# STAT 547: Bayesian Workflow

Charles C. Margossian
University of British Columbia
Winter 2026
https://charlesm93.github.io/stat_547/

**DRAFT**

# 5 Bayesian Modeling and Markov chain Monte Carlo on Modern Hardware

Much of the recent progress in compute power has come from parallel-processing, in particular the development of graphical processing units (GPUs). GPUs were originally designed for graphical rendering and then developed for numerical computation, specifically the matrix operations that underlie neural networks.

In this section, we'll examine the ways in which Bayesian modeling and MCMC can exploit GPUs and more generally parallel accelerators. We'll see that popular MCMC workflows often fail to leverage parallelization and that certain implementations, while remarkably effective on classic hardware such as central processing units (CPUs), are not mindful of the engineering constraints imposed by GPUs.

Perhaps most important takeaway from this lesson is to realize that parallel accelerators do not offer "arbitrary parallelization", i.e. the ability to perform completely distinct operations on different cores. Rather, accelerators are most effective when using Single Instruction Multiple Data (SIMD) instructions, wherein each core performs the same operation but on a different "data" or input.

## 5.1 Parallelizing the model

Let's examine some examples of Bayesian models, i.e. functions which return (log) joint densities, and see how compatible these functions are with parallelization and SIMD instructions.

**Logistic regression.** Consider a model with $N$ observations $(x_n, y_n)$, where $x_n \in \mathbb{R}^D$ and $y_n \in \{0, 1\}$:

$$\begin{aligned} \beta_d &\sim \text{normal}(0, 1) \\ y_n &\sim \text{Bernoulli}(\sigma(\beta^T x_n)), \end{aligned} \tag{1}$$

where $\sigma$ is the logistic function. There are several opportunities for parallelization, when evaluating $\log p(y, \beta; x_n)$. Due to the marginal independence of the $\beta_d$'s and the conditional independence of the $y$'s,

$$\log p(y, \beta; x_n) = \sum_{d=1}^{D} \log \text{Normal}(\beta_d; 0, 1) + \sum_{n=1}^{N} \log \text{Bernoulli}(y_n; \beta^T x_n). \tag{2}$$

The terms inside each sum can be parallelized and calculated using SIMD instructions. On the other hand, we cannot parallelize *across* the sums with SIMD instructions, since

different cores would need to perform different operations. Hence, each sum is parallelized and performed sequentially.

Fortunately, we do not need to explicitly dictate how the parallelization happens on GPU, provided we use specialized packages such as `JAX` or `PyTorch`.

**Pharmacokinetic model.** We now consider a more sophisticated example (see your homework assignment for additional details). In this example, we have observations $y_{1:T}$ on a patient, physiological parameters $\theta$, and a parameter $\sigma$ for the observational model. The generative model is,

$$\begin{aligned} \theta &\sim p(\theta) \\ \sigma &\sim p(\sigma) \\ c_t &= f(\theta, t) \\ y_t &\sim \text{logNormal}(\log c_t, \sigma), \end{aligned} \tag{3}$$

where $f$ is a function that returns the solution to a differential equation parameterized by $\theta$ at time $t$.

Once again, there are opportunities for parallelization. If the prior $p(\theta)$ factorizes into identical factors with different inputs, the calculation of $\log p(\theta)$ can be parallelized with SIMD instructions. Similarly, we can take advantage of the conditional independence of the $y_t$'s when computing $\log p(y_{1:T} \mid c_{1:T}, \sigma)$.

The function $f$ requires solving a differential equation. Depending on how we solve the differential equation, there may be some opportunities for parallelization. A numerical integrator usually requires doing sequential operations and so is not immediately amiable to parallelization (although some operation within the sequence may be).

**Population pharmacokinetic model.** An extension of the pharmacokinetic model considers observations collected across multiple patients. This leads to a hierarchical or population model, wherein each patient has their own individual parameters $\theta_n$, with a prior driven by a population level parameters $\phi$.

$$\begin{aligned} \phi &\sim p(\phi) \\ \theta_n &\sim p(\theta_n \mid \phi) \\ \sigma &\sim p(\sigma) \\ c_{nt} &= f(\theta_n, t) \\ y_{nt} &\sim \text{logNormal}(\log c_{nt}, \sigma). \end{aligned} \tag{4}$$

In addition to the parallelization opportunities we identified previously, it also seems natural to parallelize across patients.

