MASSIVELY PARALLEL POPULATION-BASED MONTE CARLO METHODS WITH MANY-CORE PROCESSORS

Wint Pa Pa Kyaw*

Abstract

This research presents the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods. Graphics cards, containing multiple Graphics Processing Units (GPUs), are self-contained parallel computational devices that can be housed in conventional desktop and laptop computers and can be thought of as prototypes of the next generation of many-core processors. For certain classes of population-based Monte Carlo (MC) algorithms they offer massively parallel simulation, with the added advantage over conventional distributed multi-core processors that they are cheap, easily accessible, easy to maintain, easy to code, dedicated local devices with low power consumption. On a canonical set of stochastic simulation examples including population-based Markov chain Monte Carlo (MCMC) methods and Sequential Monte Carlo (SMC) methods, speedups are found from 35 to 500 fold over conventional single-threaded computer code. These findings suggest that GPUs have the potential to facilitate the growth of statistical modelling into complex data rich domains through the availability of cheap and accessible many-core computation.

Keywords: Sequential Monte Carlo, Population-Based Markov Chain Monte Carlo, General Purpose Computation on Graphics Processing Units, Many-Core Architecture, Stochastic Simulation, Parallel Processing

Introduction

This research describes the utility of graphics cards involving Graphics Processing Units (GPUs) to perform local, dedicated, massively parallel stochastic simulation. GPUs were originally developed as dedicated devices to aid in real-time graphics rendering. However recently there has been an emerging literature on their use for scientific computing as they house multicore processors. Many advanced population-based Monte Carlo (MC) algorithms are ideally suited to GPU simulation and offer significant speed up over single CPU implementation. The focus is on the parallelization of general

*. Dr., Associate Professor, Department of Computer Studies, University of Yangon

sampling methods. Moreover, this research shows how the choice of population-based MC algorithm for a particular problem can depend on whether one is running the algorithm on a GPU or a CPU.

To gain an understanding of the potential benefits to statisticians this research has investigated speedups on a canonical set of examples taken from the population-based MC literature. These include Bayesian inference for a Gaussian mixture model computed using a population-based Markov Chain Monte Carlo (MCMC) method. The idea of splitting the computational effort of parallelizable algorithms amongst processors is certainly not new to statisticians. In fact, distributed systems and clusters of computers have been around for decades. Many-core processor communication has very low latency and very high bandwidth due to high-speed memory that is shared amongst the cores. Low latency here means the time for a unit of data to be accessed or written to memory by a processor is low while high bandwidth means that the amount of data that can be sent in a unit of time is high. For many algorithms, this makes parallelization viable where it previously was not. In addition, the energy efficiency of a many-core computation compared to a single-core or distributed computation can be improved. This is because the computation can both take less time and require less overhead. Finally, these features enable the use of parallel computing for researchers outside traditional high-cost centers housing high-performance computing clusters.

The speedup is chosen to investigate for the simulation of random variates from complex distributions, a common computational task when performing inference using MC. In particular, population-based MCMC methods and SMC methods are focused on for producing random variates as these are not algorithms that typically see significant speedup on clusters due to the need for frequent, high-volume communication between computing nodes. This work focuses on the suitability of many-core computation for MC algorithms whose structure is parallel, since this is of broad theoretical interest, as opposed to a focusing on parallel computation of application-specific likelihoods.

The algorithms are implemented for the Compute Unified Device Architecture (CUDA) and make use of GPUs which support this architecture. CUDA offers a fairly mature development environment via an extension to the C programming language. For applications CUDA version 5.5 with an

NVIDIA GT 750M are used. The GT 750M has 384 multiprocessors. For all current NVIDIA cards, a multiprocessor comprises 8 arithmetic logic units (ALUs), 2 special units for transcendental functions, a multithreaded instruction unit and on-chip shared memory. For example, for single-precision floating point computation, one can think of the GT 750 as having 3072 (384 × 8) single processors. The current generation of GPUs is 4-8 times faster at single precision arithmetic than double precision. Single precision seems perfectly sufficient for the applications in this research since the variance of the Monte Carlo estimates exceeds the perturbations due to finite machine precision.

