Accelerating Metropolis-Hastings Algorithm with CUDA-enabled GPU

Yanming Lai, Junyu Liang, Webster Bei Yijie

Abstract—

Markov Chain Monte Carlo (MCMC) is a technique very commonly used in Bayesian inference. Markov Chain Monte Carlo can be used to generate samples from a given distribution, and it can also be used as an optimization algorithm that finds a good configuration (or a good value for the parameter to be optimized) for a given value function. Independent Metropolis-Hastings algorithm is one type of MCMC algorithm that performs sampling from an unnormalized probability distribution. In this paper, we seek to explore the possibility of accelerating the sampling process on a CUDA-enabled GPU.

MCMC, Metropolis-Hastings, Sampling, CUDA

1 Introduction

Metropolis-Hastings is one of the most widely used Markov Chain Monte Carlo algorithms that provide a framework for performing several different tasks. To list some, two common applications of the Metropolis-Hastings algorithm are Bayesian parameter estimation and data sampling according to an unnormalized distribution function. After diving deeper into exactly how those two variants of Metropolis-Hastings algorithm work, we have identified several areas of potentially parallelizable computations that we could exploit to speed up the computation.

Specifically, we found that Metropolis-Hastings for parameter estimation is far less parallelizable since the goal of the algorithm is to generate a sequentially better value (closer to the true value) for the parameter of interest. This goal makes sure that the individual value proposal steps have to be done in linear order as opposed to done simultaneously. The only place where parallelism exists is the step where the likelihood of generating a data sample under the currently proposed parameter is computed for each input data point. The other application of Metropolis-Hastings that is purposed to generate data samples from an unnormalized distribution function, however, is perfectly parallelizable. Therefore, our work will mainly focus on this variant of the Metropolis-Hastings algorithm.

2 Background

In recent years, statisticians have been increasingly drawn to Markov chain Monte Carlo (MCMC) methods to simulate complex non-standard multivariate distributions. Sampling it-

self is of great interest not mention the potential applications that could be enabled if we can efficiently sample from an arbitrary distribution. To name a few, we could perform numerical integration or Bayes inference through sampling.

MCMC methods comprise a class of algorithms to sample from a probability distribution from which direct sampling is difficult. MCMC algorithms generates a random sequence X_i that is a Markov chain, meaning that given X_t, subsequent observations X_{t+n} are conditionally independent of previous observations X_{t-k}, for any positive integers k and n. Any two observations are not independent, but the more separate the two observations are in the sequence, the more independent they are. The last important characteristic of MCMC methods are that the resulting sequence converges to the target distribution.

One type of MCMC methods that has attracted a considerable amount of attention is the Metropolis-Hasting (M-H) algorithm. It was developed by Metropolis, Rosenbluth, Teller (1953) and subsequently generalized by Hastings (1970). The algorithm is extremely versatile, and thus can be applied in many fields, such as economics, statistics, and physics.

2.1 Independent Metropolis-Hastings Algorithm

Suppose we are given a probability distribution $P(x)$ from which we wish to obtain data points x that are distributed according to the distribution. One could perform naive rejection sampling if the given distribution is normalized (i.e. integration over the support is 1). However, in a more commonly encountered situation, the probability distribution will not be normalized. Moreover, it is often difficult to perform normalization since normalization requires integration which is also hard to achieve for wild and high-dimension functions. Metropolis-Hastings algorithm solves this problem in the following way:[9]

- Suppose the given distribution function satisfies the detailed balance condition, that is $P(x'|x)P(x) = P(x|x')P(x')$. Then we have the following relation
$$\frac{P(x'|x)}{P(x|x')} = \frac{P(x')}{P(x)}$$
- We decompose the function $P(x'|x)$ through $P(x'|x) = g(x'|x) \times A(x', x)$
- This implies that the relation
$$\frac{A(x', x)}{A(x, x')} = \frac{P(x')}{P(x)} \frac{g(x|x')}{g(x'|x)}$$
must be satisfied. This will be the constraint equation that governs the transition from one state to the next.

