Parallel Multivariate Slice Sampling

Matthew M. Tibbits
Department of Statistics
Pennsylvania State University
mmt143@stat.psu.edu

Murali Haran
Department of Statistics
Pennsylvania State University
mharan@stat.psu.edu

John C. Liechty
Departments of Statistics and Marketing
Pennsylvania State University
jcl12@psu.edu

March 23, 2010

Abstract

Slice sampling provides an easily implemented method for constructing a Markov chain Monte Carlo (MCMC) algorithm. However, slice sampling has two major drawbacks: (i) it requires repeated evaluation of likelihoods for each update, which can make it impractical when evaluations are expensive or as the number of evaluations grows (geometrically) with the dimension of the slice sampler, and (ii) since it can be challenging to construct multivariate updates, the updates are typically univariate, which often results in slow mixing samplers. We propose an approach to multivariate slice sampling that naturally lends itself to a parallel implementation. Our approach takes advantage of recent advances in computer architectures, for instance, the newest generation of graphics cards can execute roughly $30,000$ threads simultaneously. We demonstrate that it is possible to construct a multivariate slice sampler that has good mixing properties and is efficient in terms of computing time. The contributions of this article are therefore twofold. We study approaches for constructing a multivariate slice sampler, and we show how parallel computing can be useful for making MCMC algorithms computationally efficient. We study various implementations of our algorithm in the context of real and simulated data.

1 Introduction

It is well known that Markov chain Monte Carlo (MCMC), which is based on the Metropolis-Hastings algorithm, provides a very general approach for approximating integrals (expectations) with respect to a wide range of complicated distributions. When full conditional distributions are non-standard, the process of constructing MCMC algorithms is far from automatic and even when it is possible to sample from the full conditionals via Gibbs updates, the resulting samplers can exhibit poor mixing properties. It is often the case that the Metropolis-Hastings algorithm needs to be carefully tailored to each particular distribution and the search for an appropriate proposal distribution with good mixing properties can be time consuming. The slice sampler (Damien et al., 1999; Mira and Tierney, 2002; Neal, 1997, 2003a) has been proposed as an easily implemented method for constructing an MCMC algorithm and can, in many circumstances, result in samplers with good mixing properties. Slice sampling has been used in many contexts, for example in spatial models (Agarwal and Gelfand, 2005; Yan et al., 2007), in biological models (Lewis et al., 2005; Shahbaba and Neal, 2006; Sun et al., 2007), variable selection (Kinney and Dunson, 2007; Nott and Leonte, 2004), and machine learning (Andrieu et al., 2003; Kovac, 2005; Mackay, 2002). Slice samplers can adapt to local characteristics of the distribution, which can make them easier to tune than Metropolis-Hastings approaches. Also, by adapting to local characteristics and by making jumps across regions of low probability, a well constructed slice sampler can avoid the slow converging random walk behavior that many standard Metropolis algorithms exhibit (Neal, 2003a).

A slice sampler exploits the fact that sampling points uniformly from the region under the curve of a density function is identical to drawing samples directly from the distribution. Typically, the slice sampler is used for univariate updates (e.g., sampling from a full-conditional density for a single parameter). While univariate slice samplers may improve upon standard univariate Metropolis algorithms, univariate samplers in general can mix poorly in multivariate settings, especially when several variables exhibit strong dependencies. In such cases, the mixing of the sampler can be greatly improved by simultaneously updating multiple highly dependent variables at once. But, while multivariate slice samplers have been discussed in Neal (2003a), they are rarely used in practice as they can be difficult to construct and computationally expensive to use due to the large number of evaluations of the target distribution required for each update.

In this article, we explore the construction of simple, automatic multivariate slice updates, which take advantage of the latest parallel computing technology available. Since modern computing is moving towards massively parallelized computation rather than simply increasing the power of individual processors, a very interesting and important challenge is to find ways to exploit parallel computing power in the context of inherently sequential algorithms like MCMC. While parallel computing has been explored in a few other MCMC contexts, the focus appears to have been primarily on three areas: (1) parallelizing likelihood evaluations, for example, by distributing matrix computations (cf. Whiley and Wilson, 2004; Yan et al., 2007), (2) exploiting conditional independence to facilitate simultaneous updates of parameters (cf. Wilkinson, 2005), and (3) utilizing multiple Markov chains (Rosenthal, 2000). Here we investigate how updates of slice samplers, particularly the multivariate slice sampler, can be parallelized independently of the form of the likelihood. We can therefore simultaneously parallelize the construction of individual updates, as well as the likelihood computations involved in each update.

Recently, several authors have investigated the utility of using graphics processing units (GPUs) for computationally intensive statistical techniques, for example in statistical genetics (Jiang et al., 2009; Manavski and Valle, 2008; Sinnott-Armstrong et al., 2009; Suchard and Rambaut, 2009), and sequential Monte Carlo (Lee et al., 2009). In contrast to conventional parallel computing which employs perhaps a few dozen CPU-based threads, GPUs have sufficient hardware to support thousands or tens of thousands of simultaneous operations. In the context of MCMC, GPUs are capable of parallelizing the sampling algorithm while simultaneously exploiting conditional independence and performing distributed matrix computations. In short, GPUs will facilitate strategies for constructing efficient multivariate slice samplers that may take advantage of parallel computing in multiple ways.

This article makes two primary contributions: (i) we develop multivariate slice samplers and compare their performance to univariate slice samplers, and (ii) we explore how new developments in parallel computing can be used to make computationally expensive multivariate slice samplers fast and practical. The remainder of the paper is organized as follows. Section 2 outlines the univariate and multivariate slice sampling algorithms, Section 3 examines two different software approaches to parallelism (OpenMP and CUDA), Section 4 examines the performance of the various parallel sampling algorithms in the context of both simulated and real data examples involving a popular class of Gaussian process models, and Section 5 summarizes the effectiveness of the different algorithms and points to future avenues for research. Please note that the GPU kernel code utilized in this manuscript is available for download from the author's website:

`http://www.stat.psu.edu/~mmt143/parallelSliceSampler/`

2 Slice Sampling

In this section we provide a brief overview of the basic slice sampling algorithm highlighting the difference between the univariate and multivariate methods. We then examine the performance of the univariate and multivariate methods within the context of a linear regression example and a linear Gaussian Process example. The multivariate slice sampler displays better mixing, but is much more computationally intensive. We explain how exploiting parallel processing can result in a fast mixing Markov chain which is also computationally efficient.

2.1 Algorithm Overview

The main principle behind a slice sampler is based on the observation that sampling from the distribution of a random variable $X \in \mathbb{R}^n$ is equivalent to sampling from the $(n+1)$ - dimensional space under the plot of $f(x)$ where f is a function proportional to the density of X (Neal, 2003a). The univariate and multivariate slice samplers outlined in this section both use the same basic algorithm. Both samplers augment the parameter space with an auxiliary variable and then construct an appropriate region or "slice" from which a uniform sample is generated. For simplicity, we outline the algorithm in the context of a one dimensional parameter β and then highlight the differences in the multivariate setting. Note that in this section and those that follow we use subscripts to denote the components of a vector and superscripts to index iterations of the MCMC algorithm.

```
1. Sample $h \sim \text{Uniform}\{\, 0,\ f(\,\beta\,)\,\}$

2. Sample $\beta \sim$ Uniform on $A = \{\ \beta : f(\,\beta\,) \geq h\ \}$
```

Algorithm 1: Univariate Slice Sampling Algorithm for Parameter β

The i^{th} realization of β is constructed according to Algorithm 1 as follows. A sample or "height" under the distribution, h^i, is drawn uniformly from the interval $\big(0,\ f(\,\beta^{i-1}\,)\big)$. This height, h^i, defines a horizontal slice across the target density, $A = \big\{\ \beta : f(\,\beta\,) \geq h^i\ \big\}$, which is then sampled from uniformly to generate β^i. In the univariate case, the set $\big\{\ \beta : f(\,\beta\,) \geq h^i\ \big\}$ is simply an interval or, perhaps more generally, the union of several intervals (such as in the presence of multiple modes). In contrast, in the multivariate case, the set $\big\{\ \beta : f(\,\beta\,) \geq h^i\ \big\}$ may have a much more complicated form.