Graphics Processing Unit for Parallel Processing

GPUs have evolved into many-core processing units, currently with up to 30 multiprocessors per card, in response to commercial demand for real-time graphics rendering, independently of demand for many-core processors in the scientific computing community. As such, the architecture of GPUs is very different to that of conventional CPUs. An important difference is that GPUs devote proportionally more transistors to ALUs and less to caches and flow control in comparison to CPUs. This makes them less general purpose but highly effective for data-parallel computation with high arithmetic intensity, i.e. computations where the same instructions are executed on different data elements and where the ratio of arithmetic operations to memory operations is high. This Single Instruction Multiple Data (SIMD) architecture puts a heavy restriction on the types of computation that optimally utilize the GPU but in cases where the architecture is suitable it reduces overhead.

Figure 1 gives a visualization of the link between a host machine and the graphics card, emphasizing the data bandwidth characteristics of the links and the number of processing cores. A program utilizing a GPU is hosted on a CPU with both the CPU and the GPU having their own memory. Data is passed between the host and the device via a standard memory bus, similar to how data is passed between main memory and the CPU. The memory bus between GPU memory and the GPU cores is both wider and has a higher clock rate than a standard bus, enabling much more data to be sent to the cores than the equivalent link on the host allows. This type of architecture is ideally

suited to data-parallel computation since large quantities of data can be loaded into registers for the cores to process in parallel. In contrast, typical computer architectures use a cache to speed up memory accesses using locality principles that are generally good but do not fully apply to data-parallel computations, with the absence of temporal locality most notable.

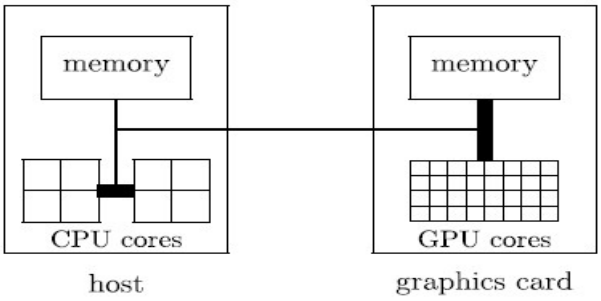

Figure 1: Link between host and graphics card. The thicker lines represent higher data bandwidth while the squares represent processor cores.

Graphics Processing Units Parallelizable Algorithms

In general, if a computing task is well-suited to SIMD parallelization then it will be well-suited to computation on a GPU. In particular, data-parallel computations with high arithmetic intensity (computations where the ratio of arithmetic operations to memory operations is high) are able to attain maximum performance from a GPU. This is because the volume of very fast arithmetic instructions can hide the relatively slow memory accesses. It is crucial to determine whether a particular computation is data-parallel on the instruction level when determining suitability. From a statistical simulation perspective, integration via classical Monte Carlo or importance sampling is ideal computational tasks in a SIMD framework. This is because each computing node can produce and weight a sample in parallel, assuming that the sampling procedure and the weighting procedure have no conditional branches. If these methods do branch, speedup can be compromised by many computing nodes running idle while others finish their tasks. This can occur, for example, if the sampling procedure uses rejection sampling.

In contrast, if a computing task is not well-suited to SIMD parallelization then it will not be well-suited to computation on a GPU. In

particular, task-parallel computations where one executes different instructions on the same or different data cannot utilize the shared flow control hardware on a GPU and often end up running sequentially. Even when a computation is data-parallel, it might not give large performance improvements on a GPU due to memory constraints. This can be due to the number of registers required by each thread or due to the size and structure of the data necessary for the computation requiring large amounts of memory to be transferred between the host and the graphics card.

Many statistical algorithms involve large data sets, and the extent to which many-core architectures can provide speedup depends largely on the types of operations that need to be performed on the data. For example, many matrix operations derive little speedup from parallelization except in special cases, e.g. when the matrices involved are sparse. It is difficult to classify concisely the types of computations amenable to parallelization beyond the need for data-parallel operations with high arithmetic intensity. However, experience with parallel computing should allow such classifications to be made prior to implementation in most cases.

