0, I)$, the geometry of $T_{\#}\pi$ makes it much easier to sample from. Once these samples are obtained, they can be pushed through T^{-1} to recover

Table 1: ESS/Second for different methods on the four benchmark problems used in Cabezas and Nemeth (2023).

	MEADS	CHEES	NeuTra	TESS	TESS-FSM
Predator Prey	1.53	nan	1.59	2.27	**3.80**
Google Stock	480.76	60.19	185.12	1116.31	**2426.16**
German Credit	141.50	**198.99**	186.17	58.95	59.04
BOD	64.846	247.02	130.59	2978.03	**3252.64**

samples from π. TESS achieves particularly good results on distributions with 'funnel' geometries, with which gradient-based methods like HMC tend to struggle (Gorinova et al., 2020).

Experimental setup. TESS has similar overall structure as the elliptical slice sampler, and conversion results in the same 'single loop' FSM in Figure 3. We compare TESS with and without FSM conversion on the four benchmark sampling problems used in Cabezas and Nemeth (2023), which are chosen for their challenging geometries. As baselines we use two recently proposed adaptive HMC variants (MEADS (Hoffman and Sountsov, 2022), and CHEES (Hoffman et al., 2021)) as well as NeuTra (Hoffman et al., 2019), which uses a similar preconditioning strategy to TESS, but with HMC as the sampling algorithm. For all methods, we run 128 chains of 1000 samples each, with each algorithm's hyperparameters pre-tuned using 400 warm-up steps.

Results. In all four cases, TESS-FSM improved ESS/second over TESS. In three these cases TESS-FSM achieved the best ESS/second, and in two cases the speed-up over TESS was roughly by an order of 2. Since all methods considered here are state-of-the-art (SOTA) approaches to sampling from distributions with challenging geometries, this demonstrates our FSM implementation method can be used to improve SOTA performance on such tasks.

HD, PG and PO are supported by the Gatsby Charitable Foundation. This work was partially supported by NSF OAC 2118201.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Ardakani, A., Ardakani, A., and Gross, W. (2020). Training linear finite-state machines. *Advances in Neural Information Processing Systems*, 33:7173–7183.

BlackJax (2019). Sample with multiple chains in parallel.

Brooks, S., Gelman, A., Jones, G., and Meng, X.-L. (2011). *Handbook of markov chain monte carlo*. CRC press.

Cabezas, A., Corenflos, A., Lao, J., Louf, R., Carnec, A., Chaudhari, K., Cohn-Gordon, R., Coullon, J., Deng, W., Duffield, S., et al. (2024). Blackjax: Composable bayesian inference in jax. *arXiv preprint arXiv:2402.10797*.

Cabezas, A. and Nemeth, C. (2023). Transport elliptical slice sampling. In *International Conference on Artificial Intelligence and Statistics*, pages 3664–3676. PMLR.

Cechin, A. L., Regina, D., Simon, P., and Stertz, K. (2003). State automata extraction from recurrent neural nets using k-means and fuzzy clustering. In *23rd International Conference of the Chilean Computer Science Society, 2003. SCCC 2003. Proceedings.*, pages 73–78. IEEE.

Fan, J., Jiang, B., and Sun, Q. (2021). Hoeffding's inequality for general markov chains and its applications to statistical learning. *Journal of Machine Learning Research*, 22(139):1–35.

Flax (2023). Flax: A neural network library for jax. `https://github.com/google/flax`. Accessed: 2025-01-30.

Gorinova, M., Moore, D., and Hoffman, M. (2020). Automatic reparameterisation of probabilistic programs. In *International Conference on Machine Learning*, pages 3648–3657. PMLR.

Hoffman, M., Radul, A., and Sountsov, P. (2021). An adaptive-mcmc scheme for setting trajectory lengths in hamiltonian monte carlo. In *International Conference on Artificial Intelligence and Statistics*, pages 3907–3915. PMLR.