This is straightforward to do if computing $f$ involves the same operations for all patients, as will be the case if we use:

- an analytical expression for $f$.
- a matrix exponential, if the ODE is linear.

- a numerical integrator with a fixed step size.

On the other hand, if we use a numerical integrator with an adaptive step size, the number of operations to compute $f$ likely differs between patients. This is in fact what we can expect from most standard numerical integrators (like the Runge-Kutta $4^{\text{th}}/5^{\text{th}}$ order in `Stan`).

In practice, population pharmacokinetic models are often fitted on a CPU cluster, with each core handling one patient in a non-SIMD fashion. `Stan` is well equipped to handle this type of parallelization, notably via the `reduce_sum` function.

Finding ways to fit population pharmacokinetic models on GPUs largely remains an open question. Some ideas:

- Use a fixed step size for the ODE solver, with a carefully chosen step size using information during the warmup.
- Use a surrogate model to solve the ODE, for example with a neural network.

**Remark 1.** *In an automatic differentiation framework, parallelization considerations for evaluating* $\log p(\theta, y)$ *naturally extend to computing the gradient* $\nabla \log p(\theta, y)$.

## 5.2 Parallelizing across chains

A general strategy to leverage access to multiple cores is to run Markov chains in parallel. Running multiple chains has several benefits:

- It increases the number of samples, which in turn reduces the variance of our Monte Carlo estimators.
- It is necessary to compute certain diagnostics, such as $\widehat{R}$.
- It can lead to adaptation strategies which share information between multiple chains.

In a classical framework, it is common to run 4 to 8 chains on a CPU.

With a GPU, it is straightforward to run hundreds or even thousands of Markov chains in parallel, especially with the help of packages such as `TensorFlow Probability` and `BlackJax`. (My personal record is 16,000.)

### 5.2.1 The Many-Short-Chains Regime

The prospect of running many chains in parallel suggests a strategy whereby the variance of Monte Carlo estimators is entirely handled by a large number of chains.

Suppose we run $M$ independent Markov chains of length $N$ and assume they are sufficiently warmed up that the bias is negligible. Let $\hat{f}$ be the Monte Carlo estimator obtained by averaging samples across all chains,

$$\hat{f} = \frac{1}{MN} \sum_{m=1}^{M} \sum_{n=1}^{N} f\left(z^{(mn)}\right). \tag{5}$$

Then,

$$\mathrm{Var}\hat{f} \leq \frac{\mathrm{Var}f}{M}, \tag{6}$$

and this upper-bound is attained when $N = 1$, that is each Markov chain contributes exactly one sample. Interpreting the ESS as a ratio of variances, we can say that the number of chains lower-bounds the ESS.

For many problems, it suffices to achieve an ESS of $\sim$100 or perhaps a few hundreds. For models of moderate dimension, it is well within a GPU's capacity to run 100 Markov chains in parallel. In that case, retaining one sample per chain is enough to achieve our target variance.

This configuration, wherein the variance is almost entirely handled by the number of chains rather than the length of the chain is the *many-short-chains* regime.

Of course, while it is possible to average out variance, we cannot average out bias. That is, regardless of the number of chains, we still need to warmup each Markov chain. A question then is: how quickly can we reduce the bias of our Monte Carlo estimator?

To build some intuition, let's consider the regime of geometric ergodicity where,

$$\left|\mathbb{E}f\left(z^{(N)} \mid z_0\right) - \mathbb{E}f\right| \leq b(z_0)\lambda^N, \tag{7}$$

for some initial point $z_0$ and $\lambda \in (0, 1)$. Integrating over the initial distribution $\pi_0$ from which $z_0$ is drawn and using Jensen's inequality,

$$\begin{aligned}
\left|\mathbb{E}f\left(z^{(N)}\right) - \mathbb{E}f\right| &\leq \left|\mathbb{E}\left(\mathbb{E}f\left(z^{(N)} \mid z_0\right)\right) - \mathbb{E}f\right| \\
&\leq \mathbb{E}\left|\left(\mathbb{E}f\left(z^{(N)} \mid z_0\right)\right) - \mathbb{E}f\right| \\
&= \mathbb{E}(b(z_0))\lambda^N.
\end{aligned} \tag{8}$$

Hence, the squared bias decays at an exponential rate $\mathcal{O}(\lambda^{2N})$, provided we discard early samples (otherwise, it only decreases at a quadratic rate).