Parallelizable Sampling Methods

A number of sampling methods for parallel implementations can be produced without significant modification. There is an abundance of statistical problems that are essentially computational in nature, especially in Bayesian inference. In many such cases, the problem can be distilled into one of sampling from a probability distribution whose density π, pointwise and up to a normalizing constant can be computed, that is, $\pi^*(\cdot)$ where $\pi(x) = \pi^*(x)/Z$ can be computed. A common motivation for wanting samples from π is so expectations of certain functions can be computed. If such a function is denoted by ϕ, the expectation of interest is

$$I \triangleq \int_{x \in \mathcal{X}} \phi(\mathbf{x})\pi(\mathbf{x})d\mathbf{x}$$

The Monte Carlo estimate of this quantity is given by

$$\hat{I}_{MC} \triangleq \frac{1}{N}\sum_{i=1}^{N}\phi(\mathbf{x}^{(i)})$$

where $\{\mathbf{x}^{(i)}\}_{i=1}^{N}$ are samples from π.

Samples from π in order to compute this estimate are needed. In practice, one often cannot sample from π directly. There are two general classes of methods for dealing with this. The first are importance sampling methods, where the weighted samples are generated from π by generating N samples according to some importance density γ proportional to γ^* and then estimating I via

$$\hat{I}_{IS} \triangleq \sum_{i=1}^{N} W^{(i)}\phi(\mathbf{x}^{(i)})$$

where $W^{(i)}$ are normalized importance weights

$$W^{(i)} = \frac{w(\mathbf{x}^{(i)})}{\sum_{j=1}^{N} w(\mathbf{x}^{(j)})} \text{ and } w(\mathbf{x}^{(i)}) = \frac{\pi^*(\mathbf{x}^{(i)})}{\gamma^*(\mathbf{x}^{(i)})}$$

The asymptotic variance of this estimate is given by $C(\phi, \pi, \gamma)/N$, that is, a constant over N. For many problems, it is difficult to come up with an importance density γ such that $C(\phi, \pi, \gamma)$ is small enough to attain reasonable variance with practical values of N.

The second general class of methods are MCMC methods, in which an ergodic π-stationary Markov chain is sequentially constructed. Once the chain has converged, all the dependent samples can be used to estimate I. The major issue with MCMC methods is that their convergence rate can be prohibitively slow in some applications.

For example, naive importance sampling, like classical Monte Carlo, is intrinsically parallel. Therefore, in applications where one have access to a good importance density γ, linear speedup can be got with the number of processors available. Similarly, in cases where MCMC converges rapidly, the estimation of I can be parallelized by running separate chains on each processor. While these situations are hoped for, they are not particularly interesting from a parallel architecture standpoint because they can run

equally well in a distributed system. Finally, this research is not concerned with problems for which the computation of individual MCMC moves or importance weights are very expensive but themselves parallelizable. While the increased availability of parallel architectures will almost certainly be of help in such cases, the focus here is on potential speedups by parallelizing general sampling methods. Example of recent work in this area can be found in this research, in which speedup is obtained by parallelizing evaluation of individual likelihoods.

Population-Based Markov chain Monte Carlo

A common technique in facilitating sampling from a complex distribution π with support in X is to introduce an auxiliary variable $a \in A$ and sample from a higher-dimensional distribution $\bar{\pi}$ with support in the joint space $A \times X$, such that $\bar{\pi}$ admits π as a marginal distribution. With such samples, one can discard the auxiliary variables and be left with samples from π. A kernel will generally refer to a Markov chain transition kernel as opposed to a CUDA kernel.

This idea is utilized in population-based MCMC, which attempts to speed up convergence of an MCMC chain for π by instead constructing a Markov chain on a joint space X^M using $M - 1$ auxiliary variables each in X. In general, one have M parallel 'subchains' each with stationary distribution $\pi_i, i \in \mathcal{M} \triangleq \{1, \ldots, M\}$ and $\pi_M = \pi$. Associated with each subchain i is an MCMC kernel L_i that leaves π_i invariant, and which one run at every time step. Of course, without any further moves, the stationary distribution of the joint chain is

$$\bar{\pi}(\mathbf{x}_{1:M}) \triangleq \prod_{i=1}^{M} \pi_i(\mathbf{x}_i)$$

and so if $\mathbf{x}_{1:M} \sim \bar{\pi}$, then $\mathbf{x}_M \sim \pi$. This scheme does not affect the convergence rate of the independent chain M. However, since mixtures of $\bar{\pi}$-stationary MCMC kernels can be cycled without affecting the stationary distribution of the joint chain, certain types of interaction between the subchains can be allowed which can speed up convergence. In general, a series of MCMC kernels that act on subsets of the variables is applied. The number of second-stage MCMC kernels are denoted by R and the MCMC kernels themselves as $K_1, \ldots, K_R$, where kernel K_j operates on variables with indices in $I_j \subset M$. The

idea is that the R kernels are executed sequentially and it is required that each K_j leave $\Pi_{I \in Ij} \pi_i$ invariant.

