Markov Chain Monte Carlo on the GPU
Final Project, High Performance Computing

Alex Kaiser
Courant Institute of Mathematical Sciences, New York University

December 27, 2012

1 Introduction

The goal of this project is to implement a Markov chain Monte Carlo (MCMC) sampling algorithm called the *Stretch Move* for GPU hardware. The algorithm was developed at the Courant Institute in 2010 by Jonathan Goodman and Jonathan Weare [2]. It is effective at producing high quality samples and efficient in terms of samples produced per second. Theoretically, the algorithm parallelizes completely with relatively minor communication. I would like to investigate whether running in parallel on GPU hardware is practical, how the real performance scales with additional parallelism and whether the ultimate speedup is enough for applications users.

If successful, this code will show that a highly parallel implementation of this algorithm is effective. If this new implementation really is superior, maybe it will be adopted by applications scientists. Astrophysicists using this and other MCMC samplers report running for multiple days. Perhaps we can take 4 days to one, or one day to a few hours for some applications user. If even one scientist reports this kind of speedup on a real problem, this project is a success.

If the project fails completely, it will suggest that this hardware is not effective for this algorithm. Or at least that a much more detailed investigation is required to find how to implement it effectively.

To state the problem formally, suppose one has a continuous, multivariate random variable X with probability density function (PDF) proportional to a function f, where

$$f : \mathbb{R}^n \to [0, \infty)$$

and

$$\int_{\mathbb{R}^n} f(x)\, dx < \infty.$$

That is, the integral of f need not be known, but must be convergent over the whole space. Additionally, it is known how to evaluate f, perhaps in closed form or perhaps only through some complicated numerical algorithm. For the duration of this paper, we only consider dimensions N greater than or equal to two. The central question is this: Given f, how can we produce numbers, or samples, that behave approximately according to the distribution of X? What algorithm can be used to produce these samples?

These samplers have many applications. One common computation is to compute the expected value of something. Let $x_1 \dots x_K$ be samples from the distribution. Then an estimator for the expected value is given

$$E[\phi(X)] \approx \frac{1}{K} \sum_{i=1}^{K} \phi(x_i).$$

Samplers can also be used to compute deterministic integrals, usually in high dimensions, by phrasing them as an expectation.

$$\int \phi(x) f(x)\, dx = E[\phi(X)] \approx \frac{1}{K} \sum_{i=1}^{K} \phi(x_i)$$

There are additional applications to inference problems and running simulations in statistical physics.

2 MCMC Methods

The algorithms we will investigate are called Markov chain Monte Carlo, or MCMC, methods. Here we summarize some relevant definitions and concepts, discuss *Metropolis*, the oldest and most used MCMC sampler, and introduce the Stretch Move algorithm.

2.1 Random Walks, Markov Chains and Detailed Balance

(These descriptions are loosely based on those of Kalos and Whitlock [3].)

An MCMC method generates samples by moving in a *random walk*, that is some sequence $X_1 \dots X_t$ for which the subsequent samples are determined according to stochastic dynamics which depend on the algorithm.

We say that a sequence of samples $X_1 \dots X_k$ has the *Markov property* if

$$P(X_{t+1} = x | X_t \dots X_1) = P(X_{t+1} = x | X_t).$$

That is, the conditional distribution of X_{t+1} given all previous elements of the sequence is independent of all but the previous state. A random walk with the Markov property is called a *Markov chain*.

A Markov chain is said to be *irreducible* if any state in the chain can reach any other in a finite number of steps. Informally, there are no "islands" of probability such that the chain cannot get to the other area. A Markov chain is called *aperiodic* if for any initial state, it is possible to return in an irregular number of times. Formally, the gcd of the number of steps in which the chain can return must be one. For example, consider a random walk on the integers that moved either left or right with probability $1/2$ from the current state. If the initial state $X_0 = 0$ would not be able to return to any even state in a any odd number of steps, so the chain is not aperiodic. If the random walk had some probability $1/3$ of moving left, right or staying still, it would be aperiodic since the state can return in any number of moves. We call an irreducible, aperiodic Markov chain *ergodic*.

Another important property convergence is called *detailed balance*. Suppose that the next value in the random walk is determined by stochastic dynamics $K(Y|X)$, where K is a conditional

probability density function of Y given the current state X. The detailed balance condition is specified

$$K(X|Y)f(Y) = K(Y|X)f(X)$$

Note that for a continuous distribution the quantities involved are not actually probabilities, but values of the PDF at a specific point.

For proofs of convergence of MCMC methods, ergodicity and detailed balance are used to prove the existence and uniqueness of the limiting distribution. There is much more subtlety in these properties, which is beyond the scope of this report. What is important for general MCMC methods is that these properties are typically required to prove the correct convergence of the random walk.