Function $g(\cdot|\cdot)$ proposes a new state given an old state, and it represents the probability of transition from one state to the other. It is a common practice to choose a symmetric g such that $g(x|x') = g(x'|x)$. Function $A(\cdot, \cdot)$ represents the probability of accepting the new state given an old state. With these assumptions, we see that the general flow of Metropolis-Hastings algorithm is to first propose an new state given the old one, and then decide whether to accept or reject the new state based on whether or not the new state is more probable then the old state (i.e. if the new state is more probable, it will be taken definitely, otherwise, it will have some chance to be taken).

For the purpose of our study, certain algorithmic decisions are made when our version of Metropolis-Hastings sampling algorithm is implemented. The general flow of our algorithm is:

- **Initialization:** produce a randomly generated vector x of length n, each entry of the vector is generated from a unit Gaussian distribution

- **State Proposition:** mutate the vector x to a new vector x', this is achieved by perturbing values in x around by an amount randomly sampled from a unit Gaussian distribution

- **Acceptance Ratio Calculation:** Compute $p_{acc} = \frac{P(x')}{P(x)}$, this is done by passing the sample parameters to a given distribution function

- **State Update:** generate a number uniformly in [0,1], if it is greater than p_{acc}, keep x, otherwise set $x = x'$

Repeat the State Proposition to the State Update steps for sufficient number of times and the collection of intermediate states x at each step will be the sample points.

Notice that we have chosen unit Gaussian distribution as our proposal distribution (i.e. $g(\cdot)$ in the derivation formulation). This distribution is symmetric and can therefore simplify our calculation for acceptance ratio. The user-provided function $P(\cdot)$ has virtually no restriction other than that it should take in a vector of parameters of arbitrary length and output a single number that represent certain notion of probability or likelihood. In the methodology section, we will analyze in more depth why and how parallelism exists in this algorithm and why and how a parallel Metropolis-Hastings sampling algorithm could enable faster convergence.

2.2 Nvidia Graphics Processing Unit Architecture and CUDA Programming Paradigm

Nvidia's graphic processing unit (GPUs) are very computational intense and highly parallel. GPU typically has hundreds of processor cores and thousands of threads concurrently running on these cores. CUDA(Compute Unified Device Architecture) is introduced by Nvidia in 2007 to be used to develop software for graphics processors and those GPU application with highly parallelizable property. CUDA is a parallel computing platform and programming model developed by Nvidia. It is used to accelerate compute-intensive application by exploiting the graphics processing units (GPUs) for parallelizable part of the computation.

2.2.1 System Model

CUDA uses language which is similar to C language and include extensions to that language tio use the GPU specific features that uses new API calls and some new type qualifiers that apply to functions and variables. CUDA has kernels which can be a function or a program which invoked by the CPU. It is executed N number of times in parallel on GPU using N number of threads. The device is consists of a set of multiprocessors which are divided into several cores and each core executes instructions from one thread at a time showed in Figure1.

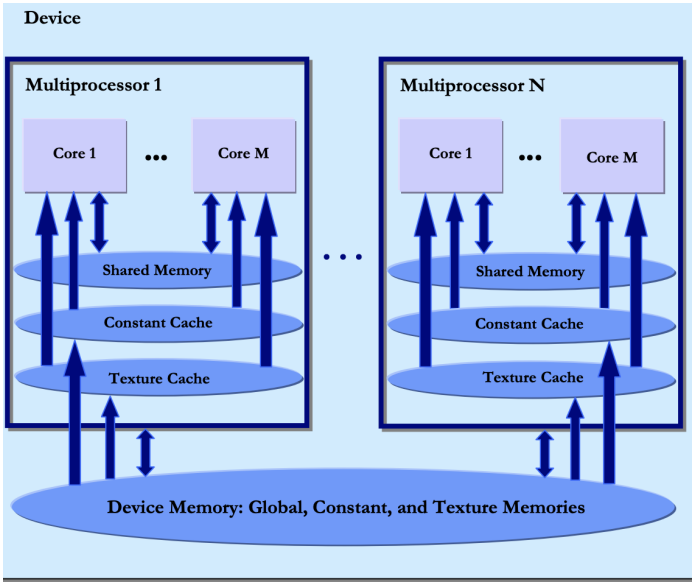

Figure 1: The CUDA Memory Model

2.2.2 Heterogeneous Programming

CUDA programming paradigm is the combination of both serial and parallel execution. Figure 2 shows a example heterogeneous programming.