Hoffman, M., Sountsov, P., Dillon, J. V., Langmore, I., Tran, D., and Vasudevan, S. (2019). Neutra-lizing bad geometry in hamiltonian monte carlo using neural transport. *arXiv preprint arXiv:1903.03704*.

Hoffman, M. D., Gelman, A., et al. (2014). The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623.

Hoffman, M. D. and Sountsov, P. (2022). Tuning-free generalized hamiltonian monte carlo. In *International conference on artificial intelligence and statistics*, pages 7799–7813. PMLR.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65.

Koul, A., Greydanus, S., and Fern, A. (2018). Learning finite state representations of recurrent policy networks. *arXiv preprint arXiv:1811.12530*.

Lao, J., Suter, C., Langmore, I., Chimisov, C., Saxena, A., Sountsov, P., Moore, D., Saurous, R. A., Hoffman, M. D., and Dillon, J. V. (2020). tfp. mcmc: Modern markov chain monte carlo tools built for modern hardware. *arXiv preprint arXiv:2002.01184*.

Mira, A. et al. (2001). On metropolis-hastings algorithms with delayed rejection. *Metron*, 59(3-4):231–241.

Modi, C., Barnett, A., and Carpenter, B. (2024). Delayed rejection hamiltonian monte carlo for sampling multiscale distributions. *Bayesian Analysis*, 19(3):815–842.

Murray, I., Adams, R., and MacKay, D. (2010). Elliptical slice sampling. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 541–548. JMLR Workshop and Conference Proceedings.

Muškardin, E., Aichernig, B. K., Pill, I., and Tappler, M. (2022). Learning finite state models from recurrent neural networks. In *International Conference on Integrated Formal Methods*, pages 229–248. Springer.

Neal, R. M. (2003). Slice sampling. *The annals of statistics*, 31(3):705–767.

Oktay, D., Mirramezani, M., Medina, E., and Adams, R. (2023). Neuromechanical autoencoders: Learning to couple elastic and neural network nonlinearity. In *International Conference on Learning Representations (ICLR 2023)*.

Pfau, D., Spencer, J. S., Matthews, A. G., and Foulkes, W. M. C. (2020). Ab initio solution of the many-electron schrödinger equation with deep neural networks. *Physical review research*, 2(3):033429.

Phan, D., Pradhan, N., and Jankowiak, M. (2019). Composable effects for flexible and accelerated probabilistic programming in numpyro. *arXiv preprint arXiv:1912.11554*.

Qiu, Y., Zhang, L., and Wang, X. (2019). Unbiased contrastive divergence algorithm for training energy-based latent variable models. In *International Conference on Learning Representations*.

Radul, A., Patton, B., Maclaurin, D., Hoffman, M., and A Saurous, R. (2020). Automatically batching control-intensive programs for modern accelerators. *Proceedings of Machine Learning and Systems*, 2:390–399.

Rudolf, D. (2011). Explicit error bounds for markov chain monte carlo. *arXiv preprint arXiv:1108.3201*.

Ruiz, F. J., Valera, I., Svensson, L., and Perez-Cruz, F. (2018). Infinite factorial finite state machine for blind multiuser channel estimation. *IEEE Transactions on Cognitive Communications and Networking*, 4(2):177–191.

Schoenholz, S. S. and Cubuk, E. D. (2021). Jax, md a framework for differentiable physics. *Journal of Statistical Mechanics: Theory and Experiment*, 2021(12):124016.

Sountsov, P., Carroll, C., and Hoffman, M. D. (2024). Running markov chain monte carlo on modern hardware and software. *arXiv preprint arXiv:2411.04260*.

Svete, A. and Cotterell, R. (2023). Recurrent neural language models as probabilistic finite-state automata. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 8069–8086.

Tiňo, P., Horne, B. G., Giles, C. L., and Collingwood, P. C. (1998). Finite state machines and recurrent neural networks—automata and dynamical systems approaches. In *Neural networks and pattern recognition*, pages 171–219. Elsevier.

Vershynin, R. (2018). *High-dimensional probability: An introduction with applications in data science*, volume 47. Cambridge university press.