Quite often, one lacks an analytic solution for the bounds of the slice A and so, in practice, an approximation to the slice A is constructed. For the single dimension case, Neal (2003a) suggested two methods, stepping out and doubling, to approximate the set A, though we only consider the step-out approach here. This is done by randomly orienting an interval around the starting location β^{i-1}. The lower bound L^i and upper bound U^i are examined, and if either $f(\,L^i\,)$ or $f(\,U^i\,)$ is above the sampled height h^i, then the interval is extended. Once the interval is constructed, a new location β^i is selected from $(\,L^i,\, U^i\,)$ provided $\beta^i \in \big\{\ \beta : f(\,\beta\,) \geq h^i\ \big\}$. The sample h^i is then discarded, a new sample h^{i+1} is drawn, and the process repeats. The resulting Markov Chain has the desired stationary distribution (cf. Neal, 2003a). While the preceding description applies to both the step-out and doubling approaches, we now explain the step-out method in detail. In the step-out method, the lower bound is examined first and extended in steps equal to the initial interval width (ω) if $f(L^i)$ is above the sampled height h^i. The upper bound is then examined and extended if needed. Once the interval is constructed, a proposed parameter value, $\widetilde{\beta}$, is drawn uniformly from $(\,L^i,\, U^i\,)$. If it falls outside the target slice, $(f(\widetilde{\beta}) < h^i)$, a shrinkage procedure is recommended to maximize sampling efficiency. If the failed proposal $\widetilde{\beta}$ is less than β^{i-1}, then set $L^i = \widetilde{\beta}$. Likewise, if $\widetilde{\beta}$ is greater than β^{i-1}, set $U^i = \widetilde{\beta}$. In this way, the interval collapses on failed proposals and given that the current location must be within the slice, the probability of drawing a point from the slice then increases after each rejected proposal.

In contrast to the univariate slice sampler, which samples from the distribution of a parameter $\beta \in \mathbb{R}^1$, the multivariate slice sampler, which samples from the distribution of $\boldsymbol{\beta} \in \mathbb{R}^k$, constructs an approximate slice $\mathbf{A}$ as a k-dimensional hypercube which bounds the target slice. As before, we update the parameter $\boldsymbol{\beta}$ by drawing a sample h^i uniformly from the interval $\big(0,\, f(\,\boldsymbol{\beta}^{i-1}\,)\big)$ where the current location, $\boldsymbol{\beta}^{i-1}$, is now a k-dimensional vector. Next, an interval is randomly oriented around the starting location $\boldsymbol{\beta}_j^{i-1}$ for each vector component. Then the value of the target density is examined at the vertices of the hypercube, which we will refer to as the lower bound vector $\mathbf{L}^i$ and the upper bound vector $\mathbf{U}^i$. If the value of the density at any vertex falls below the sampled height h^i, then the hypercube is expanded. Once the hypercube is constructed, a new location $\boldsymbol{\beta}^i$ is sampled uniformly from $\mathbf{A}$ subject to the constraint that $\boldsymbol{\beta}^i \in \big\{\ \boldsymbol{\beta} : f(\,\boldsymbol{\beta}\,) \geq h^i\ \big\}$. The sample h^i is then discarded, a new sample h^{i+1} is drawn, and the process repeats.

Multivariate slice sampling is challenging due in large part to the number of likelihood evaluations required at each iteration. First, the number of vertices, 2^k, for the k-dimensional hypercube used to approximate the target slice $\mathbf{A}$ grows exponentially as the dimension of the multivariate slice sampler increases. From a computational standpoint, the work doubles for each additional dimension considered. Second, as the dimensionality of the target distribution increases, the k-dimensional hypercube is more likely to waste space, and consequently, the performance of rejection sampling for the proposal step will deteriorate. Both of these issues can be addressed in the context of the classes of models we consider here by evaluating the likelihoods for the slice construction and proposal testing in parallel as we will see in Section 4. One final issue relates to the tuning and selection of initial interval widths for the hypercube approximation. In shrinking the hypercube in the obvious way (when proposals fall outside the slice), shrinking all dimensions simultaneously performs poorly when the density does not vary rapidly in some dimensions (see Neal, 2003a). Tuning and selection of interval widths may also be challenging. To address this issue, we performed a grid search to find optimal interval widths which maximized ES/sec.

When implementing the univariate and multivariate slice samplers, we chose the step-out method for constructing the approximate slice $\mathbf{A}$. Mira and Roberts (2003) note that the step-out method is unable to move between two disjoint modes that are separated by a region of zero probability where these regions are larger than the step size. This implies that the sampler may not be irreducible for some multimodal distributions when the initial step size is too small; however, this problem does not arise in any of the examples considered here.

2.2 Application to Linear Regression

While the univariate slice sampler is easy to implement and may have good theoretical properties, it can perform poorly, particularly when parameters are highly correlated. We illustrate this with the following two-dimensional example.

Example 1 (Linear Regression with an Intercept). Consider a simple linear regression model. We assume that the errors ϵ_i are independent and normally distributed with known variance of 1:

$$Y_i = \alpha + \beta X_i + \epsilon_i, \qquad \epsilon_i \sim N(0,1), \qquad i = 1, \ldots, N$$

We complete the Bayesian model specification by placing uniform priors on the intercept α and the regression coefficient β.

Gilks and Roberts (1996) noted that for this model, the posterior correlation of α and β is given by

$$\rho_{\alpha\beta} = -\frac{\overline{x}}{\sqrt{\overline{x}^2 + \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2}}$$

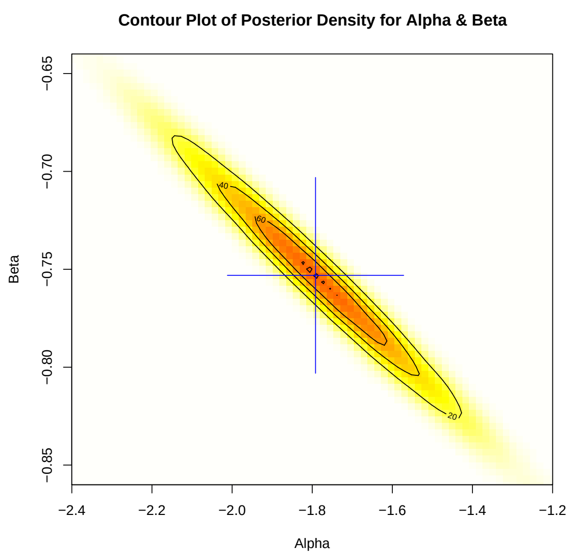

Figure 1: Joint Posterior Density of the intercept (α) and the regression coefficient (β) from Example 1

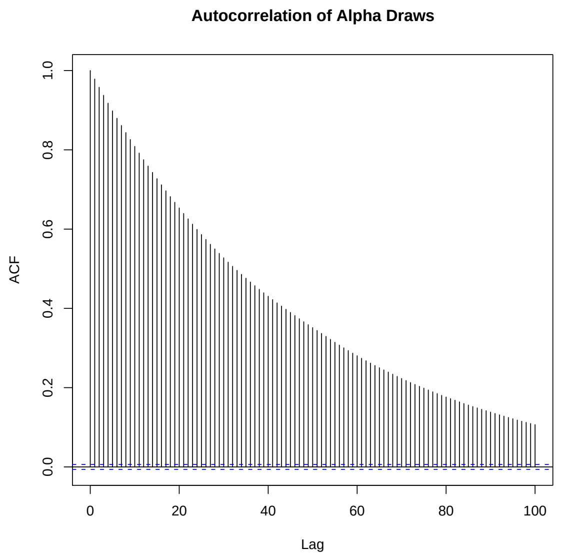

Figure 2: Autocorrelation for intercept parameter (α) using a Univariate Slice Sampler from Example 1

Using this formula, we can control the mean and variance of the predictor in our simulations and arbitrarily fix the posterior correlation of the α and β parameters. Selecting a mean of 5.0 and a variance of 0.5, we generated a dataset with $\rho_{\alpha\beta} = -0.989391$.

The efficiency of competing MCMC samplers may be compared on the basis of effective sample size (ESS) and effective samples per second (ES/sec) as described by Kass et al. (1998) and Chib and Carlin (1999). ESS is defined for each parameter as the total number of samples generated divided by the autocorrelation time τ, given by:

$$\tau = 1 + 2 \sum_{i=1}^{N} \rho(k)$$

where $\rho(k)$ is the autocorrelation at lag k. The summation is usually truncated when the autocorrelation drops below 0.1, though more sophisticated approaches are possible (cf. Geyer, 1992). ESS is a rough estimate of the number of iid draws available in an autocorrelated sample.