The C codes run serially called host. The GPU is called device.

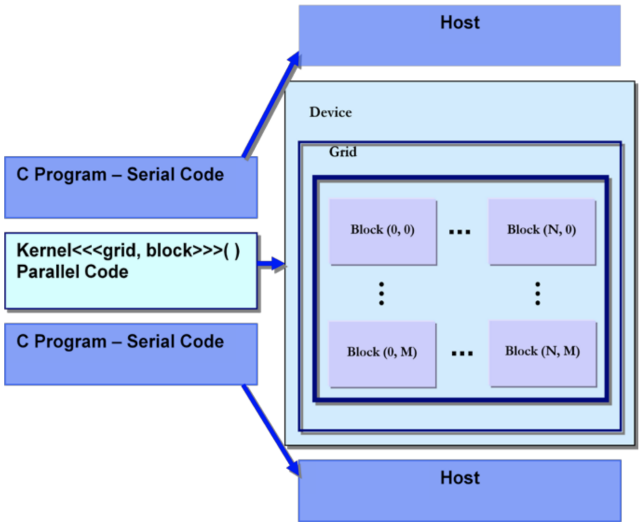

Figure 2: Heterogeneous Programming Architecture

2.2.3 CUDA Grid and Block

CUDA grid is consists of one, two or three dimensional thread blocks. Blocks are further divided into threads. All threads in one block can communicate with each other by using shared memory and execute within one same multiprocessor explained in Figure 3. The CUDA paradigm offers some variables to manipulate these structure. Use *blockIdx*(ranging from 0 to gridDIm-1) to query the id of a thread block and use *blockDim* variable to get the dimension of a grid. Thread ID can be accessed by variable *threadIdx*.

2.2.4 Memory Control Flow

The control flow within CUDA must ensure that memory be allocated on device before the kernel function waking up. If the kernel function uses the same piece of data then the data must be copied from host memory space to device memory space illustrated in Figure4. CUDA API such as *cudaMalloc()* facilitates the allocation and de-allocation of device memory during run time. On the other hand, to get the result from

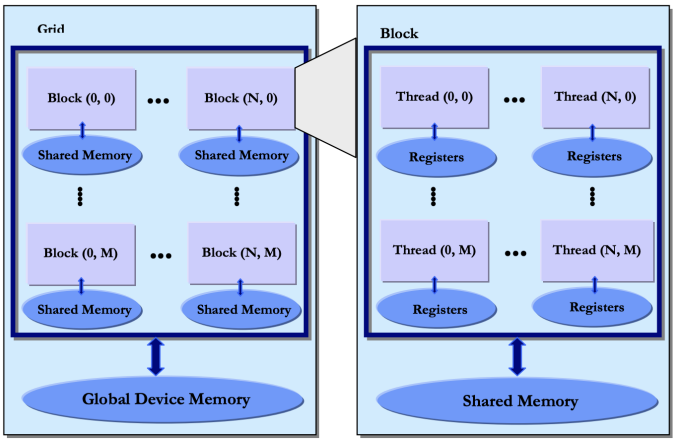

Figure 3: CUDA Grid and Block Structure

execution kernel function data in device memory must be copied back to host memory. API for example *cudaMemCpyToSymbol()*, *cudaMemCpyFromSymbol()* and *cudaMemCpy()* can be used to during this process.

The overall control flow can be summarized as following:

- Allocate both host and device memory space. Ensure device memory is R/W to host
- Copy data from host to device if needed using CUDA build-in API
- Execute kernel function among cores in parallel
- Copy result from device memory back to host

Nvidia Tesla K80 GPU from Nvidia Pascal microarchitecture is used in the experiment. K80 has four major components: an array of Graphic Processing Cluster(GPCs), Texture Processing Clusters(TPCs), Streaming Multiprocessors(SMs) and memory controllers. To be more specific, K80 has six GPCs, 60 Pascal SMs(each Streaming Multiprocessor has 64 CUDA Cores and four texture units), 30 TPCs(each of them consist of two SMs) and eight 512 bit memory controllers. CUDA programming model utilize the feature of