Yeh, I.-C. (2018). Real Estate Valuation. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5J30W.

Zeng, Z., Goodman, R. M., and Smyth, P. (1993). Learning finite state machines with self-clustering recurrent networks. *Neural Computation*, 5(6):976–990.

A Mathematical appendix

A.1 Proof of Theorem 4.1

Proof. The result broadly follows from known Hoeffding bounds for Markov chains. For clarity we restate the relevant result below[5], using our notation and set-up.

Proposition A.1 (Hoeffding Bound for Markov Chains - Theorem 1 in Fan et al. (2021))**.** *Let $(Z_i)_{i\geq 1}$ be a Markov Chain with measurable state space $\mathcal{Z}$ and stationary distribution π, and let $f : \mathcal{Z} \to [0, B]$ be measurable and bounded. Then, for any $\epsilon > 0$,*

$$\mathbb{P}_\pi\left(\left|\sum_{i=1}^{n} f(Z_i) - n\mathbb{E}_\pi f(Z_1)\right| \geq \epsilon\right) \leq 2\exp\left(-\frac{1-\lambda}{1+\lambda}\frac{2\epsilon^2}{B^2}\right) \tag{12}$$

where λ is the spectral gap of π (as defined in Fan et al. (2021)) which quantifies the speed of convergence of the Markov chain towards π (Rudolf, 2011).

Now we are ready to prove our results. We start with $C_0(m, n)$. First, note that since $(\boldsymbol{X}_{i,j}, N_{i,j})_{i\geq 1}$ is a Markov chain with stationary distribution π for every chain j, then $(\{\boldsymbol{X}_{i,j}, N_{i,j}\}_{j=1}^m)_{i\geq 1}$ is also a Markov chain with stationary distribution $\pi^m := \pi \otimes \pi \underbrace{\otimes ... \otimes}_{m-1} \pi$, because the chains are independent. Therefore, if we set $Z_i := \{\boldsymbol{X}_{i,j}, N_{i,j}\}_{j=1}^m$ and $f(Z_i) := \max_{j\in[m]}(N_{i,j}) \in [0, B]$, we get by an application of Proposition A.1 that

$$\mathbb{P}_\pi\left(\left|\sum_{i=1}^{n} \max_{j\in[m]}(N_{i,j}) - n\mathbb{E}_\pi \max_{j\in[m]}(N_{1,j})\right| \geq \epsilon\right) \leq 2\exp\left(-\frac{1-\lambda}{1+\lambda}\frac{2\epsilon^2}{nB^2}\right) \tag{13}$$

Setting $\delta = 2\exp\left(-\frac{1-\lambda}{1+\lambda}\frac{2\epsilon^2}{nB^2}\right)$ and re-arranging, we get that with probability $1-\delta$ (under the stationary distribution π),

$$\left|\sum_{i=1}^{n} \max_{j\in[m]}(N_{i,j}) - n\mathbb{E}_\pi \max_{j\in[m]}(N_{1,j})\right| \leq B\sqrt{\frac{n(1+\lambda)}{2(1-\lambda)}\ln\left(\frac{2}{\delta}\right)} \tag{14}$$

Multiplying by $B_0(m)$ and dividing by n on both sides, and adding and subtracting $A_0(m)$ from the LHS gives us the result for C_0

[5]We note that Fan et al. (2021) only present a one-sided bound, but by standard symmetry arguments this immediately implies the above two-sided bound.

$$\left| B_0(m)\frac{1}{n}\sum_{i=1}^{n}\max_{j\in[m]}(N_{i,j}) - B_0(m)\mathbb{E}_\pi \max_{j\in[m]}(N_{1,j})\right| \leq B_0(m)B\sqrt{\frac{(1+\lambda)}{2n(1-\lambda)}\ln\left(\frac{2}{\delta}\right)} \tag{15}$$

$$\left| B_0(m)\frac{1}{n}\sum_{i=1}^{n}\max_{j\in[m]}(N_{i,j}) \pm A_0(m) - B_0(m)\mathbb{E}_\pi \max_{j\in[m]}(N_{1,j})\right| \leq B_0(m)B\sqrt{\frac{(1+\lambda)}{2n(1-\lambda)}\ln\left(\frac{2}{\delta}\right)} \tag{16}$$

$$\left| C_0(m,n) - A_0(m) - B_0(m)\mathbb{E}_\pi \max_{j\in[m]}(N_{1,j})\right| \leq B_0(m)B\sqrt{\frac{(1+\lambda)}{2n(1-\lambda)}\ln\left(\frac{2}{\delta}\right)} \tag{17}$$