By contrast, the variance decreases at a linear rate $\mathcal{O}(1/N)$, with a constant which is 1 if we draw independent samples but which is otherwise larger. For $N$ sufficiently large, we have that

$$\lambda^{2N} \leq 1/N. \tag{9}$$

In fact, for $\lambda \leq 0.841$, the above inequality holds for any $N \geq 1$ and within 20 iterations, the squared bias is already two orders of magnitude smaller than the variance. Hence, provided $\lambda$ does not approach 1, decreasing the squared bias is much cheaper than decreasing the variance.

Moreover, the many-short-chains strategy holds the promise of turning an algorithm whose squared error decreases linearly with the length of the Markov chains to one whose error decreases exponentially with the length of the warmup, at least in some regimes.

### 5.2.2 Diagnostics and performance metrics

The many-short-chains regime has implications on how we measure the performance of MCMC.

Many analysis of MCMC focus on the asymptotic variance. For example, the most commonly reported empirical metric for the performance of an MCMC algorithm is the ESS/operation or ESS/s. This makes sense if we're running one long chain, since the the bias is negligible by the time we control the variance.

The many-short-chains regime flips the script: the variance is negligible (because we have many chains) and so the bias becomes the computational bottleneck. This is an invitation to revise how we analyze MCMC.

**Example: $\widehat{R}$ convergence diagnostics.** Recall the $\widehat{R}$ statistics (module 2),

$$\widehat{R} = \sqrt{\frac{N-1}{N} + \frac{\widehat{B}}{\widehat{W}}}, \tag{10}$$

where $\widehat{B}$ is the sample variance of the *per chain* Monte Carlo estimator,

$$\widehat{B} = \frac{1}{M-1} \sum_{m=1}^{M} \left( \hat{f}_N^{(m)} - \bar{f}_N^{(\cdot)} \right)^2, \tag{11}$$

and $\widehat{W}$ is the average within-chain sample variance,

$$\widehat{W} = \frac{1}{M} \sum_{m=1}^{M} \frac{1}{N-1} \sum_{n=1}^{N} \left( \hat{f}\left(z^{(nm)}\right) - \hat{f}_N^{(m)} \right)^2. \tag{12}$$

A diagnostic for convergence is to check that $\widehat{R}$ is sufficiently small, for example, $\widehat{R} \leq 1.01$.

Unfortunately, in the many-short-chains regime, the per chain variance is never small and, no matter how long the warmup phase, $\widehat{R}$ remains large. This is true even if we achieved a low bias with an adequately long warmup and a small variance with a large number of chains.

We saw that as the number of chains increases and in particular as $M \to \infty$, $\widehat{B}$ is a consistent estimator of the *per chain* variance, which admits the following decomposition,

$$\mathrm{Var} \hat{f}_N^{(m)} = \underbrace{\mathrm{Var}\mathbb{E}\left( \hat{f}_N^{(m)} \mid z_0^{(m)} \right)}_{\text{nonstationary}} + \underbrace{\mathbb{E}\mathrm{Var}\left( \hat{f}_N^{(m)} \mid z_0^{(m)} \right)}_{\text{persistent}}. \tag{13}$$

The nonstationary variance decreases with a long warmup, as the Markov chains forget their starting point. On the other hand, the persistent variance requires a long sampling phase—a condition which is incompatible with the many-short-chains regime. Ideally, we would want a direct measure of the nonstationary variance to diagnose convergence.

This can be achieved with a **nesting scheme**. Suppose we partition our Markov chains into groups of chains or *superchains*. The chains within a superchain are initialized at the same starting point and then run independently.

Adjusting our notation, we take $K$ to be the number of superchains and $M$ the number of subchain within each superchain, for a total of $KM$ chains. Then the *superchain Monte Carlo estimator* is,

$$\bar{f}_{NM}^{(k)} = \frac{1}{MN} \sum_{m,n} f\left(z^{(kmn)}\right). \tag{14}$$

We then compute the sample variance,

$$B_{\mathfrak{n}} = \frac{1}{K-1} \sum_{k=1}^{K} \left( \bar{f}_{MN}^{(k)} - \bar{f}_{KMN}^{(\cdot)} \right)^2 . \tag{15}$$

