# MCMC-550

Alper Kamil Bozkurt, Caspar Oesterheld, Harsh Parikh

November 2019

## 1 Introduction

Monte carlo markov chain or MCMC is a common algorithm in statistics and machine learning community where computing the integration over untractible probability distribution functions is common. For example, imagine we have data $x_1, \ldots, x_n$ and we want to draw conclusions about the hypothesis $\theta$ from which the data is sampled in reality. In the Bayesian paradigm, we have some prior $p(\cdot)$ over models and each model $\theta$ specifies a prior $p(\cdot \mid \theta)$ over data points, and want to calculate the (argmax of the) posterior

$$\begin{aligned} p(\theta \mid x_1, ..., x_n) &= \frac{p(x_1 \mid \theta)...p(x_n \mid \theta) p(\theta)}{p(x_1)...p(x_n)} && (1) \\ &= \frac{p(x_1 \mid \theta)...p(x_n \mid \theta) p(\theta)}{\int p(x_1 \mid \theta')...p(x_n \mid \theta') p(\theta') d\theta'}. && (2) \end{aligned}$$

As one might suspect, this is usually (depending on the given distributions) computationally infeasible because computing the denominator is usually infeasible. Instead, one generally settles for *sampling* from the posterior.

Markov chain Monte Carlo (MCMC) algorithms have been developed to solve problems like this one. Roughly, they construct a Markov chain whose states are what are the models $\theta$ in the above setting and whose equilibrium distribution of states/models is $p(\theta \mid x_1, ..., x_n)$. To sample from $p(\theta \mid x_1, ..., x_n)$ one can then sample one or more trajectories from the Markov chain and randomly pick a state from these trajectories. Since $p(\theta \mid x_1, ..., x_n)$ is only the equilibrium distribution of the Markov chain, one has to simulate the chain until the equilibrium distribution is reached (though in general it is hard to say when this is the case) and one has to pick a late state from the simulation. One well-known example of such a method is the Metropolis-Hastings algorithm.

Unfortunately, MCMC is a very computationally intensive procedure. One particular problem is that the cost per time step of the Monte Carlo simulation of the Markov chain is generally linear in the size of the data set. Hence, when the data set is very large, MCMC might need a long time to reach the desired equilibrium distribution.

To deal with this issue, Neiswanger et al. [2] propose a method for parallelizing MCMC. The proposal is to split the data points $x_1, \ldots, x_n$ into $M$ distinct data sets $N_1, \ldots, N_M$, and then for each of these sets run MCMC independently. MCMC is therefore parallelized into $M$ different MCMCs, each of which takes only $1/M$ (if the data are spread equally) of the computation time. Finally, they provide a method for aggregating the resulting samples of the "subposteriors" $p(\cdot \mid N_1), \ldots, p(\cdot \mid N_M)$ into samples from the posterior given all the data.

*Contribution.* Our project aims at adapting the work by [2] to GPGPU (General-Purpose computing on Graphics Processing Units) platforms to exploit even further parallelism. We analyze the correctness and limitations of our approach on different application of MCMC. We benchmark our performance by comparing our implementational approach with existing MCMC implementations in R by Geyer [1] and PyMC in python.

# 2 Implementation

We use CUDA as the platform for implementing the approach. We divide a standard MCMC algorithm into two levels according to the thread hierarchy in CUDA. We first split the given data set into $M$ subsets and launch $M$ thread blocks each executing an independent MCMC on one of these subsets. In this way, the thread blocks do not need to communicate each other thereby avoiding expensive synchronization operations. However, threads within a block are able to access a fast shared memory and to be easily synchronized by a CUDA primitive; therefore, they can efficiently compute the likelihood of the subset given a parameter in each iteration of the MCMC. The combination part is also inherently parallelizable. Each thread can independently draw a sample from the true distribution using the samples obtained for each subset.

We provide the python code of our implemention in the appendix.