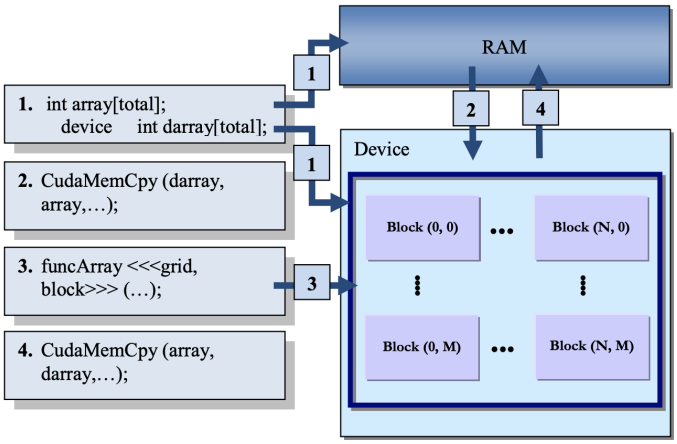

Figure 4: Control Flow Example

Unified Memory to ensure that the hardware computational units access to data directly and fast. Unified Memory lowers the bar of entry to parallel programming on GPUs by making explicit device memory management an optimization rather than a requirement.

3 Methodology

3.1 General Idea

The key idea to improve sampling throughput is to allow multiple threads to perform independent works simultaneously. As we have seen, each CUDA-enabled Nvidia graphics cards have multiple Streaming Multiprocessors(SM), and each Streaming Multiprocessor has multiple CUDA cores. With its many CUDA cores, multiple threads can be run simultaneously within each Streaming Multiprocessor. Take a graphics card with compute capability 3.5 as an example, each Streaming Multiprocessor is able to support up to 1024 threads to run simultaneously [10]. Therefore, it makes sense that we map the Metropolis-Hastings algorithm routine to the threads and allow parallelism be exploited by multiple threads.

As mentioned earlier, in Metropolis-Hastings algorithm multiple different mutations can be proposed from a single current state and whether such mutation is taken will be checked independent of the history of states. As such, we immediately see two different ways of parallelizing the algorithm using multiple threads. The first way is to pipeline the state generation step into stages shown in the algorithm flow earlier. Then

- **State Proposition**: each thread picks a state from the *Current States Pool* uniformly and randomly. Propose a new state by mutating the picked state, and throw the (picked state, proposed state) pair into *Proposal States Pool*
- **Acceptance Ratio Calculation**: each thread picks a state pair from the *Proposal States Pool*. Based on the state pair, compute the acceptance ratio and decide whether the output state should be the proposed state or the old state in the pair. Throw the output state into the *Current States Pool* (randomly and uniformly replace any of the states in the pool)

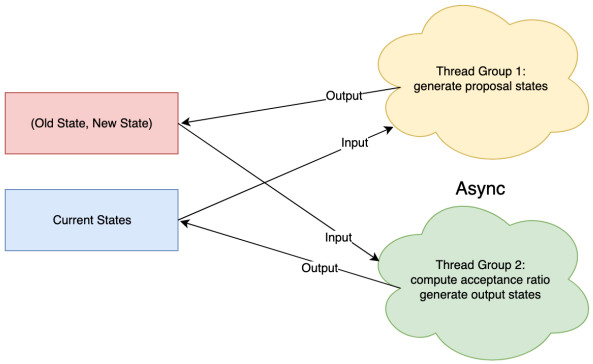

Figure 5: Visual Representation of First Idea

The two stages can be asynchronously and simultaneously performed by two groups of threads so long as they share the same memory space that contains the *Current States Pool* and *Proposal States Pool*. This simple pipeline allows the states to be processed in batches and in streams.

A little thought, however, shows that another approach can be taken. We notice that spawning multiple mutations at each stage would in

effect be equivalent to starting multiple independent Metropolis-Hastings routines from different starting points. Therefore, instead of having two groups of threads performing different stages of the Metropolis-Hastings algorithm asynchronously, we could have multiple threads performing all stages of the algorithm simultaneously so long as they are completely independent of each other. Therefore, the other implementation that we eventually went for is to map Metropolis-Hastings routines to independent threads.