For comparison, we ran three samplers to generate draws from the posterior distributions of α and β. The univariate samplers were used to draw sequentially from the full conditional distributions of $f(\alpha|\beta)$ and then $f(\beta|\alpha)$ whereas the multivariate slice sampler was used to sample from the joint distribution, $f(\alpha, \beta)$. The step-out slice sampler was run with interval widths of 0.1 and 0.02 for α and β respectively. A standard univariate random-walk Metropolis Hastings sampler was run with proposal variances of 0.0067 and 0.00026 for α and β respectively. Finally, the multivariate slice sampler was run with interval widths of 0.074 and 0.015 for α and β respectively. For all three samplers, the interval widths and proposal variances were tuned to maximize ES/sec. The highly correlated posterior distribution of α and β which we plotted using $5,000,000$ samples of the univariate step-out slice sampler is shown in Figure 1 (true parameter values are marked by the "+" symbol). It is evident in the slow decay on the autocorrelation plot (Figure 2) that the univariate slice sampler requires several iterations to move from one end of the density to the other. Examining the actual draws, we noted that the sampler would often sit in one tail for a few hundred iterations or more before traversing across the entire distribution.

Table 1: Performance comparison of samplers for α and β from Example 1. All algorithms were run for $5,000,000$ samples.

Algorithm alg:cuda:fwdBackSub	Intercept α		Coefficient β	
	ESS	ES/sec	ESS	ES/sec
Univariate Slice Sampler	53825	1231	56766	1298
Random-Walk Metropolis Hastings	12853	932	13480	933
Multivariate Slice Sampler	4312529	18400	5000000	21333

In comparing the performance of the three samplers, we see that the multivariate slice sampler clearly dominates in terms of both ESS and ES/sec, see Table 1. Both univariate samplers exhibit a high autocorrelation and therefore have a low ESS. The multivariate slice sampler, which has much better mixing properties, exhibits a much lower autocorrelation and therefore has a high ESS. Given the inexpensive likelihood evaluations of a regression model, the multivariate slice sampler also posts a high ES/sec.

In this toy example, the challenges posed by the collinearity of α and β can be easily remedied by standardizing the predictors. However, in complex hierarchical or non-linear models, removing this collinearity through transformations may be very difficult. In Section 2.3, we examine a Gaussian process model where transformations to remove the posterior correlation between κ, ψ, and ϕ are not apparent. Similarly, a funnel-shaped example is provided in Neal (2003a) where the univariate slice sampler performs poorly because the optimal interval width changes for different regions of the density. Roberts and Rosenthal (2002) present an interesting example where their polar slice sampler's convergence is shown to be insensitive to the dimension of the target distribution. However, as Neal (2003b) mentions, the polar slice sampler requires fixing an origin for the polar coordinates and this may be difficult with little or no prior knowledge of the distribution.

Multivariate sampling of blocked parameters is a general approach to accelerating MCMC convergence (cf. Liu et al., 1994; Roberts and Sahu, 1997). Hence, the problems posed by strong dependence among parameters may be mitigated by using a multivariate slice sampler, which is much more adept at navigating complicated density functions.

2.3 Application to Gaussian Process Models

In Section 2.2, we compared the performance of univariate and multivariate slice sampling algorithms in the context of a toy example. Example 1 was chosen to highlight the shortcomings of a univariate sampling algorithm (slice sampler or otherwise) when applied to a model with a highly correlated posterior distribution. However, in this example the likelihood evaluations are computationally inexpensive, making it less challenging than many other standard Bayesian models. To examine the performance of the multivariate slice sampler within the context of a more realistic example, we consider the linear Gaussian process model.

Example 2 (Linear Gaussian Process Model). Consider a linear Gaussian process model with an exponential covariance function, a very popular model for spatial data (cf. Banerjee et al., 2004; Cressie, 1993). We model a spatially-referenced response $Y(\mathbf{s}_i)$ measured at a locations s_i with covariates $X(\mathbf{s}_i)$ $(i \in \{1 \dots N\})$. The responses at the locations are correlated based on magnitude of separation and this correlation falls off at an exponential rate.

$$Y(\mathbf{s}_i) = X(\mathbf{s}_i)\boldsymbol{\beta} + \epsilon(\mathbf{s}_i), \qquad \epsilon(\mathbf{s}_i) \sim N(0, \Sigma(\mathbf{s})),$$

where the covariance matrix $\Sigma(\mathbf{s}_i, \mathbf{s}_j)$ is parameterized as:

$$\Sigma(\mathbf{s}_i, \mathbf{s}_j) = \begin{cases} \kappa + \psi & i = j \\ \kappa \exp\left(\frac{||\mathbf{s}_i - \mathbf{s}_j||^2}{\phi}\right) & i \neq j \end{cases}$$

We place a uniform prior on $\boldsymbol{\beta}$ as in Example 1. We place inverse gamma (shape = 2, scale = 1) priors on κ and ψ so that the prior means for κ and ψ are 1.0 and the prior variance is infinite. We place a uniform prior on the effective range parameter ϕ with a lower bound of 0.01 and an upper bound of 5.0.

The computational complexity of this model is easily controlled by the number of locations N included in the dataset. To compare the univariate and multivariate slice samplers across a range of different complexities, we generated five synthetic datasets with 100, 200, 300, 400, and 500 locations. Each synthetic dataset was generated from model outlined in Example 2, with $\psi = \kappa = 1.0$ and $\phi = 0.2$. In Table 2, all of the reported effective sample sizes (and effective samples per second) reflect an MCMC run of $10,000$ iterations. We found this to be sufficient in most cases to meet the minimum benchmark of 1000 effective samples. As the comparison of the algorithms is based on sampling efficiency, we chose to start all samples at the true values in order to avoid issues related to choice of starting value. We will address these issues in the context of a real data set in Section 4.3.

In Table 2, we see that the multivariate slice sampler provides a significant improvement in ESS over the univariate slice sampler for the parameters κ and ψ. For instance, ESS for the 200 location simulation was increased by a factor of roughly 5, while for the 500 location case it was increased by a factor of roughly 7.5. For ϕ, the multivariate slice sampler mixes well, but its ESS is roughly the same or occasionally lower than the univariate slice ESS. We attribute this to the fact that the ϕ component of the Markov chain already mixes well in the univariate slice sampler and block updates with the slow mixing parameters (κ, ψ) may reduce the ESS of ϕ.

2.4 Motivation for Exploring Parallelism

The multivariate slice sampler described above can make large moves across the target density and explore it with reasonably efficiency. Under relatively weak conditions, Roberts and Rosenthal (1999) showed that the slice sampler is nearly always geometrically ergodic — a convergence criterion that a generic random walk sampler often fails to meet. Further, Mira and Tierney (2002) provide a sufficient condition under which they show that the slice sampler is uniformly ergodic.

However, the real challenge is to appropriately construct the slice $\mathbf{A} = \{\boldsymbol{\beta} : f(\boldsymbol{\beta}) \geq h\}$. Often, the computational challenge of evaluating all boundary points of a k-dimensional hypercube makes the multivariate slice sampler perform well below the univariate methods. For computationally expensive density functions, additional evaluations of the likelihood can add considerably to computation time. Further, in a sequential computing environment, an algorithm that requires even a few expensive likelihood evaluations quickly becomes computationally infeasible. In Section 2.3, we provided simulation results for a Gaussian process model where additional likelihood evaluations significantly slow down computation (see Example 2). However, since the likelihood evaluations needed to construct the slice $\mathbf{A}$ and test the proposals can be done independently, they can easily be parallelized. In a parallel computing environment, all of the needed likelihood evaluations can take place in little more than the time needed for a single evaluation in a serial environment and thus, greatly improve the speed of a slice sampler.

Parallel computation allows for efficient use of the multivariate slice sampler, which also has the benefit of being able to use additional likelihood evaluations to optimally inform the algorithm of the multivariate shape of the target distribution, allowing for the construction of a sampler that is better able to traverse

Table 2: Comparison of single-threaded slice samplings algorithms for κ, ψ, and ϕ from Example 2. All algorithms were run for $10,000$ iterations.

Slice Sampler Algorithm		100		200		300		400		500	
		ESS	(ES/sec)	ESS	(ES/sec)	ESS	(ES/sec)	ESS	(ES/sec)	ESS	(ES/sec)
Univariate	κ	1490	(21.21)	657	(2.07)	522	(0.59)	483	(0.29)	336	(0.10)
	ψ	1552	(22.08)	644	(2.03)	516	(0.58)	490	(0.29)	338	(0.10)
	ϕ	3127	(44.49)	3709	(11.70)	3585	(4.02)	3295	(1.96)	5559	(1.69)
Multivariate	κ	5207	(40.30)	3251	(4.80)	3051	(1.33)	3395	(0.85)	2519	(0.28)
	ψ	4782	(37.02)	3374	(4.98)	3028	(1.32)	3482	(0.87)	2576	(0.29)
	ϕ	2160	(16.72)	3157	(4.66)	3535	(1.54)	1467	(0.37)	4657	(0.52)

Number of Locations

Note: Here we compare the effective sample size(ESS) and effective samples per second(ES/sec) for the univariate and multivariate slice samplers. We see that even in analyzing the more computationally expensive linear Gaussian process model of Example 2 in a non-parallel environment, the multivariate slice sampler's ESS and ES/sec are higher than that of the univariate slice sampler.

quickly to all regions of the density. As demonstrated in Section 4, a multivariate slice sampler can more effectively utilize parallel likelihood evaluations than a univariate slice sampler. Clearly, as the dimension and hierarchical complexity of the desired model increases, the general nature of the parallel multivariate slice sampler affords us an efficient sampler, capable of providing an accurate picture of the target density in a reasonable amount of time.