Also, since the convergence is only asymptotic, depending on the distribution to sample it may be necessary to run a "burn-in." That is, the first few steps of iteration are thrown out since the initial distribution is not necessarily a valid sample of the distribution.

2.2 Metropolis

The standard algorithm for this type of sampling problem is called the *Metropolis algorithm*, introduced by Metropolis, Rosenbluth, Rosenbluth, Teller and Teller in 1953 [4]. A single step of the algorithm works as follows:

- Let X_t be the sample at time t.
- Propose a move Y by sampling from a *proposal distribution* $T(Y|X)$.
- Evaluate a likelihood function for the move

  $$q(Y|X_t) = \frac{T(X_t|Y)f(Y)}{T(Y|X_t)f(X_t)},$$

  or in the uniform case the T cancel

  $$q(Y|X_t) = \frac{f(Y)}{f(X_t)}.$$

- If $q(Y|X_t) > 1$, accept. Otherwise, accept with probability q.
- If we accept $X_{t+1} = Y$, else $X_{t+1} = X_t$.

The final two steps can be phrased alternately as follows: Define $a(Y|X) = \min(1, q(Y|X))$, sample $p \sim U(0,1)$, accept if $a(X|Y) > p$. Most references use this notation.

Heuristically, the algorithm works by taking an arbitrary move near the current point. If the PDF has greater value at that point, then we always move there. If the PDF has a smaller value, then we accept with probability dependent on how much less likely.

The algorithm generates a sequence that is ergodic, and it satisfies detailed balance. The proposal distribution at time $t+1$ depends only on the previous state, not any prior to that, and the single random number generated for the acceptance step is independent of everything else. Then the

sequence is indeed a Markov chain. Also, The proposal distribution T must be made such that the chain is irreducible. If the support of the PDF is connected, then a uniform T satisfies this. But if the PDF is supported only on two or more disjoint sets, the T must be designed accordingly. The algorithm includes a rejection step, and thus the dynamics it specifies are aperiodic because the distribution can return to any current state through a rejection. Then with a properly designed T the sequence generated is ergodic.

To see that the distribution satisfies detailed balance, let X and Y be arbitrary states. The dynamics are $a(Y|X)\ T(Y|X)$, so the detailed balance condition for this algorithm is

$$a(Y|X)\ T(Y|X)\ f(X) = a(X|Y)\ T(X|Y)\ f(Y).$$

Suppose first that $q(X|Y) > 1$, so $q(Y|X) < 1$ and $a(Y|X) = q(Y|X)$. Then

$$\begin{aligned} a(Y|X)\ T(Y|X)\ f(X) &= \frac{T(X|Y)f(Y)}{T(Y|X)f(X)}\ T(Y|X)\ f(X) \\ &= T(X|Y)f(Y) \\ &= a(X|Y)\ T(X|Y)\ f(Y), \end{aligned}$$

and similarly for the $q(X|Y) < 1$ case. Thus, detailed balance is satisfied. The full proof of the convergence of this method requires properties of ergodic Markov chains and conditions implied by detailed balance, but is beyond the scope of this report.

One advantage of this method is that it does not require knowledge of normalization constants, which will be discussed further in the descriptions of individual algorithms. Because of the form of the ratio, any constant will cancel.

This method, however, has limitations. Ideally, the samples generated would be independent, but since they are generated in a chain, they are correlated. This can be measured by the *autocorrelation time*, denoted τ, of the sequence [2]. This is defined as

$$\tau = \sum_{t=-\infty}^{\infty} \frac{C(t)}{C(0)}$$

where $C(t)$ is the lag t *autocovariance*, defined as

$$C(t) = \lim_{s\to\infty} \mathrm{Cov}(X_{t+s}, X_s).$$

The number of effective independent samples is given by M/τ, where M is the total number of samples. Metropolis tends to have high τ, and this is a major disadvantage of the method.

Another disadvantage is that the algorithm is inherently serial. If evaluating the PDF is very time consuming, it may be possible to parallelize inside that routine, but that is about the limit.

The third major disadvantage is that the algorithm is sensitive to skewed, or *anisotropic*, distributions. Consider a two dimensional Gaussian with PDF given

$$f(\vec{x}) \propto \exp\left(-\frac{(x_1 - x_2)^2}{2\epsilon} - \frac{(x_1 + x_2)^2}{2}\right),$$

taken from [2] and displayed in figure 1. Despite that this would be easily sampled by other methods and is only two dimensional, Metropolis has difficulty if the standard uniform proposal distribution is used. To see this, note that if moving in the (1,1) direction the steps would need to be of order one to sample the distribution in a reasonable number of moves. However, steps in the (-1,1) direction would need to be much smaller, because a large step moves to a region with low probability and is likely to be rejected.