Now we follow similar steps for $C_F(m,n)$. To start, we bound the distance from $\max_{j\in m}\frac{1}{n}\sum_{i=1}^{n}N_{i,j}$ and $\mathbb{E}_\pi[N_{11}]$ in terms of a sum of individual distances using the union bound.

$$\begin{aligned}
\mathbb{P}_\pi\left(\left|\max_{j\in m}\frac{1}{n}\sum_{i=1}^{n}N_{i,j} - \mathbb{E}_\pi N_{11}\right| \geq \epsilon\right) &= \mathbb{P}_\pi\left(\max_{j\in m}\frac{1}{n}\sum_{i=1}^{n}N_{i,j} - \mathbb{E}_\pi N_{11} \geq \epsilon\right) \\
&\quad + \mathbb{P}_\pi\left(\max_{j\in m}\frac{1}{n}\sum_{i=1}^{n}N_{i,j} - \mathbb{E}_\pi N_{11} \leq -\epsilon\right) && (18)\\
&= \mathbb{P}_\pi\left(\bigcup_{j=1}^{m}\left\{\frac{1}{n}\sum_{i=1}^{n}N_{i,j} - \mathbb{E}_\pi N_{11} \geq \epsilon\right\}\right) \\
&\quad + \mathbb{P}_\pi\left(\bigcup_{j=1}^{m}\left\{\frac{1}{n}\sum_{i=1}^{n}N_{i,j} - \mathbb{E}_\pi N_{11} \leq -\epsilon\right\}\right) && (19)\\
&\leq \sum_{j=1}^{m}\left[\mathbb{P}_\pi\left(\frac{1}{n}\sum_{i=1}^{n}N_{i,j} - \mathbb{E}_\pi N_{11} \geq \epsilon\right)\right. \\
&\quad \left. + \mathbb{P}_\pi\left(\frac{1}{n}\sum_{i=1}^{n}N_{i,j} - \mathbb{E}_\pi N_{11} \leq -\epsilon\right)\right] && (20)\\
&= \sum_{j=1}^{m}\mathbb{P}_\pi\left(\left|\frac{1}{n}\sum_{i=1}^{n}N_{i,j} - \mathbb{E}_\pi N_{11}\right| \geq \epsilon\right) && (21)\\
&= m\mathbb{P}_\pi\left(\left|\frac{1}{n}\sum_{i=1}^{n}N_{i,1} - \mathbb{E}_\pi N_{11}\right| \geq \epsilon\right) && (22)\\
&= m\mathbb{P}_\pi\left(\left|\sum_{i=1}^{n}N_{i,1} - n\mathbb{E}_\pi N_{11}\right| \geq n\epsilon\right) && (23)
\end{aligned}$$

Applying Proposition A.1 on the Markov Chain $(\boldsymbol{X}_{1,i}, N_{1,i})_{i\geq 1}$ with $f(\boldsymbol{X}_{1,i}, N_{1,i}) =$