3 Parallel Computation

In this section, we briefly describe two software packages which can be used to parallelize MCMC algorithms. These two packages, OpenMP and CUDA, are only a small sample of a number of packages available for parallel computing.

3.1 OpenMP

Open Multi-Processing (OpenMP) is a specification that has the primary purpose of providing a generic interface to support shared memory multiprocessing (cf. Chandra et al., 2001). For the purpose of statistical computation, the power of OpenMP lies in its ease of implementation. The addition of a single compiler directive will transform a non-threaded "for loop" and separate it across several CPU cores within the same system (See Listing 1). The major limitation of OpenMP is not in the software specification, but in the hardware for which it is designed. Current multi-core, multi-processor shared memory systems can scale with the dimension of a given problem but only at a huge expense. Most multi-core, multi-processor systems have only a dozen or so CPUs which greatly limits the practical speedup one can achieve. Even with these limitations, OpenMP lends utility to problems of moderate size due to the incredible ease of implementation (a single line of code) and a reasonable improvement in speed attained through parallel computation.

Listing 1: Loop toevaluate likelihoods with OpenMP compiler directive

```
#pragma omp parallel for shared(parameterSets,logLikelihood)
    for ( int i=0; i < nSets; i++ )
    {
        logLikelihood[i] = logDensity(parameterSets[i]);
    }
```

OpenMP is designed for multithreaded processing within a single computer (with one or more multicore chips). It does not support communication between computers. A different specification known as the Message Passing Interface (MPI), is designed to overcome this limitation (cf. Passing, 2008). MPI allows for processing with a cluster of computers. Parallel implementations of simple matrix algorithms (multiplication, Cholesky decomposition, etc.) are available in PLAPACK (cf. Alpatov et al., 1997) using MPI and ScaLAPACK (cf. Blackford et al., 1996) using OpenMP or MPI. Yan et al. (2007) demonstrate that PLAPACK based block matrix algorithms provide similar factors of improvement to the OpenMP results described in this article. As the focus of this paper is primarily on software which runs on a single physical machine (with a few processors / cores), we have not run simulations which use MPI. However, in the future, we envision combining these technologies where MPI is used to distribute likelihood evaluations across several computers, each of which is equipped with multiple graphics cards. In this setting OpenMP and CUDA, discussed in Section 3.2, could be used cooperatively to evaluate a single likelihood function within each machine in the cluster using multiple processors and multiple graphics cards.

3.2 CUDA

Compute Unified Device Architecture (CUDA), introduced in 2006, is a C/C++ interface to the vectorized processing available on a graphics processing unit (GPU) (cf. CUDA, 2009; Garland et al., 2008, and the references therein). A GPU differs significantly from a traditional CPU in a few key ways. First, creating and destroying threads on a CPU is very expensive, often requiring thousands of cycles or more, but on a GPU, one can create several thousand threads in less than ten clock cycles. The second key difference is that on a CPU most instructions work with a single set of operands: adding two numbers, multiplying two numbers, etc. On a GPU, all instructions use vectors of operands - they add two vectors, multiply two vectors, etc. This paradigm is often referred to as "Single Instruction, Multiple Data" (SIMD). These two differences allow a program which could only use a few CPU threads efficiently to instead utilize $30,000$ or more GPU threads and reach a level of parallelism which was previously unattainable.

We now provide a brief overview of the relevant terminology and concepts. A set of instructions which CUDA executes on a GPU is known as a kernel. When a kernel is launched from a parent CPU program, in addition to functional inputs (e.g. parameter values), the calling function must specify the number of independent blocks of parameters to process and the number of threads that will work cooperatively on each block. In the context of the multivariate slice sampler, we use the kernel function to calculate the likelihood at a given location. The number of blocks will equal the number of parameter sets to evaluate. In our implementation, the number of blocks equals the number of hypercube vertices when constructing the slice, and it equals the number of simultaneous proposals to evaluate once the approximate slice is constructed. The number of threads assigned to each block will vary based on the size and complexity of the likelihood calculation. Clearly, there is also a tradeoff between throughput and computational speed. If more threads are assigned to a block, then presuming they can be utilized, it will complete faster. However, if less threads are assigned per block, then more blocks can be processed simultaneously. The GPU used in Section 4, a GTX 280, is organized into 30 multiprocessors, each of which is capable of executing 1024 threads simultaneously (512 per block) and a maximum of eight blocks. At full capacity, a single GTX 280 can simultaneously execute anywhere from 60 blocks with 512 threads each to 240 blocks with 128 threads each. In Examples 2 and 3 in particular, we never evaluated more than 30 blocks at once; hence, to maximize both throughput and the usage of the GPU, we allocated the maximal number of threads (512) per block.

While the syntax of the GPU kernel appears nearly identical to standard C / C++, there are two major differences. First, one must read the kernel syntax assuming that 512 threads are simultaneously executing line by line without constraints on the ordering (of the threads) unless explicitly enforced. Second, memory locality is crucial to efficient kernel execution. Hence, variable declarations, unless specifically flagged by CUDA keywords, are treated as local to the thread. So a simple declaration of "**float** temp;" actually represents a unique register for each thread. In a traditional CPU, this would be stored as a vector, but here each thread maintains only its own element. See Appendix A for further details and a description of the parallel implementation of forward and backward substitution steps of Algorithm 4. The manuscript by Lee et al. (2009) also provides an excellent introduction to general purpose statistical computation using CUDA.

In Section 4 we return to the Gaussian process example from Section 2.3. OpenMP proves to be quite simple to implement and tune. The CUDA based sampler requires more tuning to achieve optimal results but this is mostly because with greater than 30,000 threads available we chose to parallelize the matrix computations in addition to batching the likelihood evaluations. Both OpenMP and CUDA show utility in parallelizing the multivariate slice sampler — OpenMP for its ease of use and CUDA for its immense processing power.

4 A Parallel Implementation of Multivariate Slice Sampling

In this section, we return to the spatial Gaussian process model presented in Section 2.3. As the multivariate slice sampler involves multiple likelihood evaluations at each iteration of the algorithm, we investigate running the parallel multivariate slice sampler using both OpenMP and CUDA on the datasets generated in Section 2.3. We outline the tuning and implementation details for the OpenMP and CUDA based samplers and compare the results of the parallel samplers with the univariate slice sampler. Then we turn to the analysis of a real dataset. As we show in the sections that follow, the results simultaneously demonstrate both the utility of the multivariate slice sampler and the improvement in speed gained through parallelization.

4.1 Implementation Details

To parallelize the multivariate slice sampler for Examples 2 and 3, we focused on three portions of the algorithm: the slice construction, the proposals, and the likelihood function itself. When constructing the approximate slice $\mathbf{A}$ using a three dimensional hypercube (shown in Algorithm 2), the likelihood at each of the 8 hypercube vertices can be evaluated independently. After evaluating the likelihoods, if there is a need to step out in any dimension, the re-evaluation and updating of the hypercube vertices is done in parallel. When proposing a new location (shown in Algorithm 3), we chose to use a simple rejection sampler and evaluate batches of proposals of size K in parallel. The first proposal to fall within the target slice is then accepted and the parameters are updated. For the OpenMP sampler, we set the proposal batch size equal to the number of threads (either three or four). For the CUDA sampler, we set the proposal batch size equal to 30 (the number of blocks allowed for maximal throughput). Finally, in the CUDA-based multivariate slice sampler, we also chose to parallelize the matrix operations in the likelihood function itself.

4.1.1 OpenMP

As noted in Section 2.4, to implement the parallel multivariate slice sampler using OpenMP, there are two places where the algorithm must evaluate the log likelihood for a list of parameters: (1) testing each hypercube vertex (in Algorithm 2) and (2) evaluating sets of proposed parameter values (in Algorithm 3).