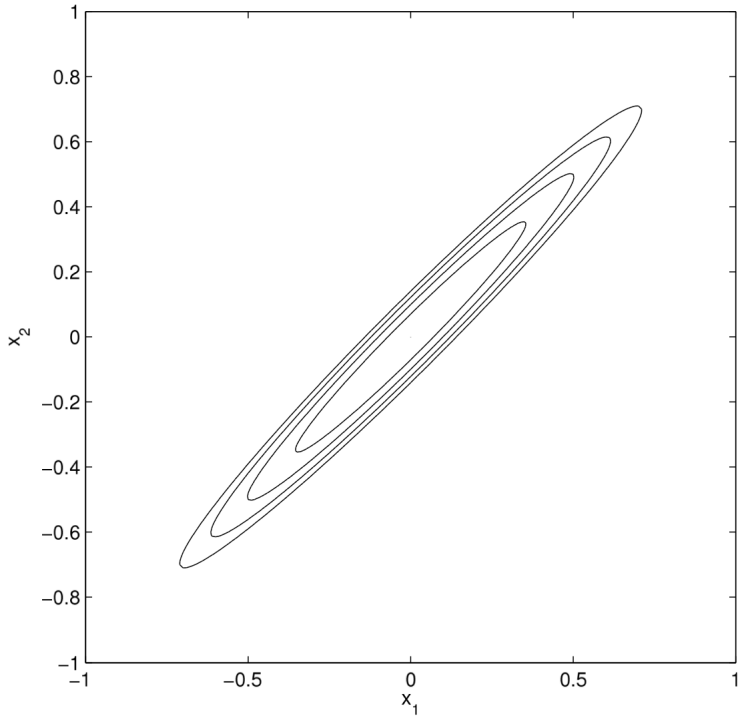

Figure 1: An anisotropic Gaussian distribution

2.3 The Algorithm: Stretch Move

Fortunately, there is an algorithm that addresses all of these concerns. This algorithm is called the Stretch Move, and was developed at the Courant Institute by Jonathan Goodman and Jonathan Weare in 2010, see [2]. The algorithm has a similar structure to Metropolis, and still uses a proposal and an accept/reject step. In this algorithm, we use a group or *ensemble* of sequences, also called *walkers*, since each one is proceeding in its own random walk. Each walker on each iteration of the algorithm represents a valid sample of the distribution. Thus, on each iteration the algorithm generates multiple samples. To update one walker, another walker is randomly selected from the *complementary ensemble*. A move is then proposed alone a line between the current walker k and the complementary walker j according to the distribution

$$Y = X(j) + z(X(k) - X(j))$$

and then accepted or rejected. This generally gives better autocorrelation time as compared to other MCMC methods.

The algorithm is efficient. It requires a comparable amount of arithmetic to update a single walker to other MCMC methods. Crucially, the algorithm can be parallelized. To accomplish this, split the walkers into two groups X_{red}, X_{black}. If there are K total walkers, then the algorithm allows precisely $K/2$ updates to be performed in parallel. This splitting allows the detailed balance condition to remain satisfied when the walkers are updated in parallel.

The algorithm is *affine invariant*, which means that for A, a linear transformation, and vector b, sampling a PDF $g(x) = Af(x) + b$ is equivalent to sampling f then applying A and b after. Heuristically, this means that the algorithm is not sensitive to skewed distributions.

A description of a single iteration of the algorithm to updated one walker is as follows. All of the walkers in each group can be updated completely in parallel. The algorithm uses an auxiliary random variable Z described in 2.3.1.

- To compute $X_{red}(k, t+1)$
- Randomly select a walker from the complementary ensemble $X_{black}(j)$
- Sample $z \sim g(Z)$.
- Compute a proposed move Y

$$Y = X_{black}(j) + z(X_{red}(k,t) - X_{black}(j))$$

- Compute a likelihood

$$q = z^{N-1}\frac{f(Y)}{f(X_{red}(k,t))}$$

- If $q > 1$, accept. Otherwise, accept with probability q.
- If accept $X_{red}(k, t+1) = Y$, else $X_{red}(k, t+1) = X_{red}(k, t)$.

The additional factor of z^{N-1} in the likelihood ratio is to ensure that detailed balance is satisfied.

2.3.1 The Random Variable Z

The algorithm requires an auxiliary random variable Z PDF $g(z)$, which must satisfy $g(1/z) = (1/z)g(z)$. This distribution is usually $g(z) = 1/\sqrt{2z}$ on $(1/2, 2)$, and this is the only distribution used in this project and paper.