$N_{1,i} \in [0, B]$ and following the same steps as for $C_0(m, n)$, we similarly recover

$$\left| C_F(m,n) - A_F(m) - B_F(m)\mathbb{E}_\pi \max_{j\in[m]}(N_{1,j}) \right| \leq B_F(m)B\sqrt{\frac{(1+\lambda)}{2n(1-\lambda)}\ln\left(\frac{2m}{\delta}\right)} \quad (24)$$

which is the result in the Theorem. □

Proposition A.2. *Fix $m, K \in \mathbb{N}\backslash\{0\}$ and let $\mathbb{P}_N$ be a probability measure on $\mathbb{R}_+$ strictly positive first moment. Suppose (i) $N_1, ..., N_m \overset{iid}{\sim} \mathbb{P}_N$, (ii) $c_1(m), ..., c_K(m) \geq 0$ and (iii) $\alpha \in [\max_{j\in[K]} c_j(m)/\sum_{j\in[K]} c_j(m), 1]$. Then, we have*

$$E(m) := \frac{c_{\neg k}(m) + c_k(m)\mathbb{E}\max_{j\in[K]} N_j}{\alpha(c_{\neg k}(m) + c_k(m))(K - 1 + \mathbb{E}N_1)} \leq \frac{\mathbb{E}\max_{j\in[K]} N_j}{\mathbb{E}N_1} =: R(m) \quad (25)$$

where $c_{\neg k}(m) = \sum_{j\neq k} c_j(m)$. The bound is tight.

Proof. Note $\frac{a+b}{c+d} = \frac{a}{c}\gamma + \frac{b}{d}(1-\gamma)$ where $\gamma = \frac{c}{c+d}$ for any $a, b, c, d \in \mathbb{R}$. Applying this to our case, we get

$$E(m) = \frac{c_{\neg k}(m)}{\alpha(c_{\neg k}(m) + c_k(m))(K-1)}w + \frac{c_k(m)}{\alpha(c_{\neg k}(m) + c_k(m))}R(m)(1-w) \quad (26)$$

where $w = \frac{\alpha(c_{\neg k}(m)+c_k(m))}{\alpha(c_{\neg k}(m)+c_k(m))(K-1+\mathbb{E}N_1)} \in [0, 1]$. Now we split into two cases for $K = 1$ and $K > 1$. For the case $K = 1$ we only have a single iterative state and so $C_{\neg k}(m) = 0$, $\alpha = 1$. In this case we trivially have $E(m) = R(m)$. Now suppose $K > 1$. In this case, since $\alpha(c_{\neg k}(m) + c_k(m)) \geq \max_{j\in[K]} c_j(m)$, we have

$$\frac{c_k(m)}{\alpha(c_{\neg k}(m) + c_k(m))} \leq \frac{c_k(m)}{\max_{j\in[K]} c_j(m)} \leq 1 \quad (27)$$

Which means we can bound $E(m)$ by removing the term in front of $R(m)$,

$$E(m) \leq \frac{c_{\neg k}(m)}{\alpha(c_{\neg k}(m) + c_k(m))(K-1)}w + R(m)(1-w) \quad (28)$$

By the same logic, we have

$$\frac{c_{\neg k}(m)}{\alpha(c_{\neg k}(m) + c_k(m))(K-1)} \leq \frac{c_{\neg k}(m)}{\max_{j\in[K]} c_j(m)(K-1)} = \frac{\sum_{j\neq k} c_j(m)}{\max_{j\in[K]} c_j(m)(K-1)} \leq 1 \quad (29)$$

which means

$$E(m) \leq w + R(m)(1-w) \quad (30)$$
$$\leq R(m) \quad (31)$$

Where the last line uses the fact that $R(m) \geq 1$ since $N_1 \geq 0$. The bound is tight because when $K = 1$ we have $E(m) = R(m)$. This completes the proof

□

B Additional Details and Results

B.1 Algorithms

Note here we use $\tilde{\boldsymbol{x}}$ to denote a batch of inputs $[\boldsymbol{x}_1, ..., \boldsymbol{x}_m]$ for m different chains, and the same for other variables.