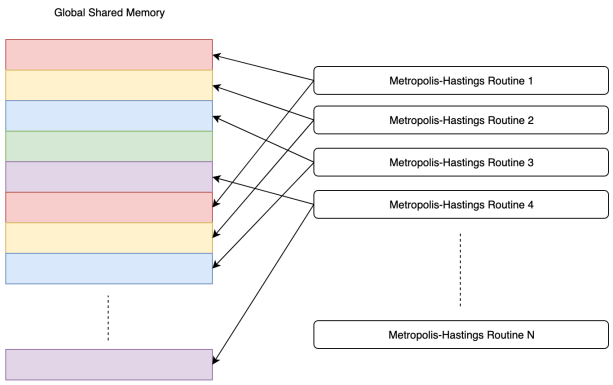

Figure 6: Visual Representation of Second Idea

3.2 Implementation Details

Both the CPU and CUDA versions of Metropolis-Hastings are implemented. The CPU version is written in C++ and the GPU version is implementation in CUDA C++ language. There are several important features in the implementations that need to be mentioned here.

3.2.1 Customizable Distribution Function

Both the CPU and CUDA implementations allow custom distribution function according to which samples will be taken. The provided distribution function can be of arbitrarily high dimensional (experiments show that 300 dimensions is about the practical limit for the CUDA implementation). This is so that the scalability of our implementations can be compared when sampling from a very high dimensional sample space is needed. In practical applications, the sampling space is often high dimensional.

3.2.2 Single Precision Float

Since there are less double precision arithmetic units than single precision units in a typical Nvidia graphics card, we default to using single precision float in our implementation so that arithmetic unit contention can be minimized to allow greater throughput.

3.2.3 Own Implementation of Proposal Distribution

State mutation and proposal requires the ability to sample from a known distribution function that is symmetric. In our case, we have chosen unit Gaussian distribution. Since the unit Gaussian distribution sampling implementations provided by CUDA library and GNU may be differently optimized, we resort to implementing our own sampling function for a fair comparison.

Specifically, we implemented a sampling function based on the Box-Muller transform[8], which allows a pair of randomly generated numbers from the uniform distribution to be turned into two numbers as if they were sampled from a Gaussian distribution.

3.2.4 Timing and Profiling

Run times are produced by the running instances themselves through the use of *clock()* function. For both CUDA and CPU implementations, memory allocations time is defined to be the time taken to perform *malloc()* or *cudaMallocManaged()*. For the CPU implementation, computation time is defined to be the time taken to generate specified number of data samples before writing out. Similarly for the CUDA implementation, computation time is clocked when *cudaDeviceSynchronize()* returns.

3.3 Experiment Platform

Experiments are done on Duke Compute Cluster with the following hardware specification.

CPU: Intel(R) Xeon(R) CPU E5-2640 v3
Clock: 2700MHz
Graphics Card: Nvidia Tesla K80
Clock: 824MHz(Max)
CUDA cores: 2880
Compute Capability: 3.7
Maximum number of threads per block: 1024

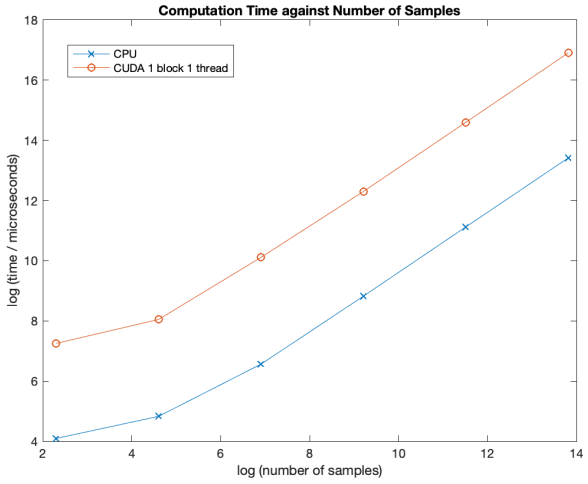

Figure 7: Computation time comparison for CPU and GPU of one thread and one block

4 Results

4.1 Performance Compared to CPU Implementation

To measure the effectiveness of our implementation, we compared the performance of Metropolis-Hastings sampling algorithm on CPU and GPU, using "number of data points generated per second" as the metric. The time taken to perform the actual sampling process is significantly faster on CPU given that only one thread and one block is used on GPU.