# 3 Results and Timeline

## 3.1 Timeline

TABLE 1 Timeline

| | |
|---|---|
| October 25 | Acquaint with CUDA; Acquaint with MCMC; Workout MCMC-550 pseudo-code. |
| November 01 | Implement MCMC-550 (beta version) without aggregation. |
| November 03 | Submit Progress Report. Implement aggregation. |
| November 05 | Find and install existing implementation. |
| November 07 | Compare against existing implementation. |
| November 12 | Apply MCMC550 on multiple different problems. |
| November 16 | Have results and preliminary analysis. |
| November 28 | Complete Project Report/Presentation. |
| December 5 | Buffer time for spillovers from past. |

## 3.2 Correctness

In this section, we discuss the experiments concerning the correctness of MCMC550's implementation. The main goal of MCMC is to recover the true underlying distribution. We will be dividing this section into two parts. In the first part, we will show experiments with standard and simple data generative proces (DGP) such as normal or exponential distributions. In the second part, we generate the data using complex distribution for which calculating a closed form integral of probability distribution function might be infeasible. Such cases are important while testing MCMC algorithm as these cases will be where MCMC might be used.

For first set of simple DGP experiments, we draw $\theta$ from a multinormal distribution and the data $\mathscr{X}$ is drawn i.i.d from a multinormal distribution with $\theta$ as the mean.

$$\theta \sim \mathcal{N}(\mu_\theta, \Sigma_\theta) \text{ WHERE } \mu_\theta = [1,1], \Sigma_\theta = [[1,0][0,1]]$$
$$\mathscr{X} \sim \mathcal{N}(\theta, \Sigma) \text{ WHERE } \Sigma = [[0.1,0][0,0.1]]$$

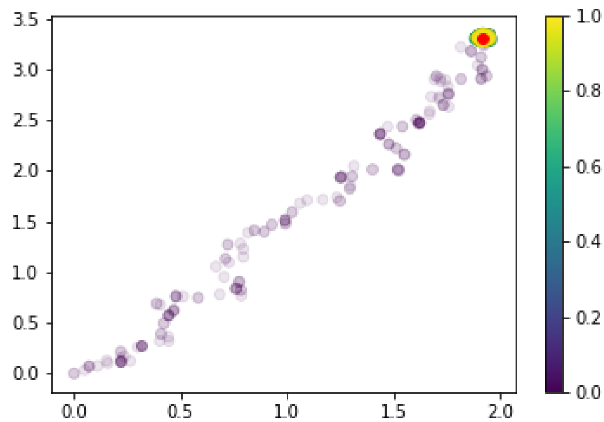


Figure 1: The plot showing that the MCMC550 algorithm converging to a true normal distribution after a brief burn-in iteration.

Figure 1 shows that after a burn-in period of approximately 50 iteration, we observe that the MCMC550 implementation essentially starts to sample around the observed $\theta$. This works as a first but preliminary proof of concept that MCMC550 is performing as expected. (As mentioned above, we will be trying more experiments to support our claim about the method)

### 3.3 Efficiency

In this section, we will be discussing the speedup that we achieve by using MCMC550 compared to the existing implemenations in popular programming languages like python, Matlab or R. We aim to work on this and get the result by the end of November as shown on the timeline chart.

## References


[1] Charles J Geyer. Mcmc package example (version 0.9. 6). 2012.

[2] Willie Neiswanger, Chong Wang, and Eric P. Xing. Asymptotically exact, embarrassingly parallel mcmc. In Nevin Zhang and Jin Tian, editors, *UAI'14 Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence*, pages 623–632. 2014.