Algorithm 6 Vectorized MCMC algorithm with `vmap(sample)` function

Inputs: sample $\tilde{\boldsymbol{x}}_0$, seed $\tilde{r}_0$
for $i \in \{1, ..., n\}$ **do**
  generate $\tilde{\boldsymbol{x}}_i, \tilde{r}_i \leftarrow$ `vmap(sample)`$(\tilde{\boldsymbol{x}}_{i-1}, \tilde{r}_{i-1})$
end for
return $\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_n$

Algorithm 7 Vectorized FSM MCMC algorithm with `vmap(step)` function

input: initial value $\tilde{\boldsymbol{x}}_0$, # samples n
initialize: $\tilde{z} =$ `vmap(init)`$(\tilde{\boldsymbol{x}}_0), \tilde{X} = \text{list}(), \tilde{B} = \text{list}()$
Set $\tilde{t} = 0$ and $\tilde{k} = 0$
while $\min_{\tilde{t}_i \in \tilde{t}}\{\tilde{t}_i\} < n$ **do**
  $(\tilde{k}, \tilde{z}, \widetilde{\texttt{isSample}}) \leftarrow$ `vmap(step)`$(\tilde{k}, \tilde{z})$
  append current sample value $\tilde{\boldsymbol{x}}$ stored in $\tilde{z}$ to $\tilde{X}$
  append $\widetilde{\texttt{isSample}}$ to $\tilde{B}$
  update sample counter $\tilde{t} \leftarrow \tilde{t} + \widetilde{\texttt{isSample}}$
end while
return $\tilde{X}[\tilde{B}]$

Algorithm 8 Transition kernel for elliptical slice sampler with log-pdf $\log p$, covariance matrix Σ.

Input: Sample $\boldsymbol{x}$
Choose ellipse $\boldsymbol{\nu} \sim \mathcal{N}(\mathbf{0}, \Sigma)$
Set threshold $\log y \leftarrow \log p(\boldsymbol{x}) + \log u : u \sim \mathcal{U}[0, 1]$
Set bracket $[\theta_{\min}, \theta_{\max}] \leftarrow [\theta - 2\pi, \theta] : \theta \sim \mathcal{U}[0, 2\pi]$
Make proposal $\boldsymbol{x}' \leftarrow \boldsymbol{x} \cos\theta + \boldsymbol{\nu} \sin\theta$
while $\log p(\boldsymbol{x}') > \log y$ **do**
    Shrink bracket and update proposal:
    if $\theta < 0$ **then**
        $\theta_{\min} \leftarrow \theta$
    else
        $\theta_{\max} \leftarrow \theta$
    end if
    $\boldsymbol{x}' \leftarrow \boldsymbol{x} \cos\theta + \boldsymbol{\nu} \sin\theta : \theta \sim \mathcal{U}[\theta_{\min}, \theta_{\max}]$
end while
Return $\boldsymbol{x}'$

B.2 FSM Design

Detailed Construction in the two sequential while-loops case Here, we describe the FSM construction for programs with two sequential while-loops in detail. Following the constructions introduced in the main body, let us note

- $\mathcal{F}_1 \coloneqq (\{S_{11}, S_{12}, S_{13}\}, \mathcal{Z}, \delta_1, S_{11}, S_{13})$ the FSM associated to B_1
- $\mathcal{F}_2 \coloneqq (\{S_{21}, S_{22}, S_{23}\}, \mathcal{Z}, \delta_2, S_{21}, S_{23})$ the FSM associated to B_2

By construction, S_{21} is empty. Both FSMs share the same input space, which is the set of local variables values associated to the original `sample` function.

Then, the resulting FSM representation of `sample` is $(\mathcal{S}, \mathcal{Z}, \delta, S_{11}, S_{23})$, where

- $\mathcal{S} = \{S_{11}, S_{12}, S_{13}, S_{22}, S_{23}\}$.
- Transition function δ defined as

$$\delta(S, z) = \begin{cases} \delta_1(S, z) & \text{if} \quad S \in \{S_{11}, S_{12}\} \\ \delta_2(S_{21}, z) & \text{if} \quad S = S_{13} \\ \delta_2(S, z) & \text{if} \quad S = \{S_{22}, S_{23}\}, \end{cases} \tag{32}$$

whose construction illustrates the "FSM stitching" operation performed.