To generate the samples of the random variable Z, we will use the standard technique of inverting CDFs. Let $a = 1/2$. To get the PDF of $g(z)$ we have $\int_{1/2}^{2} z^{-1/2}\, dz = \sqrt{2}$ so

$$g(z) = \begin{cases} \frac{1}{\sqrt{2z}} & : z \in (1/2, 2) \\ 0 & : else \end{cases}$$

The CDF is given

$$F_Z(z) = \begin{cases} 0 & : z \leq 1/2 \\ \sqrt{2z} - 1 & : z \in (1/2, 2) \\ 1 & : 2 \leq z \end{cases}$$

Let $X \sim U(0,1)$, then use the standard relationship of the CDFs, $F_Z(z) = F_X(h^{-1}(z))$ to find the appropriate transformation. Expand each of the CDFs by their definition to find h.

$$\sqrt{2z} - 1 = h^{-1}(z)$$

so

$$h(x) = \frac{1}{2}x^2 + x + \frac{1}{2}$$

Then to sample Z, sample X and return $h(x)$.

3 Related software

One related package is called emcee: the MCMC hammer [1]. This package implements the Stretch Move in Python and has been successfully used in inference problems. The algorithm is sufficiently new that there are not many libraries that use it, so if a significant speedup would have real impact. It is possible that this code will be included in emcee in the future, which would hopefully give the code a quick audience.

The random number generator used is ranluxcl [5], and the kernel requires precisely three random numbers per update of a single walker.

4 GPU Implementation and goals

The GPU implementation is written in OpenCL. Effort has been made to keep the code concise and straightforward. The kernel evaluates all the moves for a group, so a new kernel is launched twice for each iteration. The group of walkers is brought back to the host on each iteration for use in histograms or other computations.

The main goal is to be able to make very many samples quickly. More specifically, the goal is to see whether the code is much faster than emcee, in which case it will be useful for applications users of that software.

At all sizes, latency of transferring the samples to the host should be significant, and is likely the highest cost of the algorithm besides evaluating the PDF. Also, the since hundreds or thousands of kernels will need to be launched, at all levels the kernel launch time will be significant. If the number of walkers is very high such that not that many iterations are required, this cost will be reduced. If this number of walkers is small and many iterations are required, it's possible that this becomes the dominant cost of the computation.

At upper limits, the kernel will use very many walkers and will be limited by GPU memory size. For larger test problems allocation did fail, and tests have been run up to sizes at the edge of failure.

For smaller problems that do not function well with very many walkers, other approaches (most likely serial) may be superior, since the latency with launching and reading samples back to the host is high.

Throughout, we will use two simple test problems. The first is a Gaussian debug problem specified by

$$f(\vec{x}) \propto \exp\left(\sum_{i=0}^{N}(x_{i+1}-x_i)^2\right)$$

where $x_0 = x_{N+1} = 0$. The second is the same problem but with a restriction that each component is nonnegative. That is

$$f(\vec{x}) \propto \exp\left(\sum_{i=0}^{N}(x_{i+1}-x_i)^2\right)\ r(\vec{x})$$

where $r(\vec{x}) = 1$ if all components are nonnegative and zero otherwise.

4.1 Expectations

I expect the performance to suffer somewhat compared to the serial version from the overhead of memory transfer. Once the baseline is in place, the performance of the sampler kernel in samples per second should scale roughly linearly as the number of work items increases. The two evaluations of the PDF are likely the most expensive part of the computation, and being able to execute them fully in parallel should give the code strong scaling. For a distribution for which the PDF takes a consistent amount of time to evaluate, the algorithm should be load balanced well. Because of latency issues in reading various quantities, especially the opposing walker from global, and the further benefits of latency hiding, more work items should make for better performance. This scaling will likely stop once all the latency is hidden, that is when the various schedulers and pipelines are filling the entire GPU.

Another expectation is that performance will slow slightly because of burn-in. The burn-in time usually thought of as a fixed number of iterations, regardless of the total number of walkers used. Thus the parallelism does not save any time on this step. However, in measurements of total time, the burn in is should not be that significant. In particular, there is no communication with the CPU since the samples are thrown out, copying them is a main cost of the algorithm.

4.2 Performance Measurements and tuning

All tests in this section were run on an Nvidia GeForce GTX 590.

One performance consideration is whether to use local memory for the walker that is moving and the proposal, or to declare them private and take the risk that the private array will move to global. To local versus private evaluate this, we will look performance of both versions on a ten dimensional Gaussian test problem. Figure 2 shows that the performance is generally slightly better with local, so we will use this version of the code.