As shown in Figure 7, the time taken to perform the actual sampling process is significantly faster on CPU given that only one thread and one block is used on GPU. That means without any parallelism, GPU performs worse than CPU. This is expected since the clock of a CUDA core is significantly lower than that of a CPU core that we benchmarked with. However, as shown in 8, computation time of a multi-threaded GPU implementation is exponentially faster than that of the CPU implementation. In particular, we used an 16-block 256-thread GPU implementation (optimal configuration found through grid search). This shows that by parallelizing the Metropolis-Hasting algorithm, we have achieved significant acceleration.

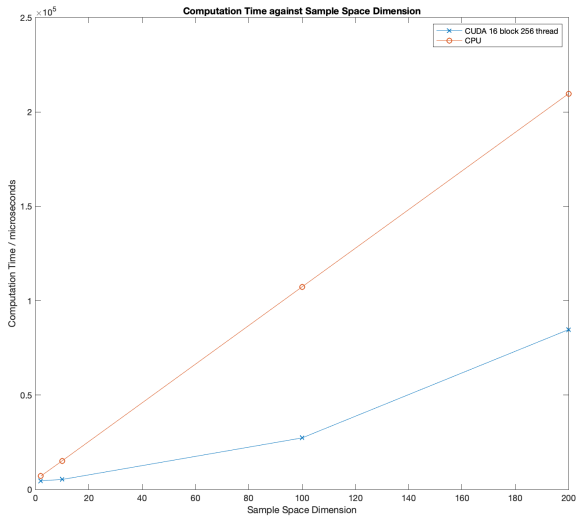

Figure 8: Computation time comparison for CPU and GPU with 16 blocks 256 threads

4.2 Influence of CUDA Configuration Parameters

We then investigated some CUDA-enabled GPUs parameters that may influence the performance of the algorithm with GPU. The factors we looked at include number of GPU blocks and the number of threads. These two parameters are the two fundamental configurations when configuring a CUDA implementation. As shown in 9 and 10, neither increasing the number of blocks or increasing the

number of threads alone achieve significant performance improvements beyond certain point. This is due to the bottleneck in global shared memory access time, when either setting is too large.

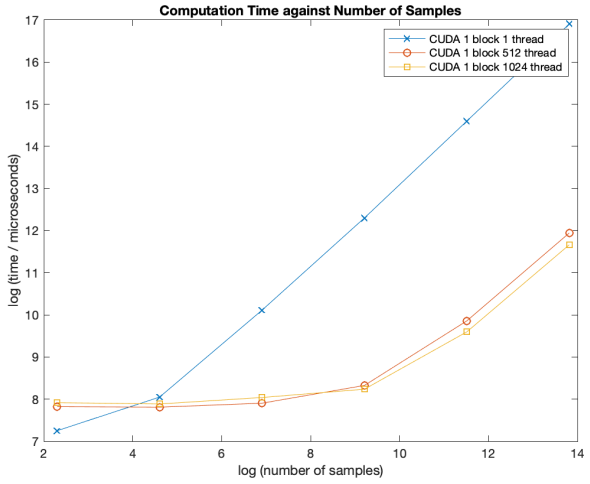

Figure 9: Computation time comparison for GPU with 1,512,1024 threads

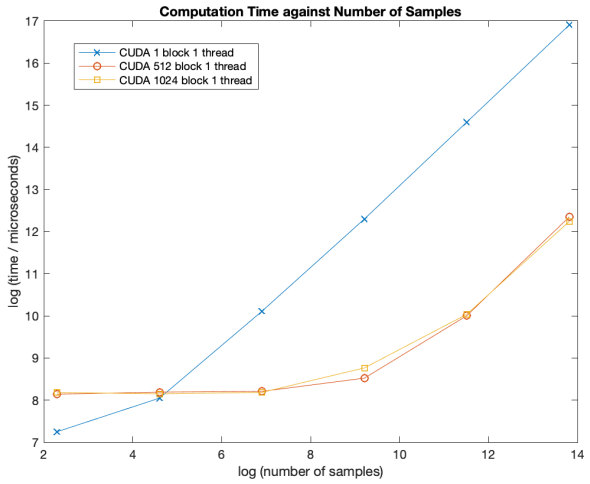

Figure 10: Computation time comparison for GPU with 1,512,1024 blocks

To find the optimal configuration of number of blocks and thread, we did a grid search with number of blocks, number of threads, and computation time. The results are displayed in 11 It is found that the lowest computation time is achieved at the configuration of 16 threads and 256 threads.