Detailed Construction in the two nested while-loop case Here, we describe the FSM construction for programs with two nested while-loops. Following the constructions introduced in the main body, let us note
$\mathcal{F}_i \coloneqq (\{S_{i1}, S_{i2}, S_{i3}\}, \mathcal{Z}, \delta_i, S_{i1}, S_{i3})$ the (inner) FSM associated to B_2.
Then, the resulting FSM representation of `sample` is $(\mathcal{S}_o, \mathcal{Z}, \delta_o, S_1, S_3)$, where

- $\mathcal{S}_o = \{S_1, S_{i1}, S_{i2}, S_{i3}, S_3\}$.
- Transition function δ_o defined as
  - $\delta_o(S_1, z) = \delta_o(S_{i3}, z)$ runs the outer while-loop condition on z, goes to S_{i1} if True, and S_3 otherwise.
  - $\delta_o(S, z) = \delta_i(S, z)$ if $S \in \{S_{i1}, S_{i2}\}$

B.3 Additional Implementation Details

FSM wrapper design. In our experiments, we use a native Python while loop in Algorithm 3, and use a `jax.lax.scan` to run the FSM step function for blocks of 100 steps, when drawing $n > 100$ samples. This gives us the flexibility to store the results in dynamically shaped lists/arrays and transport to the CPU for faster array slicing when CPU memory is available, whilst still reaping the benefits of JIT compilation.

Compilation. We JIT compile both `vmap(step)` and `vmap(sample)` functions for each MCMC algorithm implementation. For Delayed Rejection and the Elliptical Slice Sampler (with $n = 25$) we remove compilation time to get more accurate results or the runs with small numbers of chains m, due to the low cost of the computations involved.

JAX implementation. When comparing to non-FSM implementations, we used BlackJAX (Cabezas et al., 2024) for fair comparison with our method, since we use BlackJAX primitives for the key computations in some of our algorithms (e.g. HMC-NUTS). Where a BlackJAX implementation was not available (e.g. Delayed Rejection), we wrote our own for fair comparison with our FSM implementation.

