MCMC550 - A parallelized MCMC on GPGPU

Alper Bozkurt, Caspar Oesterheld, Harsh Parikh

Abstract

Motivation

- Computing an *explicit posterior* based on data is often *infeasible*.
- *Markov chain Monte Carlo* (MCMC) is a common approach in the ML and statistics communities to *sample* from the posterior.
- MCMC is a very *computationally intensive* procedure.
  - Cost per Monte Carlo simulation step of the Markov chain is linear in the data size.
- Neiswanger et al. (2014) propose an algorithm that first *splits* the data and then *combines* the samples from separate MCMCs.

Contribution

- Adapting their approach to *GPGPU* (General-Purpose computing on Graphics Processing Units) platforms.
- Analyze the correctness and limitations of our approach on different applications of MCMC.

Benchmarking

- We benchmark our performance by comparing our implementational approach with existing MCMC implementations in R by Geyer (2019) and PyMC3 in Python.
- We observe a significant speedup compared to existing MCMC implementations while recovering distributions.

Objectives

- With Neiswanger et al.'s scheme, MCMC becomes especially suitable for implementation on a GPU (see "Methods" section).
- We therefore implemented Neiswanger et al.'s algorithm on a GPGPU using CUDA.
  - In particular, we distributed the batches of data to different thread blocks.
  - Each thread block runs the Metropolis-Hastings algorithm. Each individual thread separately calculates the (log) probability of a data point given a new proposed model.
  - Finally, we combine the samples generated by the different thread blocks via the "quick and dirty" method of Neiswanger et al. (Section 3.1).
- To see whether it can indeed be used to speed up MCMC, we compared the runtime of our implementation with the runtimes of existing CPU-based MCMC toolboxes.

Methods

1. **For *each batch* of sampled data**
   a. Sample using a *Gaussian step*
   b. Calculate new *log probability*
      i. Parallel reduction using sequential addressing
   c. *Accept or Reject* the update
      i. $P(accept) = \min(1, \frac{p_{new}}{p_{old}})$
   d. Repeat
2. **Merge**
   a. $\Sigma = \left(\sum_m \Sigma_m^{-1}\right)^{-1}$