Definitions

$\boldsymbol{\beta}$	the current parameter value ($\boldsymbol{\beta} \in \mathbb{R}^k$)
ω	the initial interval widths for constructing a hyperrectangle in k dimensions.
$\mathbf{L}$	Lower bounds of hypercube ($\mathbf{L} \in \mathbb{R}^k$)
$\mathbf{U}$	Upper bounds of hypercube ($\mathbf{U} \in \mathbb{R}^k$)
f	function proportional to the full conditional distribution for $\boldsymbol{\beta}$.

Algorithm

```
Sample $h \sim \text{Uniform}\{\, 0,\ f(\,\boldsymbol{\beta}\,)\,\}$
For $j \in \{1 \ldots k\}$
  $u \sim \text{Uniform}(0, 1)$
  $\mathbf{L}_j = \boldsymbol{\beta}_j - \omega_j * u$
  $\mathbf{U}_j = \mathbf{L}_j + \omega_j$
Do
  Construct list of $k$ dimensional hypercube
  vertices from $\mathbf{L}$ and $\mathbf{U}$
  Evaluate $2^k$ vertices in parallel
  Test each hypercube vertex, if $(h < f(vertex))$
    extend $\mathbf{L}$ and/or $\mathbf{U}$ by $\omega$ as appropriate
While ($\mathbf{L}$ or $\mathbf{U}$ changed)
```

Algorithm 2: A parallel step-out procedure for constructing an approximate multivariate slice for the parameter $\boldsymbol{\beta}$.

Definitions

$\boldsymbol{\beta}$	the current parameter value ($\boldsymbol{\beta} \in \mathbb{R}^k$)
$\mathbf{L}$	Lower bounds of hypercube ($\mathbf{L} \in \mathbb{R}^k$)
$\mathbf{U}$	Upper bounds of hypercube ($\mathbf{U} \in \mathbb{R}^k$)
f	function proportional to the full conditional distribution for $\boldsymbol{\beta}$.

Algorithm

```
Repeat until proposal not rejected
  Draw batch of $K$ proposals uniformly from
  hypercube bounded by $\mathbf{L}$ and $\mathbf{U}$
  Evaluate $K$ proposals in parallel
  On first proposal $m$ that $(h < f(\text{proposal}[m]))$
    Update $\boldsymbol{\beta} = \text{proposal}[m]$ and stop
```

Algorithm 3: A batched parallel rejection sampler for the multivariate slice sampler update of the parameter $\boldsymbol{\beta}$.

In Listing 1, we saw that adding a single OpenMP compiler directive transformed the serial evaluation of these parameter sets into a simultaneous evaluation across the available processors.

Before comparing the relative efficiency of the parallel multivariate slice sampler using OpenMP with the other samplers, we first had to run a short tuning study to determine the optimal number of threads. The optimal number of threads depends on the number of locations (N) in the model, as the matrix operations involved in each likelihood evaluation, such as Cholesky decompositions, are of order N^3. Our simulations were performed on a dual quad-core E5430 Xeon system. We found that for 100 to 300 locations, three OpenMP threads was optimal. For 400 to 500 locations, four OpenMP threads was optimal. Clearly the geometry and configuration of the processor hardware will have an impact on the speed of execution and the realized speedup (or lag) of using additional processor cores.

The results provided in Section 4.2 demonstrate that parallelizing the multivariate slice sampler with OpenMP attains only a marginal speedup. This is due to the moderately expensive Gaussian process model likelihood function for these relatively small choices of model size. We chose 500 locations as an upper bound due the hardware register constraints within the graphics card. As the location count increases the workload will increase. Hence, the OpenMP threads will be better utilized and that will yield a higher relative speedup. Finally, although the relative speedup using OpenMP may not be large, the burden of adding two additional lines of code should be weighed against the 40% performance improvement.

4.1.2 CUDA

As with the OpenMP based sampler, we parallelize the evaluation of the hypercube vertices in Algorithm 2 and also parallelize the batched proposal evaluations in Algorithm 3. Taking advantage of about $30,000$ threads in the CUDA based sampler, we also parallelize the matrix computations for each likelihood evaluation. The evaluation of the full conditional distribution for κ, ψ, and ϕ, given below in Equation 1, can be decomposed into five major steps, outlined in Algorithm 4. Note that the bulk of the computational expense occurs in the lower triangular Cholesky factorization in step two.

$$\pi(\kappa,\psi,\phi|-) \propto |\Sigma|^{-1/2} \exp\left\{-\frac{1}{2}\left(Y - X\boldsymbol{\beta}\right)\Sigma^{-1}\left(Y - X\boldsymbol{\beta}\right)\right\}\kappa^{-(\alpha_\kappa+1)}\exp\left(\frac{-\beta_\kappa}{\kappa}\right)\psi^{-(\alpha_\psi+1)}\exp\left(\frac{-\beta_\psi}{\psi}\right). \tag{1}$$

1. Construct the covariance matrix Σ from the given κ, ψ, and ϕ values and the list of locations s_i. Recall that Σ is defined as follows:

$$\Sigma(\mathbf{s}_i,\mathbf{s}_j) = \begin{cases} \kappa+\psi & i=j \\ \kappa\exp\left(\frac{||\mathbf{s}_i-\mathbf{s}_j||^2}{\phi}\right) & i\neq j \end{cases}$$

2. Compute the lower triangular Cholesky factorization of Σ.
3. Solve for $\Sigma^{-1}\left(Y - X\boldsymbol{\beta}\right)$ via forward and back substitution.
4. Compute the dot product of $(Y - X\boldsymbol{\beta})$ with the results of step #3.
5. Compute the determinant of Σ and the other remaining terms.

Algorithm 4: An algorithmic representation for the evaluation of the full conditional distribution of κ, ψ, and ϕ in Examples 2 and 3. Each of these steps is parallelized as described in Section 4.1.2.

The evaluation of the full conditional distribution for κ, ψ, and ϕ using the CUDA kernel can be parallelized at each step of Algorithm 4. In step one, the construction of Σ is done column by column. By virtue of having more threads (512) than locations (at most 500), each thread is assigned to compute a single element in the current column of Σ. Then that column of Σ is immediately used to compute the lower triangular Cholesky factorization via a standard block decomposition (cf. Golub and Van Loan (1996)). As a result, storage of more than one column of Σ is not required, which is pivotal because, as noted in Section 3.2, efficient memory usage is one of the most important considerations in programming for a GPU.

Similar memory optimizations can be made in steps three, four, and five. In step three of Algorithm 4, both the forward and back substitution are performed using one column and one row, at a time and again, each element in the vector is given to a separate thread. During the back substitution, each thread simultaneously computes its contribution to the dot product for step four and the diagonal element's contribution to the determinant in step five. As a result, upon completion of the back substitution, steps four and five are also finished, leaving only the contribution of the priors to be computed. The full GPU kernel code written for this section is available for download from the author's website (See Appendix A).

There are other hardware constraints (shared memory capacity, memory transfer penalties, etc.) which are beyond the scope of the discussion here, but were taken into account in designing the CUDA likelihood evaluation. As mentioned in the previous section, the current GPU hardware and software (CUDA 2.2) limits the maximum thread block size to 512 threads. As a result, we did not test models with greater than 500 locations. As new hardware is developed which supports larger block sizes, we hope to attain better improvements for larger models. Future research will also focus on utilizing multiple GPUs at a time. Current technology limits us to using a maximum of four GPUs simultaneously, but extending to this environment would allow for up to 960 simultaneous likelihood evaluations.

4.2 Summary of Simulation Study

Results of a simulation based on Example 2 using the univariate slice sampler, the single-threaded multivariate slice sampler, the optimal OpenMP multivariate slice sampler, and the optimal CUDA multivariate slice sampler are summarized in Table 3. We note that for all of the samplers shown, the ES/sec decreases as the number of locations increase. For example, the ES/sec of κ and ψ for the 100 location model is roughly ten times that of the 200 location model for all of the samplers. The dependence of κ and ψ is evident in Table 3. In simulating these datasets, we selected a range parameter, ϕ, of 0.2.

As was discussed in Section 2.3, even the relatively slow single CPU multivariate slice sampler outperforms the univariate slice sampler in ES/sec for κ and ψ mainly because of a much larger ESS. However, the ES/sec for ϕ for the single CPU multivariate slice sampler is much lower than that of the univariate slice sampler. Here again, this is due to the moderate spatial dependence; blocking the update of ϕ with κ and ψ in a three dimensional sampler results in a sampler where the ability to mix across the distribution of ϕ is greatly hampered by the strong dependence between κ and ψ. However, the ES/sec of κ and ψ is still clearly the limiting factor.