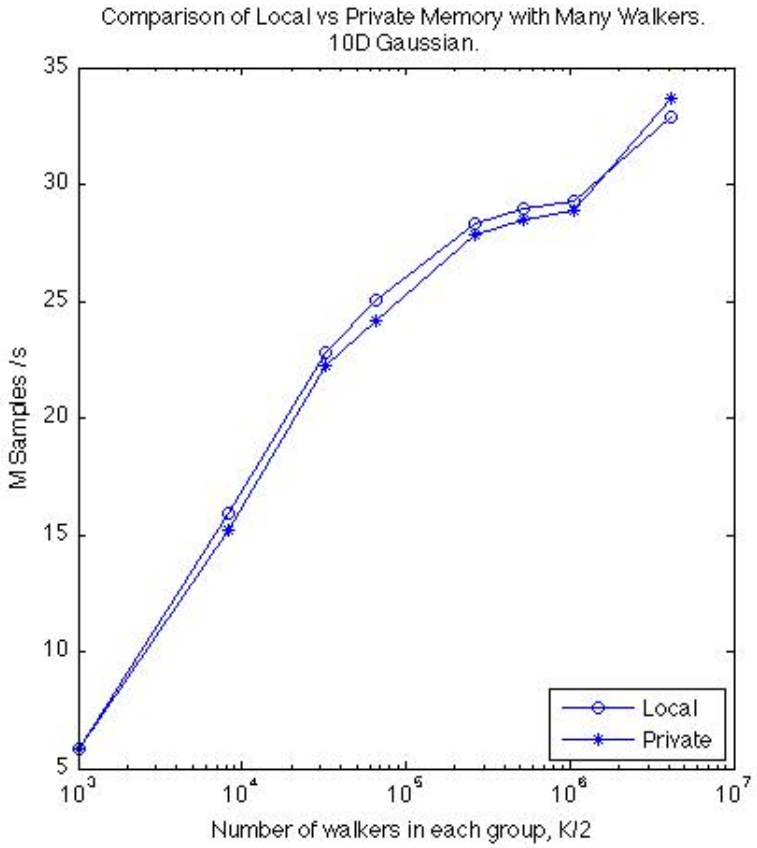

Figure 2: Scaling on local and private memory

An important check is that these samples are right, so a histogram on the final version is shown in figure 3. A histogram of the Gaussian with restriction in 20 dimensions is shown in 4, computed at a sample rate of 11.3 m. samples/s. on the kernel and 8.0 m. samples/s. in total time. Disparity in the kernel versus total performance is higher, most likely since the code is doing more processing.

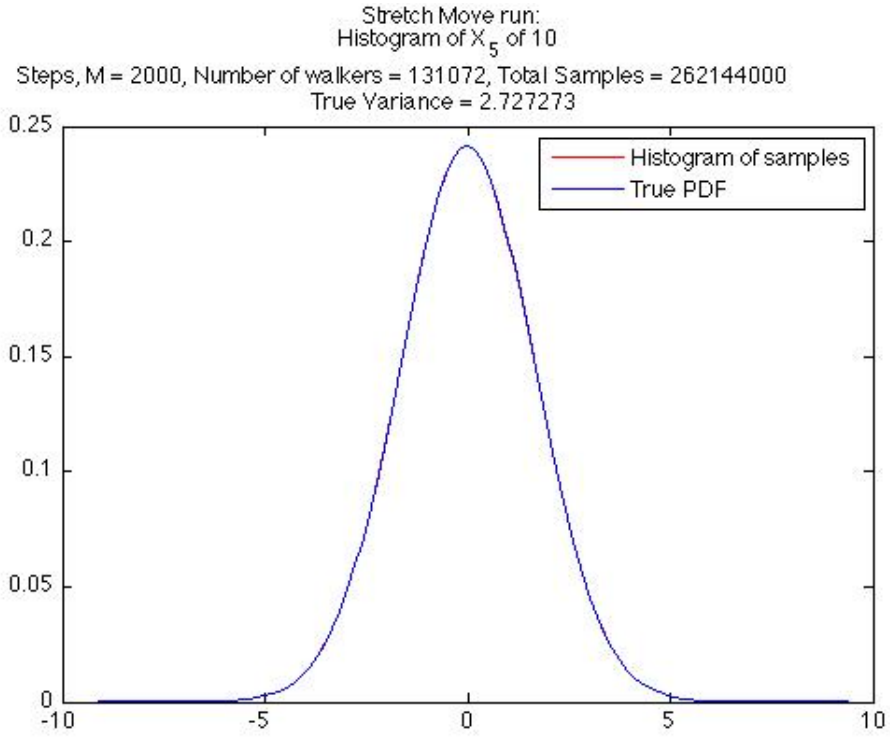

Figure 3: Correct histogram of 10D Gaussian debug problem. Note that the histogram and the true PDF are indistinguishable.