## Appendix

```python
"""A Parallel Implementation of MCMC in CUDA"""

from numba import jit, cuda, float64
from numba.cuda.random import xoroshiro128p_uniform_float64, xoroshiro128p_normal_float64, init_xoroshiro128p_state, xoroshiro128p_jump, xoroshiro128p_dtype

import numpy as np
import math
import matplotlib.pyplot as plt

class pMCMC:
    """Host code of our parallel MCMC implementation.
    """
    def __init__(self, data, block_size, n_iter, seed=0):
        self.data = cuda.to_device(data)
        self.n_iter = n_iter

        # Parameters for the kernel launch
        self.block_size = block_size
        self.n_samples = data.shape[0]
        self.n_blocks = self.n_samples // block_size

        # Allocate an output array on the GPU
        self.output = cuda.device_array((n_iter,self.n_blocks,2))

        # Create random number generators for each thread
        # NOTE: The threads within the same block should generate the same random numbers
        rng_states = np.empty(self.n_samples, dtype=xoroshiro128p_dtype)
        for i in range(self.n_samples):
            init_xoroshiro128p_state(rng_states, i, seed)  # Init to a fixed state
            for j in range(i//block_size):  # Jump forward block_index*2^64 steps
                xoroshiro128p_jump(rng_states, i)
        self.rng_states = cuda.to_device(rng_states)  # Copy it to the GPU

    def launch(self):
        """Launches the kernel and returns the MCMC samples.
        """
        mcmc[ self.n_blocks, self.block_size ]( self.data, self.output, self.rng_states, self.n_iter)
        return self.output.copy_to_host()

    @staticmethod
    def generate_data(n_samples):
        """Generates and returns a hyperparameter theta and n_samples noisy observations of theta."""
        theta =  np.random.multivariate_normal([1,1],cov=[[1, 0],[0, 1]])
        data = np.random.multivariate_normal(theta,cov=[[0.1, 0],[0, 0.1]],size=n_samples)
        return theta, data



@cuda.jit
def mcmc(data, output, rng_states, n_iter):
    """Device code of our parallel MCMC implementation.
    """
    shared = cuda.shared.array(shape=(2**9,), dtype=float64)  # Shared Memory
    tx = cuda.threadIdx.x  # Thread ID
    ty = cuda.blockIdx.x  # Block ID
    bw = cuda.blockDim.x  # Block Size
    idx = bw*ty+tx  # Global ID

    theta = (0.,0.)  # Initialize theta
    x = data[idx]  # Fetch the data point
    logp_x = -(((theta[0]-x[0])**2)/(2*0.1) + ((theta[1]-x[1])**2)/(2*0.1))  # Log-likelihood of the data point
```

```python
    shared[tx] = logp_x  # Put the log-likelihood to the shared memory
    cuda.syncthreads()

    # Reduction using sequential addressing. NOTE: Increasing the data points per thread might increase the performance
    s = bw//2
    while s>0:
        if tx < s:
            shared[tx] += shared[tx+s]
        cuda.syncthreads()
        s>>=1
    # Get the log-likelihood of the sub-dataset from the first position
    logp = shared[0]  #  NOTE: Might cause some performance issues

    # Add the log-prior
    log_prior = -(((theta[0]-1)**2)/2 + ((theta[1]-1)**2)/2)
    logp += log_prior

    # Main MCMC Loop
    for i in range(n_iter):
        # Propose a new theta
        theta_ = (theta[0] + 0.1*xoroshiro128p_normal_float64(rng_states, idx), theta[1] + 0.1*xoroshiro128p_normal_float64(rng_states, idx))
        logp_x = -(((theta_[0]-x[0])**2)/(2*0.1) + ((theta_[1]-x[1])**2)/(2*0.1))  # Log-likelihood of the data point
        shared[tx] = logp_x  # Put the log-likelihood to the shared memory
        cuda.syncthreads()

        # Reduction using sequential addressing
        s = bw//2
        while s>0:
            if tx < s:
                shared[tx] += shared[tx+s]
            cuda.syncthreads()
            s>>=1
        # Get the log-likelihood
        logp_ = shared[0]

        # Add the log-prior
        log_prior = -(((theta_[0]-1)**2)/2 + ((theta_[1]-1)**2)/2)
        logp_ += log_prior

        # Acceptance ratio
        alpha = math.exp(min(0,logp_-logp))
        # Draw a uniform random number
        u = xoroshiro128p_uniform_float64(rng_states, idx)
        # Accept/Reject?
        if u < alpha:
            theta = theta_
            logp = logp_

        # Write the sample to the memory
        if tx == 0:
            output[i,bw] = theta
```

Listing 1: Implementation in Python