Practically speaking, it is the smallest ES/sec which determines the minimal run time of a given algorithm. In other words, to produce at least $10,000$ effective samples for κ, ψ, and ϕ using the 300 location model with the univariate slice sampler would require $10000/0.58 = 17,240$ seconds (roughly 5 hours). To generate the same $10,000$ effective samples for κ, ψ, and ϕ using the single CPU multivariate slice sampler would require $1000/1.32 = 7,575$ seconds (2.1 hours). However, when we examine the 400 location model, this factor of two improvement nearly vanishes because the ratio of the smallest multivariate ES/sec (ϕ ES/sec $= 0.37$) to the smallest univariate ES/sec (κ ES/sec $= 0.29$) is only 1.27. Hence, the net gain of using a more complicated algorithm is only an improvement of 27%. However, the picture changes quite dramatically when we use the parallel samplers.

The OpenMP based sampler shows a reasonable 40% to 50% improvement in ES/sec on top of the improvement that the multivariate slice sampler makes in ES/sec over the univariate slice sampler. The largest improvement occurs in the 300 location model, but even in this case, the ϕ ES/sec is still only 57% of the univariate slice sampler's ES/sec. However, the ratio of the smallest OpenMP multivariate ES/sec (ϕ ES/sec $= 0.52$) to the smallest univariate ES/sec (κ ES/sec $= 0.29$) is 1.79. Hence, the speedup of the algorithm directly translates to shorter run times.

The CUDA-based multivariate slice sampler provides an even more compelling case for the parallel multivariate slice sampler. In Table 3, we note that the speedup factor increases significantly as the size of the model increases. In the 500 location model, the CUDA based multivariate slice sampler is 5.4 times faster than the single CPU multivariate slice sampler. When we examine the sampling times, the univariate slice sampler would require $10000/.10 = 100,000$ seconds (roughly 28 hours) to produce $10,000$ effective samples for all parameters; whereas, the CUDA based multivariate slice sampler would require only $10000/1.51 = 6622$ seconds (or roughly 1.8 hours). Hence, the CUDA based multivariate slice sampler improves upon the efficiency of the univariate slice sampler by a factor of roughly 15.

Table 3: Comparison of effective samples per second and relative speedup for single and multi-threaded slice samplings algorithms for κ, ψ, and ϕ from Example 2. All algorithms were run for $10,000$ iterations.

Slice Sampler Algorithm		Number of Locations									
		100		200		300		400		500	
Univariate Single CPU	κ	21.21		2.07		0.59		0.29		0.10	
	ψ	22.08		2.03		0.58		0.29		0.10	
	ϕ	44.49		11.70		4.02		1.96		1.69	
Multivariate Single CPU	κ	40.30	(1.90)	4.80	(2.31)	1.33	(2.27)	0.85	(2.96)	0.28	(2.73)
	ψ	37.02	(1.68)	4.98	(2.45)	1.32	(2.28)	0.87	(2.99)	0.29	(2.78)
	ϕ	16.72	(0.38)	4.66	(0.40)	1.54	(0.38)	0.37	(0.19)	0.52	(0.31)
Multivariate OpenMP	κ	56.73	(2.67)	6.83	(3.30)	1.96	(3.34)	1.19	(4.14)	0.37	(3.59)
	ψ	52.10	(2.36)	7.09	(3.49)	1.94	(3.35)	1.22	(4.20)	0.38	(3.65)
	ϕ	23.53	(0.53)	6.64	(0.57)	2.27	(0.56)	0.52	(0.26)	0.68	(0.40)
Multivariate CUDA	κ	116.77	(5.51)	19.21	(9.26)	6.16	(10.51)	3.72	(12.94)	1.51	(14.78)
	ψ	119.83	(5.43)	19.19	(9.44)	6.31	(10.88)	3.90	(13.38)	1.53	(14.84)
	ϕ	64.56	(1.45)	17.57	(1.50)	8.61	(2.14)	1.59	(0.81)	2.88	(1.70)

ES/sec (speedup)

Note: Table 3 uses the univariate slice sampler's run time (single-threaded CPU-based) as the baseline for determining algorithmic speedups shown above for Example 2 (cf. Sections 2.3 and 4)

After examining the results of Table 3, we investigated a two dimensional multivariate slice sampler for only κ and ψ, but found that the added cost of a univariate update for ϕ pulled the performance of the multivariate sampler down dramatically, especially in the CUDA based implementation. This highlights the fact that there is a negligible difference between running one, two, three, four, or five dimension multivariate slice samplers on the GPU because all will fit within the 60 block maximum (to achieve peak performance). Further, in situations which require higher dimensional slice samplers (up to nine dimensions), four graphics cards could be combined to compute all needed likelihood evaluations simultaneously.

Though we have not included a study of parallelized univariate sampling, we would like to point out that while it would be beneficial for both multivariate and univariate slice sampling to speed up or parallelize each individual likelihood calculation, the univariate slice sampler would not benefit nearly as much from the two parallelizations in Algorithms 2 and 3. As the univariate slice sampler usually requires only three or four likelihood evaluations to construct the approximate slice and test a few proposals, a speedup factor of two or three gained by evaluating these likelihoods in parallel would only place the ES/sec on par with the single CPU multivariate slice sampler.

4.3 Application to Surface Temperature Data

In this section, we apply the linear Gaussian process model from Example 2 to the analysis of the mean surface temperature over the month of January, 1995 on a 24×21 grid covering Central America. These data were obtained from the NASA Langley Research Center Atmospheric Science Data Center.

Example 3 (ASDC Surface Temperature Dataset). We wish to model the mean surface temperature $Y(\mathbf{s}_i)$ measured at 500 locations s_i with a set of covariates $X(\mathbf{s}_i)$ which includes an intercept and the latitude and longitude of each grid point. We assume the same linear Gaussian process model studied in Example 2. We complete the Bayesian model specification by placing inverse gamma (shape = 2, scale = 1) priors on κ and ψ and a uniform prior on ϕ with a lower bound of 0.1 and an upper bound of 50.0 (which is larger than the prior in the earlier example due to the increased separation in the grid points), and a flat prior on β.

For the analysis of the surface temperature data, we tried several different starting values and found that our results were insensitive to initial conditions. The initial interval widths for all of the samplers were tuned in order to maximize ES/sec by using a large grid search. The multivariate slice samplers converged to the support of the posterior densities within three samples. The univariate slice sampler took slightly longer, but even with extreme starting values it never took more than 30 samples to reach the support of the posterior density. We ran each sampler for $100,000$ iterations.

In examining the posterior parameter estimates for this dataset, we have $\mu_\kappa = 55$, $\mu_\phi = 49$, and $\mu_\psi = 0.2$. This dataset exhibits a much stronger spatial dependence than the simulated data set in Section 4. In Table 4 we see that the parameters κ and ϕ are strongly correlated whereas κ and ψ were strongly correlated in the simulation study. We find, in comparing the univariate slice sampler to the parallelized multivariate slice sampler that many of the same relationships from simulation study hold here as well. For example, we see that the ESS for κ and ϕ of the multivariate samplers is more than ten times the ESS for the univariate sampler. And though the multivariate slice sampler is much more computationally expensive, the parallelization through OpenMP and especially through CUDA is sufficient to mitigate the increased computational burden. By comparing the minimum ES/sec, we see that the minimum ES/sec of the CUDA multivariate slice sampler (ES/sec = 2.42 for κ) is roughly 13 times the minimum ES/sec of the univariate slice sampler (ES/sec = 0.18 for ϕ). As in the simulation study described in Section 4, the CUDA multivariate slice sampler provides a significant improvement over the univariate slice sampler.

5 Discussion

We have examined the performance of the univariate and multivariate slice samplers within the context of two synthetic examples and one real data example involving a spatial model for surface temperatures. In our simulations, we found that the multivariate slice sampler was much more efficient than the univariate methods when the posterior distribution is highly correlated along one or more dimensions. However, the multivariate slice sampler is computationally expensive per iteration, especially when likelihood evaluations are expensive. Even in the context of a more expensive linear Gaussian process model, we found that the multivariate slice sampler's excellent mixing properties resulted in greater efficiency when compared to the univariate slice sampler.

Table 4: Comparison of effective sample size (ESS), effective samples per second(ES/sec), and relative speedup of ES/sec for κ, ψ, and ϕ from Example 3. All algorithms were run for $200,000$ iterations, but the first $100,000$ were discarded to allow for sampler burnin.