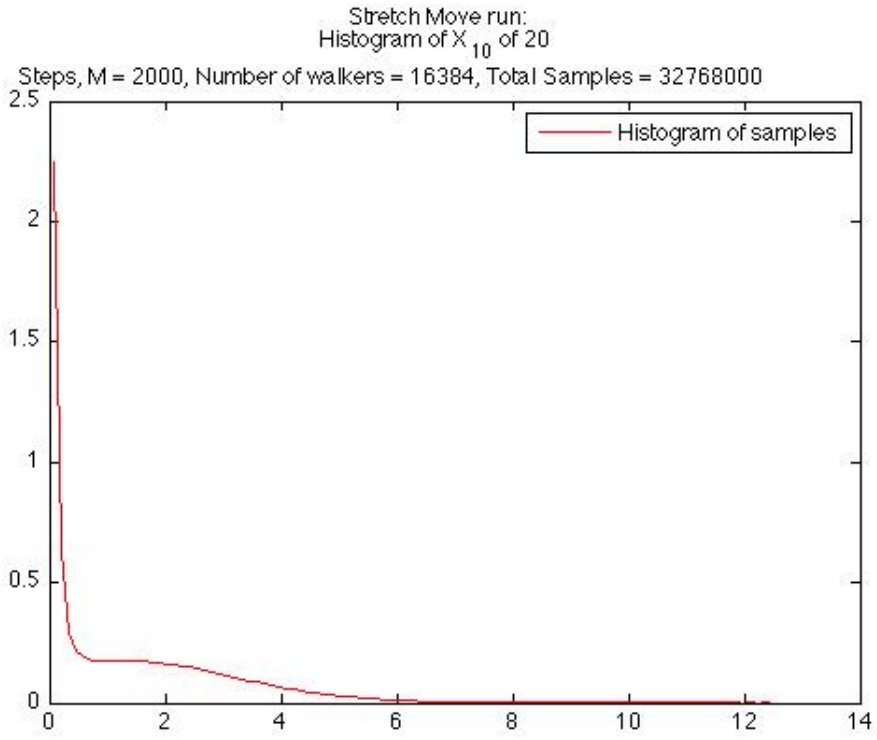

Figure 4: Histogram of 20D Gaussian with restriction.

Figure 5 shows the a performance scaling curve for the 20 dimensional Gaussian. The scaling is roughly linear as expected. Note that a baseline value for this plot with 1024 work-items is 5.1 m. samples/s. on the kernel and 3.3 m. samples/s. in total time. The burn-in time does not appear to be a large factor and appears to be consistent, because the total time is a roughly constant amount lower than the kernel time. One high cost is that a new kernel must be launched two times for each iteration. To create the full barrier to global memory, a new kernel must be launched. When running hundreds or thousands of iterations, this seems to be a large cost.

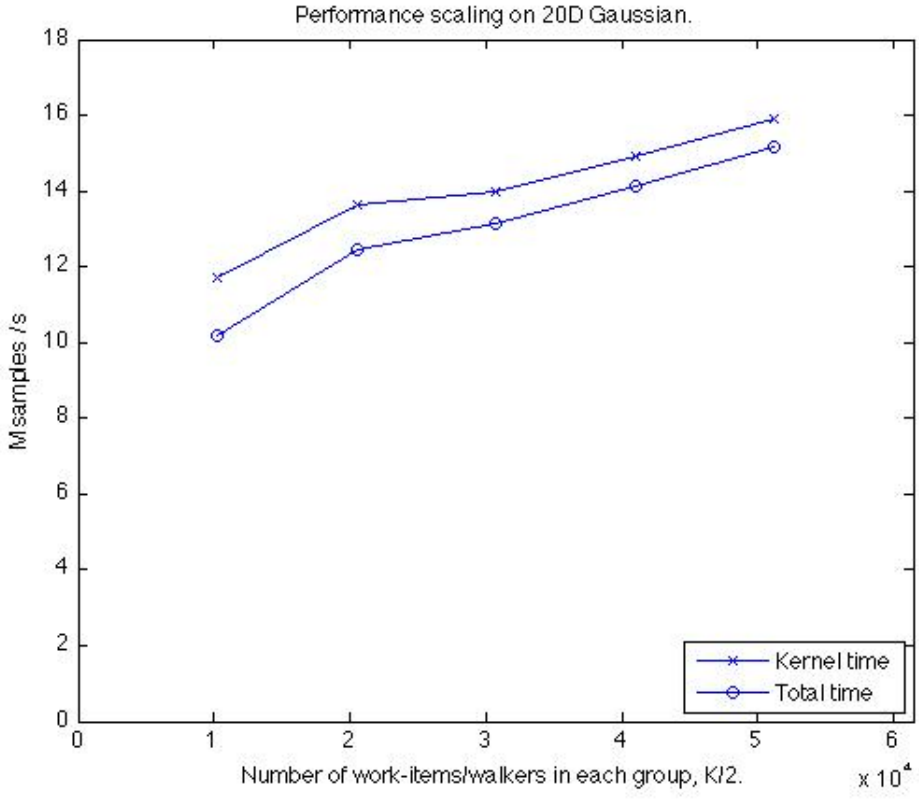

Figure 5: Performance scaling on a 20D Gaussian.