As $K \to \infty$, $B_{\mathfrak{n}}$ converges to,

$$\begin{aligned} \mathrm{Var}\bar{f}_{MN}^{(k)} &= \mathrm{Var}\mathbb{E}\left( \hat{f}_{MN}^{(k)} \mid z_0^{(k)} \right) + \mathbb{E}\mathrm{Var}\left( \hat{f}_{MN}^{(k)} \mid z_0^{(k)} \right) \\ &= \mathrm{Var}\mathbb{E}\left( \hat{f}_{N}^{(m)} \mid z_0^{(k)} \right) + \frac{1}{M}\mathbb{E}\mathrm{Var}\left( \hat{f}_{N}^{(m)} \mid z_0^{(k)} \right), \end{aligned} \tag{16}$$

where in the second line, I used the conditional independence of the Markov chains within a superchain to show that (i) the super chain has the same expectation value as each individual subchain and (ii) the superchain's variance is reduced by a factor of $1/M$.

Our nesting strategy allows us to reduce the persistent variance, while leaving the nonstationary variance intact. With additional work, the contribution of the persistent variance can be exactly corrected for and a convergence diagnostic can be devised based on how quickly the nonstationary variance decreases.

**Remark 2.** *The behavior of the nonstationary variance reflects that of the squared bias, which also does not change with the number of chains. Additional arguments can be made to show that, under certain conditions, the nonstationary variance decreases at a "comparable" rate as the squared bias. (This is ongoing research!)*

**Remark 3.** *With some additional adjustments, it is possible to compute $\widehat{R}_{\mathfrak{n}}$ for chains of length 1. This opens the possibility of "continuously" monitoring the nonstationary variance at each iteration and developing an automatic stopping rule.*

### 5.2.3 GPU-friendly MCMC

To leverage the parallelization capacities of GPUs, we need SIMD instructions. The first step is to ensure that evaluation of the log density $\log \pi(z)$ and its gradient $\nabla \log \pi(z)$ can be done with SIMD instructions (Section 5.1). Then, at each "step", the GPU returns an array of gradients (one for each Markov chain).

Unfortunately, sophisticated adaptation strategy may not be compatible with SIMD.

**NUTS is not GPU-friendly.** NUTS and its variants are not GPU-friendly. Indeed, at each iteration, we need to construct orbits until the No-U-Turn termination criterion is met. Depending on the current "location" $z_0$ of a Markov chain and the sampled momentum $\rho_0$, the number of leapfrog steps to compute a valid orbit will vary between chains.

For example, if we consider a pair of Markov chains, one which takes one leapfrog step and another one two, the SIMD execution model runs both chains for two leapfrog steps and discards

the second leapfrog from the second chain. Moreover, the number of leapfrog steps is always driven by the largest orbit, yielding wasteful computation for most chains.

**ChESS-HMC: a GPU friendly alternative.** The SIMD constraint motivates new adaptive algorithms, such as ChEES-HMC [Hoffman et al., 2021], which ensures that each iteration of the Markov chain can be run in lock-step. This is simply done by fixing the trajectory length $L$.

Which $L$ should we choose then? Rather than achieving the No-U-Turn criterion for each chain at each iteration, we instead try to maximize the *expected squared jump distance* (ESJD),

$$\text{ESJD}_f = \mathbb{E}\left[||f(z_T) - f(z_0)||^2\right]. \tag{17}$$

If we take $f$ to be the identity, we obtain the expected squared distance from the initial point $z_0$ to the final point $z_T$. In other words, while we cannot hope to maximize the travelled distance for each chain at each iteration, we can try to maximize that distance on average.

Other choices of $f$ can be considered for the ESJD. For example, ChEES-HMC focuses on the second moment of $z$, arguing that it is a more difficult quantity to estimate than the first moment.

**Remark 4.** *When the target distribution has varying scales across dimensions (or somewhat equivalently, if we're not able to construct a good mass matrix $M$), we can improve the performance of HMC by jittering the trajectory length, i.e. sampling the number of steps size uniformly from $\{1, 2, \cdots, L\}$.*

**Opportunities from running many chains.** There are a number of MCMC strategies that benefit from running many chains. For example, the mass matrix and step size adaptation can pool information across many chains, meaning the adaptation per iteration is much faster. In some cases, this can lead to faster bias decay.

Certain methods use Markov chains targeting different distributions to sample from a challenging distribution. An example of this is tempering schemes.

## References

M. Hoffman, A. Radul, and P. Sountsov. An adaptive-mcmc scheme for setting trajectory lengths in hamiltonian monte carlo. In *International Conference on Artificial Intelligence and Statistics*, volume 130, pages 3907–3915, 2021.