	ψ			κ			ϕ		
Algorithms	ESS	ES/sec	ES/sec Speedup	ESS	ES/sec	ES/sec Speedup	ESS	ES/sec	ES/sec Speedup
Univariate Slice Sampler	65254	2.89		4284	0.19		4147	0.18	
Multivariate Slice Sampler (using OpenMP)	54640	1.86	(0.64)	52324	1.79	(9.42)	55306	1.89	(10.50)
Multivariate Slice Sampler (using CUDA)	54794	2.59	(0.90)	51177	2.42	(12.74)	54229	2.57	(14.28)

Note: Table 4 uses the univariate slice sampler's run time as the baseline for determining algorithmic speedups shown above for Example 3 (cf. Section 4.3)

We also noted that the likelihood evaluations utilized in constructing an approximate slice could be independently evaluated, allowing hypercube vertices and proposals to be evaluated in parallel. We investigated two different threading implementations of this approach. The OpenMP implementation requires minimal modifications to the code, but only attained a 40% to 50% improvement over the single-threaded multivariate slice sampler. In contrast, the CUDA implementation was five times faster than the single-threaded CPU-based implementation. When combining this speedup with the already superior ESS of the multivariate algorithm, we found that the CUDA solution yielded a sampler which was 14 times more efficient than the single-threaded univariate slice sampler for both the simulation study in Section 4 and the empirical analysis of surface temperatures in Section 4.3. We note, however, that the examples investigated utilize three dimensional slice samplers. Clearly as the dimensionality of the target distribution increases, the hypercube approximate slice will become less and less efficient. As the multivariate slice sampler employed here uses a the rejection sampler to generate from the approximate slice, it will perform poorly in high dimensions and more intelligent methods using, for example, a numerical gradient to optimally shrink the hypercube should be explored. However, even if the acceptance rate were to fall to 0.5%, a combination of four graphics cards could evaluate 240 proposals simultaneously and would be expected to generate an accepted proposal within the first batch ($240 \times 0.005 = 1.2$). Hence, the parallelized multivariate slice sampler can utilize more and more hardware to overcome the significant computational burden for problems of moderate dimension.

Our results suggest that parallel multivariate slice sampling can offer substantial gains in efficiency over univariate slice samplers which, in turn, are often known to be more efficient than standard MCMC algorithms like the Metropolis-Hastings random walk. We believe this work demonstrates how massively parallel computing may be used to greatly improve the efficiency of even inherently sequential MCMC algorithms.

References

Agarwal, D. K. and Gelfand, A. E. (2005). Slice sampling for simulation based fitting of spatial data models. *Statistics and Computing*, 15(1):61–69.

Alpatov, P., Baker, G., Edwards, C., Gunnels, J., Morrow, G., Overfelt, J., and jye J. Wu, Y. (1997). Plapack: Parallel linear algebra package.

Andrieu, C., de Freitas, N., Doucet, A., and Jordan, M. I. (2003). An Introduction to MCMC for Machine Learning. *Machine Learning*, 50(1):5–43.

Banerjee, S., Carlin, B., and Gelfand, A. (2004). *Hierarchical modeling and analysis for spatial data*. Chapman & Hall Ltd.

Blackford, L. S., Choi, J., Cleary, A., Petitet, A., Whaley, R. C., Demmel, J., Dhillon, I., Stanley, K., Dongarra, J., Hammarling, S., Henry, G., and Walker, D. (1996). Scalapack: a portable linear algebra library for distributed memory computers - design issues and performance. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 5, Washington, DC, USA. IEEE Computer Society.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. (2001). *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Chib, S. and Carlin, B. P. (1999). On MCMC sampling in hierarchical longitudinal models. *Statistics and Computing*, 9(1):17–26.

Cressie, N. A. C. (1993). *Statistics for Spatial Data*. Wiley Series in Probability and Statistics. Wiley-Interscience, New York, 2nd. edition.

CUDA (2009). *NVIDIA Compute Unified Device Architecture - Programming Guide - Version 2.2.1*. NVIDIA Corporation, `http://developer.download.nvidia.com/compute/cuda/2_21/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.1.pdf`.

Damien, P., Wakefield, J., and Walker, S. (1999). Gibbs sampling for Bayesian non-conjugate and hierarchical models by using auxiliary variables. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, 61(2):331–344.

Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., and Volkov, V. (2008). Parallel computing experiences with cuda. *Micro, IEEE*, 28(4):13–27.

Geyer, C. J. (1992). Practical Markov Chain Monte Carlo. *Statistical Science*, 7(4):473–483.

Gilks, W. R. and Roberts, G. (1996). Strategies for improving MCMC. In Gilks, W. R., Richardson, S., and Spiegelhalter, D. J., editors, *Markov Chain Monte Carlo in Practice*, pages 89–114. London: Chapman & Hall/CRC.

Golub, G. H. and Van Loan, C. F. (1996). *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)*. The Johns Hopkins University Press.

Jiang, R., Zeng, F., Zhang, W., Wu, X., and Yu, Z. (2009). Accelerating genome-wide association studies using cuda compatible graphics processing units. In *Bioinformatics, Systems Biology and Intelligent Computing, 2009. IJCBS '09. International Joint Conference on*, pages 70–76.

Kass, R. E., Carlin, B. P., Gelman, A., and Neal, R. M. (1998). Markov Chain Monte Carlo in Practice: A Roundtable Discussion. *The American Statistician*, 52(2):93–100.

Kinney, S. K. and Dunson, D. B. (2007). Fixed and random effects selection in linear and logistic models. *Biometrics*, 63:690–698(9).

Kovac, K. (2005). Machine Learning for Bayesian Neural Networks. Master of science, University of Toronto.

Lee, A., Yau, C., Giles, M. B., Doucet, A., and Holmes, C. C. (2009). On the utility of graphics cards to perform massively parallel simulation of advanced monte carlo methods. *Statistics and Computing*, .(.):. Submitted for Publication July 2009.

Lewis, P. O., Holder, M. T., and Holsinger, K. E. (2005). Polytomies and Bayesian Phylogenetic Inference. *Systematic Biology*, 54(2):241–253.

Liu, J. S., Wong, W. H., and Kong, A. (1994). Covariance Structure of the Gibbs Sampler with Applications to the Comparisons of Estimators and Augmentation Schemes. *Biometrika*, 81(1):27–40.

Mackay, D. J. C. (2002). *Information Theory, Inference & Learning Algorithms*. Cambridge University Press.

Manavski, S. and Valle, G. (2008). Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10.

Mira, A. and Roberts, G. O. (2003). [Slice Sampling]: Discussion. *The Annals of Statistics*, 31(3):748–753.

Mira, A. and Tierney, L. (2002). Efficiency and convergence properties of slice samplers. *Scandinavian Journal of Statistics*, 29:1–12(12).

Neal, R. M. (1997). Markov Chain Monte Carlo Methods based on 'Slicing' the Density Function. Technical report, Department of Statistics, University of Toronto.

Neal, R. M. (2003a). Slice Sampling. *The Annals of Statistics*, 31(3):705–741.

Neal, R. M. (2003b). [Slice Sampling]: Rejoinder. *The Annals of Statistics*, 31(3):758–767.

Nott, D. J. and Leonte, D. (2004). Sampling schemes for Bayesian variable selection in generalized linear models. *Journal of Computational and Graphical Statistics*, 13(2):362–382.

Passing, M. (2008). Mpi: A message-passing interface standard. Message Passing Interface Forum. Version 2.1.

Roberts, G. O. and Rosenthal, J. S. (1999). Convergence of slice sampler Markov chains. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61:643–660(18).

Roberts, G. O. and Rosenthal, J. S. (2002). The Polar Slice Sampler. *Stochastic Models*, 18(2):257–280.

Roberts, G. O. and Sahu, S. K. (1997). Updating Schemes, Correlation Structure, Blocking and Parameterization for the Gibbs Sampler. *Journal of the Royal Statistical Society: Series B (Methodological)*, 59(2):291–317.

Rosenthal, J. S. (2000). Parallel computing and Monte Carlo algorithms. *Far East Journal of Theoretical Statistics*, 4:207–236.

Shahbaba, B. and Neal, R. (2006). Gene function classification using Bayesian models with hierarchy-based priors. *BMC Bioinformatics*, 7(1):448.

Sinnott-Armstrong, N., Greene, C., Cancare, F., and Moore, J. (2009). Accelerating epistasis analysis in human genetics with consumer graphics hardware. *BMC Research Notes*, 2(1):149.

Suchard, M. A. and Rambaut, A. (2009). Many-core algorithms for statistical phylogenetics. *Bioinformatics*, 25(11):1370–1376.

Sun, S., Greenwood, C. M., and Neal, R. M. (2007). Haplotype inference using a Bayesian Hidden Markov model. *Genetic Epidemiology*, 31(8):937–948.