$$\mu = \Sigma \cdot \left(\sum_m \Sigma_m^{-1} \cdot \mu_m\right)$$

Results

Simulation Setup

Normally distributed outcome Y

$$Y \sim \mathcal{N}(\mu, \sigma^2)$$

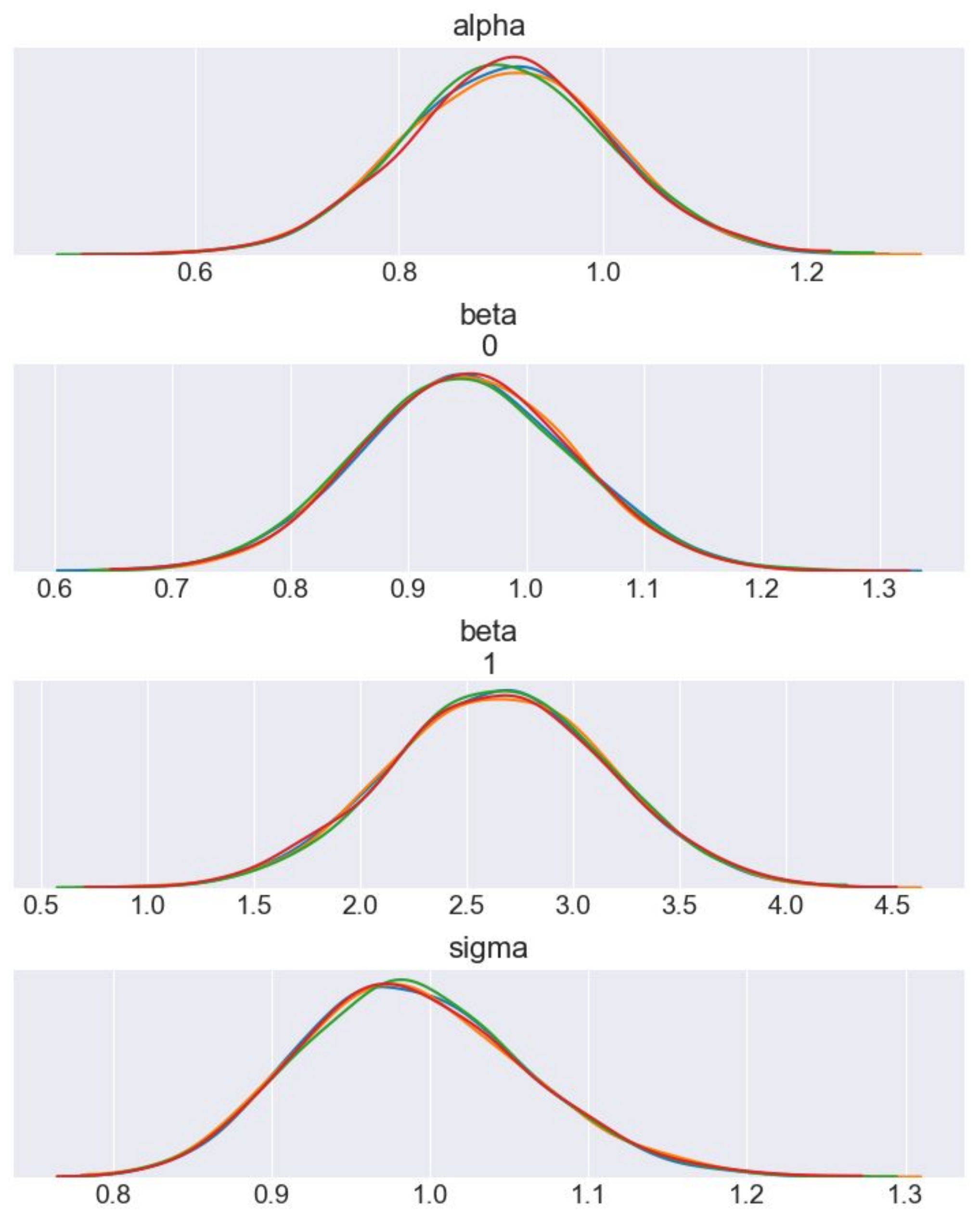
Expected value of outcome is a linear function

$$\mu = \alpha + \beta_0 X_0 + \beta_1 X_1$$

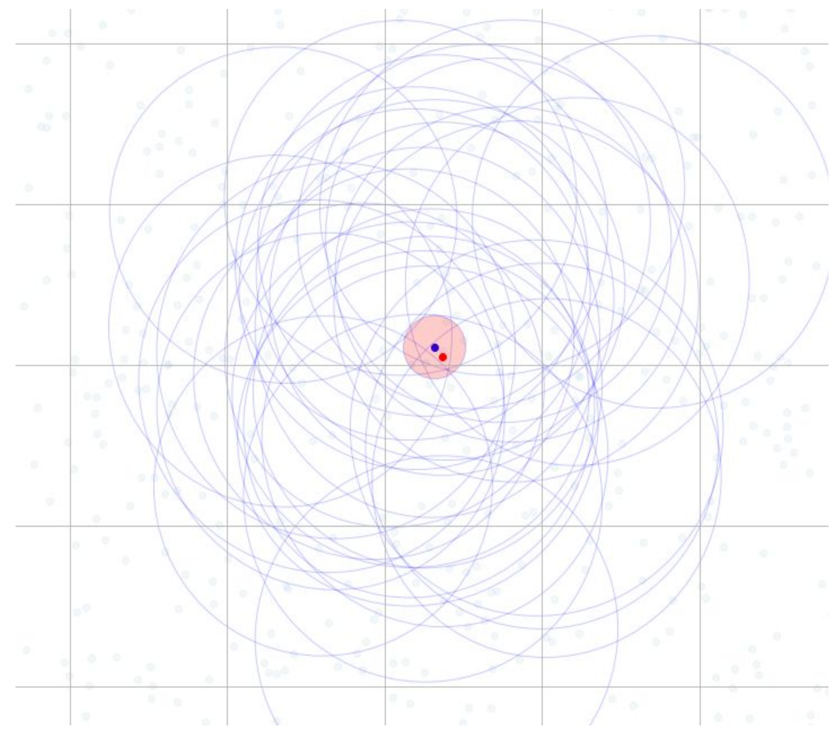
Parameters drawn from normal-esque distributions

$$\alpha \sim \mathcal{N}(0, 100)$$
$$\beta_i \sim \mathcal{N}(0, 100)$$
$$\sigma \sim |\mathcal{N}(0, 1)|$$