B.4 Additional Results

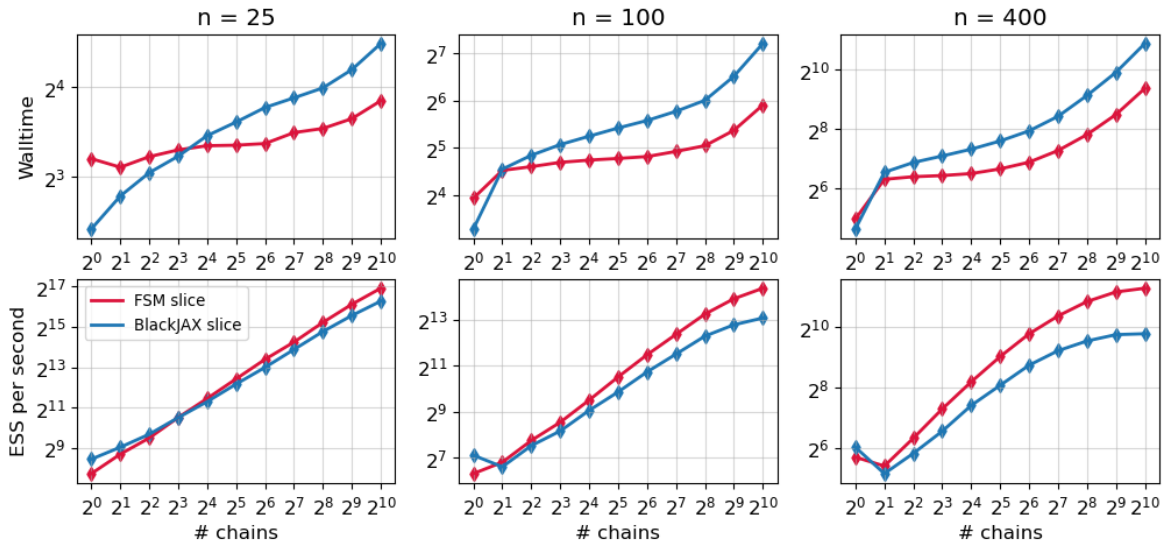

Figure 8: Walltimes and ESS per second using the Elliptical Slice Sampler (non-FSM vs FSM implementation) on the Real Estate Dataset described in Section 7, when restricting the dataset to the first $n \in \{25, 100, 400\}$ datapoints. For each dataset size, the best walltime and ESS/second is obtained by both implementations when using $m = 1024$ chains. Our FSM implementation can obtain the greatest efficiency for all dataset sizes. As the log-likelihood cost increases (the log-likelihood in GPR regression costs $\mathcal{O}(n^3)$), we see the FSM efficiency gain increase, reflecting the benefits of amortization.

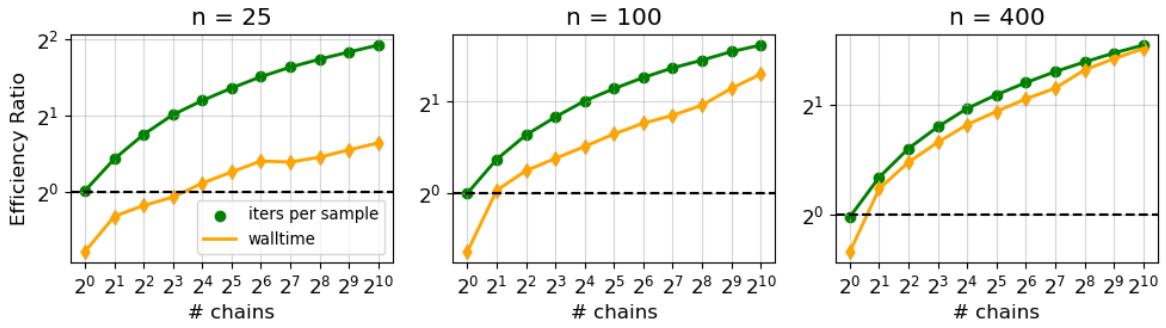

Figure 9: Efficiency Ratio of our elliptical slice FSM against BlackJAX's elliptical slice algorithm (as measured by estimated $R(m) = \mathbb{E}[\max_{j\in[m]} N_j]/\mathbb{E}[N_1]$ (i.e. iters per sample) and walltime) on the Real Estate Dataset described in Section 7 when restricting the dataset to the first $n \in \{25, 100, 400\}$ datapoints. The relative efficiency of the FSM improves as the number of chains used increase, and as the log-likelihood cost increases. When $n = 400$, we almost achieve the theoretical bound $R(m)$ in speed-ups.

Table 2: Effective Sample Size per Second and Kernel Stein Discrepancy (KSD) for transport elliptical slice sampling (TESS (Cabezas and Nemeth, 2023)) with and without our FSM implementation against NeuTra (Hoffman et al., 2019), which also uses preconditioning flows, and two adaptive HMC variants (MEADS (Hoffman and Sountsov, 2022) and CHEES (Hoffman et al., 2021)). Each method uses 128 chains where each chain draws 400 burn-in samples for tuning followed by 1000 warm samples. Our FSM implementation of TESS achieved the best ESS/Second on three out of four problems, in two cases improving the state-of-the-art by roughly a factor of 2.

Dataset	Effective Sample Size per Second					KSD				
	MEADS	CHEES	NeuTra	TESS	TESS-FSM	MEADS	CHEES	NeuTra	TESS	TESS-FSM
Predator Prey	1.53	nan	1.59	2.27	**3.80**	1.13e+6	1e+6	4.5e+5	**4.37**	7.18
Google Stock	480.76	60.19	185.12	1116.31	**2426.16**	1.83	**0.57**	2.73	0.84	0.78
German Credit	141.50	**198.99**	186.17	58.95	59.04	4.68	6.16	4.58	4.37	**4.23**
BOD	64.846	247.02	130.59	2978.03	**3252.64**	2.61e+14	9.63e+15	1.39e+14	49.27	**10.80**