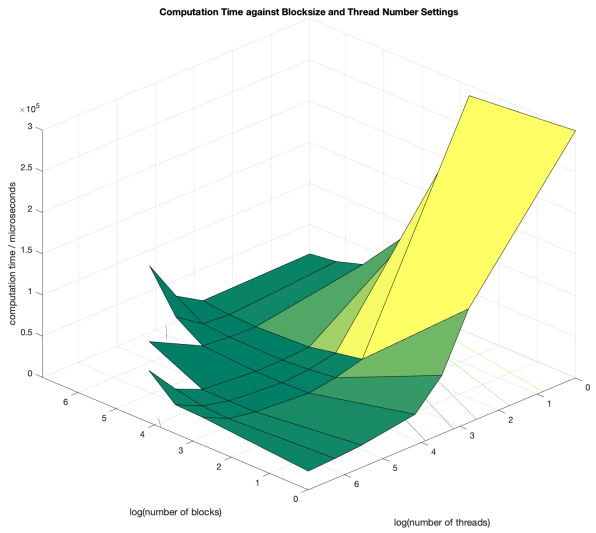

Figure 11: Grid search for optimal number of block and threads configuration

4.3 Overhead of Shared Memory Allocation In CUDA

In the GPU implementation, shared memory allocation incurs significant overhead. As shown in 12, the memory allocation time for CUDA-enabled GPU is orders of magnitude larger than that of the CPU implementation. This exposes the overhead due to the shared memory ac cross multiple blocks and threads in the GPU. The influence of the overhead also deteriorates as the number of samples increases.

5 Related Works

Historically, MCMC methods, including the Metropolis-Hasting method, is limited by its inherently sequential nature. Despite regular increases in available computing power, Markov chain Monte Carlo (MCMC) algorithms can still be computationally very expensive; many thousands of iterations may be necessary to obtain low-variance estimates of the required quantities with an oftentimes complex statistical model being evaluated for each set of proposed parameters [7].

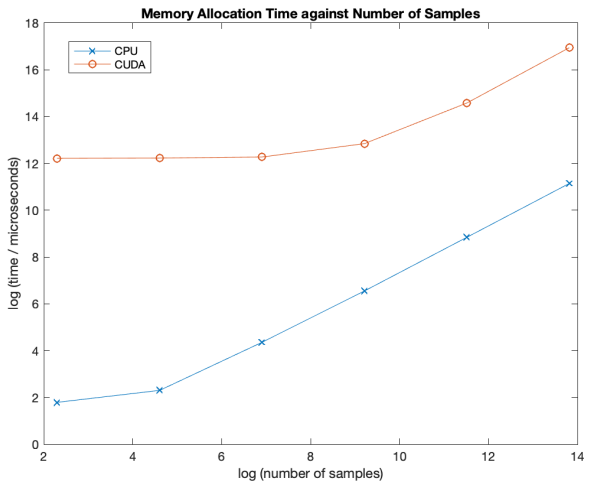

Figure 12: Memory Allocation Time Against Number of Samples for CPU and GPU

In the last 5 years we have witnessed a significant increase in the number of statistical and econometric applications leveraging the availability of powerful and effective Graphical Processor Units (GPU). There are a number of approaches for using GPUs for accelerating computation in Bayesian inference—some approaches are completely model specific, others are generic.

Most of the work on accelerating MCMC methods on GPUs focuses on parallelizing the likelihood computations [4]. Previous studies explored different techniques to enhance some degree of parallelism in the Metropolis-Hastings algorithm. The most basic scheme for applying parallelism in a generic Metropolis-Hastings algorithm is to run a number of MCMC algorithms independently in parallel and merge the results (Mykland, Tierney, and Yu 1995 [1]).