Whiley, M. and Wilson, S. P. (2004). Parallel algorithms for Markov Chain Monte Carlo methods in latent spatial Gaussian models. *Statistics and Computing*, 14(3):171–179.

Wilkinson, D. J. (2005). Parallel bayesian computation. In Kontoghiorghes, editor, *Handbook of Parallel Computing and Statistics*, pages 481–512. Marcel Dekker/CRC Press.

Yan, J., Cowles, M. K., Wang, S., and Armstrong, M. P. (2007). Parallelizing MCMC for Bayesian spatiotemporal geostatistical models. *Statistics and Computing*, 17(4):323–335.

Table 5: List of (1) Memory Scope, (2) Memory Size (for GTX 280 graphics card), (3) Added Latency (in clock cycles) per instruction for memory accesses

Type	Scope	Size	Additional Latency
Register File	Thread	16 KB^1	0
Shared	Block	16 KB^1	0^2
Constant^3	Kernel	64 KB	0
Device	Device	1 - 4 GB	400 - 600
System	System	> 1 GB	> 1000

^{1} Per Multiprocessor ^{2} Out of order access incurs additional latency ^{3} Read-Only

Appendix A

We describe here the parallel implementation of the forward and backward substitution steps of Algorithm 4. As mentioned in Section 3.2, there are two primary considerations for designing GPU algorithms and specifically CUDA kernels. First, in contrast to a serial algorithm, a GPU kernel will have several threads simultaneously executing the instructions. As we will see in the description of Listings 2 and 3 below, unless ordering is specifically enforced, one must carefully consider thread cooperation and interdependence. Second, memory locality must be strictly specified by the kernel author. On a CPU, the cache hierarchies (L1, L2, L3, RAM, etc.) are managed transparently by the hardware and the software compiler. However, as of the current CUDA specification (2.2), the kernel author must explicitly define where a variable should be kept in the memory hierarchy, as well as directly calling the appropriate load and store instructions. This greater granularity of control allows a kernel to acheive maximal performance, but also requires careful planning and consideration as the fastest memory spaces are quite limited.

In Table 5, we list the main five memory types used in GPU kernels (For details on Texture Memory, see CUDA, 2009). Each CUDA thread is given access to a super-fast register file which has no latency. However, this register file must be spread across all blocks currently executing on a multiprocessor. If a kernel uses too many registers per thread, this will limit the number of blocks which can execute simultaneously. In Examples 2 and 3 we assigned 512 threads per block. Had each thread utilized more than 16 single precision registers, the maximum number of simultaneously executing blocks would be cut in half. Shared memory provides a means for threads to communicate with each other with virtually no latency, but as with the register file, the size is very limited. Constant memory is four times the size of the register file and is equally fast. However, constant memory is read-only for the life of the kernel. Device memory is much higher latency, but also has a much larger capacity. The GTX 280 card we used had 1 GB of device memory, whereas newer hardware offers up to 4 GB. System memory, more commonly referred to as system Random Access Memory (RAM) is controlled by the CPU and used to pass data from the CPU to GPU. As system RAM is not physically located on the graphics card, it has a much higher latency which is very dependent on the system configuration. Clearly, careful planning and resource allocation are required for a kernel to acheive maximum performance. However, it is worth noting that nVidia provides a CUDA Occupancy calculator which, when given the shared memory requirements and register usage of the algorithm, will provide the optimal thread count to maximize throughput.

As shown in Listings 2 and 3, the syntax of a CUDA kernel appears nearly identical to standard C / C++, save for the addition of a few keywords, built-in variables, and functions. For example, the keyword "**__device__**" signfies that this function exists only on the GPU. The built-in variable "threadIdx.x" uniquely identifies each thread's index within the block of threads. Lastly, the function "__syncthreads()"

acts as a barrier to the parallel execution and forces a thread to wait until all threads have reached that function call. As noted in Section 3.2, there is also an important symantic difference in line 17 of Listing 2. When variables within a CUDA kernel are declared without a modifier, they are treated as registers. Hence, each thread will have its own unique register labelled “threadSub” in which it stores the partial results of foward and backward substitution. In code written for a CPU, these elements would be stored as a vector and declared as “**float** threadSub[nLocations];”.

The “forward substitution” algorithm is used to solve a lower triangular system $\mathbf{L}x = b$ where x and b are both vectors. The standard algorithm is written as (cf. Golub and Van Loan, 1996):

$$x_i = \frac{1}{l_{ii}} \left(b_i - \sum_{j=1}^{i-1} l_{ij} x_j \right)$$

In Examples 2 and 3, forward substitution is used to solve the equation $\mathbf{LA} = (\mathbf{X} - \mathbf{Y}\beta)$ where $\mathbf{L}$ is the lower triangular Cholesky Decomposition, $\Sigma \equiv \mathbf{LL}^T$. A CUDA kernel which performs forward substitution is given in Listing 2. Note how the diagonal elements are handled separately by a single thread (which contains the appropriate partial result) while all other threads wait at a __syncthreads() instruction.

Listing 2: CUDA code to perform Forward Substitution, Kernel continued in Listing 3

```
    __device__ void ForwardBackwardSubstitute
    (
       float* tempCholesky,     /*  Current Cholesky Decomposition  */
       float* cholDiag,         /*  Diagonal of Cholesky Matrix  */
  5    float* yMinusXBeta,      /*  (Y - X * Beta) Precomputed  */
       float* llProposal        /*  Output - Log Likelihood  */
    )
    {
       /*  Cholesky Decomposition Stored by Block Index  */
 10    __shared__ float* localCholesky = tempCholesky +
         blockIdx.x * nLocations * nLocations * sizeof(float);

       __shared__ float tempSubstitute[nLocations];
       __shared__ float logDensity = 0.0F;
 15
       /*  Copy (Y - X*beta) to threadSub  */
       float threadSub = yMinusXBeta[threadIdx.x];

       /*   Forward Substitution  */
 20    for ( unsigned int i=0; i < nLocations; i ++ )
       {
          if ( i == threadIdx.x )
          {
             threadSub *= cholDiag[i];
 25          tempSubstitute[i] = threadSub;
          }
          __syncthreads();

          if ( i < threadIdx.x )
 30       {
             threadSub -= tempSubstitute[i] *
          localCholesky[threadIdx.x + nLocations * i];
          }
       }
 35    __syncthreads();
```

The “backward substitution” algorithm is used to solve an upper triangular system $\mathbf{U}x = b$ where x and b are both vectors. The standard algorithm is written as (cf. Golub and Van Loan, 1996):

$$x_i = \frac{1}{u_{ii}} \left(b_i - \sum_{j=i+1}^{n} u_{ij} x_j \right)$$

In Examples 2 and 3, backward substitution is used to solve the equation $\mathbf{L}^T\mathbf{B} = \mathbf{A}$ where $\mathbf{L}$ is again the lower triangular Cholesky Decomposition, $\Sigma \equiv \mathbf{L}\mathbf{L}^T$, $\mathbf{A}$ is the partial result from forward substitution, and the result is $\mathbf{B} \equiv \Sigma^{-1} (\mathbf{X} - \mathbf{Y}\beta)$. A CUDA kernel which performs backward substitution is give in Listing 3. Note how, as in Listing 2, the diagonal elements in the backward substitution algorithm are handled separately by a single thread (which contains the appropriate partial result) while all other threads wait at a __syncthreads() instruction. In Listing 3, we also accumulate the logarithm of the determinant and the likelihood.

Listing 3: CUDA code to perform Backward Substitution, Kernel continued from Listing 2

```
    /*  Back Substitution  */
    for ( int i=nLocations-1; i >= 0; i-- )
    {
       if ( i == threadIdx.x )
       {
          threadSub *= cholDiag[i];
          tempSubstitute[i] = threadSub;

          /*  Test for Singularity  */
          if ( cholDiag[i] <= 0 )
             logDensity = -1.0e30F;

          /*  Compute Likelihood and Determinant  */
          logDensity -= 0.5F * threadSub * threadYXB - logf(cholDiag[i]);
       }
       __syncthreads();

       if ( threadIdx.x < i )
       {
          threadSub -= tempSubstitute[i] *
       localCholesky[i + nLocations * threadIdx.x];
       }
    }

    if ( 0 == threadIdx.x )
       llProposal = logDensity;
 }
```

Examples 2 and 3 used a single CUDA kernel to perform all five steps in Algorithm 4 (Cholesky decomposition, forward and backward substitution, dot product, and determinant). The kernel shown in Listings 2 and 3 are greatly simplified to increase readability. Further information on CUDA, including an extensive list of electronic courses available (for free) directly from the NVIDIA Corporation and at universities around the world is available here:

`http://www.nvidia.com/object/cuda_education.html`