If the dimension is large, and the number of walkers is also very large, then the algorithm has problems converging to the correct distribution. It is not clear what is happening, or why this convergence is so poor. More analysis, both numerical and theoretical, is needed to understand this behavior. This error on a 50 dimensional Gaussian is shown in figure 6. Performance for this test is 5.56 m. samples/s. kernel time, and 4.74 m. samples/s. total.

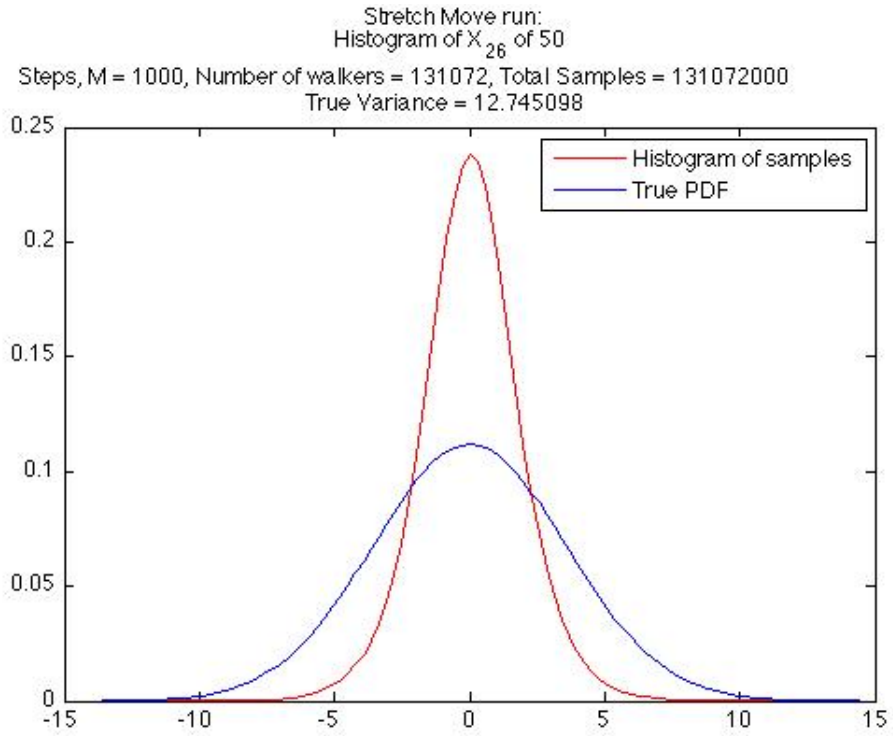

Figure 6: Errors on high dimensional problems with very many walkers.

One final aside on performance, a previous version used a barrier within the kernel, rather than a restart. This creates a race condition on read of the walkers — the algorithm may read an old walker because the write has not fully gone through. This race condition was fixed, but fixing had a dramatic effect on overall performance. Performance on a 10 dimensional Gaussian is shown in figure 7, and is really much faster than synchronizing by launching a new kernel. If there is a more efficient method of ensuring correct reads of the data, then this would be of serious benefit. Even an expensive and complicated method with atomics may be better than launching a new kernel and is worth looking into for the future.

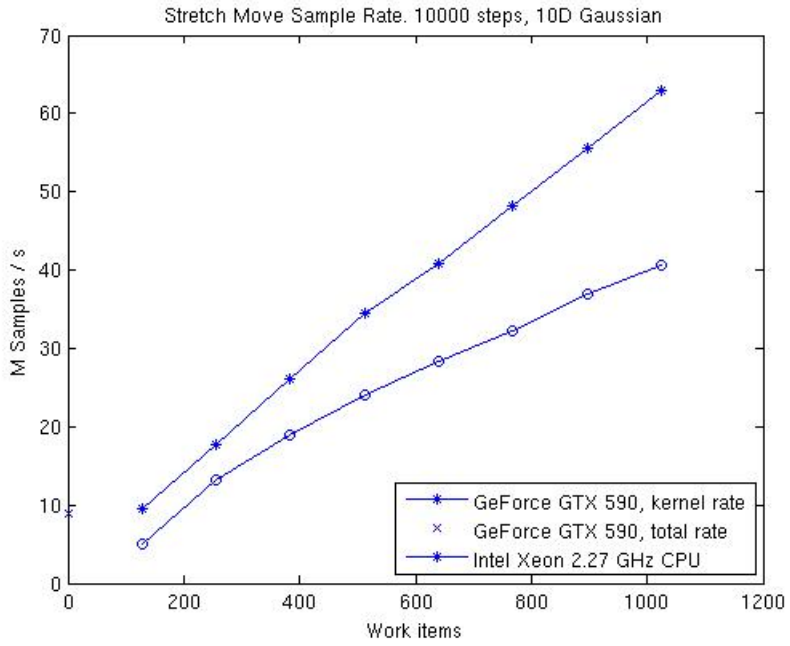

Figure 7: Performance scaling with data race. Performance is significantly better than with correct synchronization, so more sophisticated synchronization strategies are worth pursuing.