Building on the basic scheme, the group Craiu, Rosenthal, and Yang (2009)[2] built an adaptive MCMC algorithm, which takes the product of the target densities over the chain as the overall target. Studies as such showed the benefits of running MCMC algorithms in parallel. Another paper by Jacob et. al. [3]. proposed "block Independent Metropolis-Hastings" for generic Independent Metropolis-Hastings (IMH) algorithm to produce an output the corresponds to a much more improved Monte Carlo approximation machine at the same computational cost.

6 Conclusion

In this study, we show how to leverage the parallel computing capabilities of a GPU in a block independent Metropolis-Hasting algorithm. It is important to note that there are some limitations in the acceleration, such as the bottleneck of global shared memory access and overheads in shared memory allocation.

Regardless, our study has shown a significant improvement in performance by parallelizing the sampling computations in independent Metropolis-Hasting algorithm, without jeopardizing the Markovian convergence properties of the algorithm. Though inherently sequential, we were able to exploit the parallelism within the stages of the independent Metropolis-Hasting algorithm to accelerate the process with CUDA-enabled GPU.

Furthermore, due to limitations in our permission to access the experiment platform, there was an unsuccessful attempt to calculate the power consumption during computation when using CPU and GPU for sampling. Now that we see CUDA is able to significantly improve the throughput of Metropolis-Hastings sampling, concerns with power consumption will be a more significant consideration.

7 Future Work

There is much potential in this field. The last ten years have seem the rapid development of GPUs. They are becoming more accessible with increasing performances. Therefore, future work should extend to more generic MCMC methods. Most parallelization strategies for Bayesian computations are unexplored,

8 Acknowledgement

We would like to share out gratitude towards Professor Alvin R. Lebeck and the TA team for COMPSCI550 Fall 2019 who supported us and provided guidance throughout the research, Joe Shamblin for help with CUDA setup on cluster.

References

[1] Mykland, P., Tierney, L., and Yu, B. *Regeneration in Markov Chain Samplers*. Journal of the American Statistical Association, 1995.

[2] Craiu, R., Rosenthal, J., and Yang, C. *Learn From Thy Neighbour: Parallel-Chain and Regional Adaptive MCMC*. Journal of the American Statistical Association, 104 (408), 1454–1466.

[3] Jacob, P., C. Roberts, and M. Smith. *Using Parallel Computation to Improve Independent Metropolis-Hastings Based Estimation*. Journal Of Computational And Graphical Statistics 20, 1–18.

[4] A. F. da Silva. cudaBayesreg. *Bayesian computation in CUDA*. The R Journal, 2(2):48–55, 2010.

[5] Nvidia Corporation *Tesla K80 GPU Accelerator White Paper*. $BD-07317-001_v05$, January 2015

[6] Rafia Inam *An introduction to gpgpu programming-cuda architecture*. Malardalen University, Malardalen Real-Time Research Centre , 2011

[7] Calderhead, B. *A general construction for parallelizing MetropolisHastings algorithms*. PNAS, 111 (49) 17408-17413, 2014.

[8] https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform, accessed December 2019

[9] Ilker Yildirim, *Bayesian Inference: Metropolis-Hastings Sampling*. MIT.edu, August 2012

[10] Tesla K80 https://wiki.gacrc.uga.edu/wiki/GPU_Hardware, accessed December 2019

[11] Maclaurin, D., Adams, R. P. *Firefly Monte Carlo: Exact MCMC with subsets of data*. In N. L. Zhang, J. Tian (Eds.), Uncertainty in Artificial Intelligence - Proceedings of the 30th Conference, UAI. AUAI Press. 2014.

[12] Quiroz, M., Kohn, R., Villani, M. Tran, M. *Speeding Up MCMC by Efficient Data Subsampling*. Journal of the American Statistical Association, 2019.

[13] Robert, C., and Casella, G. *Monte Carlo Statistical Methods (2nd ed.)*. JNew York: Springer-Verlag. [616, 618,619,623], 2004.

[14] Mainak Adhikari and Sukhendu Kar. *Advanced Topics GPU Programming and CUDA Architecture*. 2016.