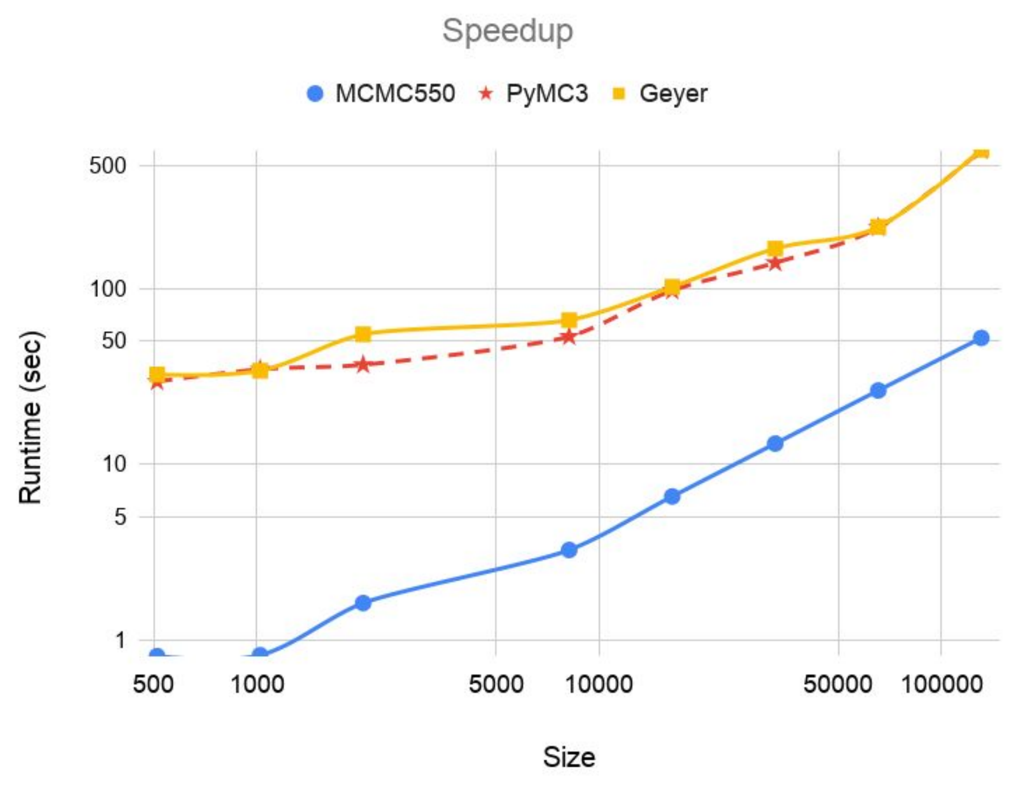
Marginal Posterior Distributions



Merging Batches



Time & Resources



Discussion

- **Implementation**: Our course-project work shows the promise of implementing parallel MCMC on a GPU.
- **Benchmarking:** Outperforms recent CPU-based implementations by 1-2 orders of magnitude.

Limitations and possible future directions

- We have used only a specific, relatively simple and synthetic example of a data distribution.
- We only considered runtime for a fixed number of MCMC steps and not in detail the costs in terms of accuracy of Neiswanger et al.'s algorithm.

References

Chib, S. and E. Greenberg (1995): "Understanding the Metropolis-Hastings Algorithm." *The American Statistician* 49(4), pp. 327–335.

Geyer, C.J. (2019). Mcmc package example (version 0.9.6). https://cran.r-project.org/web/packages/mcmc/vignettes/demo.pdf

Harris, M. (n.d.): Optimizing Parallel Reduction in CUDA. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Neiswanger, W., C. Wang, and E.P. Xing (2014): "Asymptotically exact, embarrassingly parallel MCMC". *UAI'14 Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence* (eds. N. Zhang and J. Tian), pp. 623–632.