5 Conclusions and Further Work

This code has shown that a GPU implementation of the Stretch Move algorithm sometimes performs well on the GPU. Reasonable and significant improvements have been seen compared to the implementation running serially.

The next thing to test is harder problems. In particular, I plan on contacting scientists who have cited use of the algorithm. It will be very instructive to see whether the code is effective on real inference problems. I would like to further analyze the success and failures in using very many walkers, especially on high dimensional problems. It would also be useful to do a more sophisticated evaluation of the quality of the random numbers from ranluxcl.

In an applications use, a more sophisticated approach might use the samples on the GPU. If the computation of the samples is some sort of reduction operation, then getting the result back to the CPU could be much faster than getting the samples back every iteration.

One additional goal is to make the code more user friendly. As is, the code requires modification of many files and there are many constants. Ideally the code would require less tinkering for a user who is not familiar with OpenCL.

References

[1] Foreman-Mackey, D., Hogg, D. W., Lang, D., and Goodman, J. emcee: The MCMC Hammer. *ArXiv e-prints* (Feb. 2012).

[2] Goodman, J., and Weare, J. Ensemble samplers with affine invariance. *Commun. Appl. Math. Comput. Sci. 5* (2010), 65–80.

[3] Kalos, M. H., and Whitlock, P. A. *Monte Carlo methods. Vol. 1: basics.* Wiley-Interscience, New York, NY, USA, 1986.

[4] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics 21* (June 1953), 1087–1092.

[5] Nikolaisen, I. U. ranluxcl v1.3.1, 2011. https://bitbucket.org/ivarun/ranluxcl/.

6 Appendix - Code

6.1 Building

The included makefile should build the code on any generic Linux or OSX system using GNU compilers. Tests were run on the Rice University system "Box" using Nvidia GPUs. Debugging and other test runs also has been run and tested on a stock Apple laptop on the CPU.

6.2 Options

The following constants can be modified in `stretch_move_main.c`. Various parameters that define the output and flow of the program are set using definitions, and parameters for the sampling itself are set in the main routine of the code.

Constant	Purpose
(*definitions*)	
DEBUG	Print a few samples to eyeball
OUTPUT_FULL_DATA	Output a file containing the full generated sequence of samples
OUTPUT_HISTOGRAMS	Compute histograms for the samples
GNU_PLOT_HISTOGRAMS	Write a gnuplot file for the histograms
MATLAB_HISTOGRAMS	Write a matlab file for the histograms to be read by the included script `plot_hist_from_files.m`
NONNEGATIVE_BOX	Start all the components of the walkers with nonnegative values
RUN_ACOR	Run the acor module. Currently unreliable, use with caution
PDF_NUMBER	Which PDF to use. See below.
(*Declared in "main"*)	
M	Number of steps to run
N	Dimension of the problem and the walkers
K_over_two	Number of walkers in each group
burn_num	Destroy with this many moves before using samples
num_to_save	Number of samples to save
indices_to_save_host	Indices of the components to process This is an array of length num_to_save

6.3 Arranging Your PDF to sample

The probability density functions to sample are specified in the file `stretch_move.cl`. Before the kernel code begins, there is a function with the following header:

```
float pdf(__local float *x)
```

The body of this function depends on a constant `PDF_NUMBER`, defined in `stretch_move_main.c`. Add an additional if clause and place the code to evaluate the PDF here. Two simple examples are included in the file. If the constant is unchanged, the code defaults to the Gaussian debug problem. One consequence of this is that the PDF must be a "pure function" of the walker X, that is, its result may depend only on the input. As implemented, for a dimension N PDF, the function must be

$$f : \mathbb{R}^n \to \mathbb{R}.$$

If the PDF to evaluate requires observations or other previously computed data, the function header for the PDF will need to be modified accordingly and data will need to be read on the CPU and passed in. Also, since OpenCL kernels do not permit recursion, the code cannot be recursive. Otherwise, an arbitrarily complicated PDF may be supplied.

Two other points: The algorithm has a feature that it does not function correctly or converge if given a sample with zero probability. This means that the initial walkers must be initialized so they have nonzero probability. Second, the algorithm is fails to work if all walkers are identically zero, thus they must be initialized to some other value.

6.4 Running

Run the executable `stretch_move_main`. The program takes no command line inputs, all changes the user may wish to make are in the source files.

If Matlab histograms are output, then `plot_hist_from_files.m` can plot them. If gnuplot histograms are output, then running gnuplot on the data files will produce histograms, the output is already formatted. If a full sequence of samples is output, then `plot_data.m` will compute and plot histograms.

6.5 Getting the code

The code is currently in my private repository on Forge. I will release publicly (probably on github, maybe as part of emcee) upon further investigation and testing.