Given π, there are a wide variety of possible choices for M, $\pi_{1:M-1}$, $L_{1:M}$, R, $I_{1:R}$ and $K_{1:R}$ which will affect the convergence rate of the joint chain. The first stage of moves involving $L_{1:M}$ is trivially parallelizable. However, the second stage is sequential in nature. For a parallel implementation, it is beneficial for the I_j's to be disjoint as this allows the sequence of exchange kernels to be run in parallel. Of course, this implies that $I_{1:R}$ should vary with time since otherwise there will be no interaction between the disjoint subsets of chains. Furthermore, if the parallel architecture used is SIMD (Single Instruction Multiple Data) in nature, it is desirable to have the K_j's be nearly identical algorithmically. The last consideration for parallelization is that while speedup is generally larger when more computational threads can be run in parallel, it is not always helpful to increase M arbitrarily as this can affect the convergence rate of the chain. However, in situations where a suitable choice of M is dwarfed by the number of computational threads available, one can always increase the number of chains with target π to produce more samples.

Population-Based MCMC Algorithm

There are two types of moves:

1. In parallel, each chain i performs an MCMC move targetting π_i.
2. In parallel, adjacent chains i and $i + 1$ perform an MCMC 'exchange' move targeting $\pi_i \pi_{i+1}$

A simple exchange move at time n proposes to swap the values of the two chains and has acceptance probability

$$\min\{1, \frac{\pi_i(x_{i+1}^{(n)})\pi_{i+1}(x_i^{(n)})}{\pi_i(x_i^{(n)})\pi_{i+1}(x_{i+1}^{(n)})}\}.$$

In order to ensure (indirect) communication between all the chains, the exchange partners are picked at each time with equal probability from

$$\{\{1,2\},\ldots,\{M-1,M\}\} \text{ and } \{\{2,3\},\ldots,\{M-2,M-1\}\}.$$

Sequential Monte Carlo Samplers

SMC samplers are a more general class of methods that utilize a sequence of auxiliary distributions $\pi_0, \ldots, \pi_T$, much like population-based MCMC. However, in contrast to population-based MCMC, SMC samplers start from an auxiliary distribution π_0 and recursively approximate each intermediate distribution in turn until finally $\pi_T = \pi$ is approximated. The algorithm has the same general structure as classical SMC, with differences only in the types of proposal distributions, target distributions and weighting functions used in the algorithm.

The difference between population-based MCMC and SMC samplers is subtle but practically important. Both can be viewed as population-based methods on a similarly defined joint space since many samples are generated at each time step in parallel. However, in population-based MCMC the samples generated at each time each have different stationary distributions and the samples from a particular chain over time provide an empirical approximation of that chain's target distribution. In SMC samplers, the weighted samples generated at each time approximate one auxiliary target distribution and the true target distribution is approximated at the last time step.

Algorithmic Details

1. At time t = 0:

   For i = 1,…,N, sample $\mathbf{x}_0^{(i)} \sim \eta(\mathbf{x}_0)$

   For i = 1,…,N, evaluate the importance weights:

$$w_0(\mathbf{x}_0^{(i)}) \propto \frac{\pi_0(\mathbf{x}_0^{(i)})}{\eta(\mathbf{x}_0^{(i)})}$$

2. For times t = 1,…, T:

   For i = 1,…,N, sample

$$\mathbf{x}_t^{(i)} \sim K_t(\mathbf{x}_{t-1}^{(i)}, \cdot).$$

For i = 1,…,N, evaluate the importance weights:

$$w_t(\mathbf{x}_t^{(i)}) \propto w_{t-1}(\mathbf{x}_{t-1}^{(i)}) \frac{\pi_t(\mathbf{x}_t^{(i)}) L_{t-1}(\mathbf{x}_t^{(i)}, \mathbf{x}_{t-1}^{(i)})}{\pi_{t-1}(\mathbf{x}_{t-1}^{(i)}) K_t(\mathbf{x}_{t-1}^{(i)}, \mathbf{x}_t^{(i)})}.$$

Normalize the importance weights. Depending on some criteria, resample the particles. Set

$$w_t^{(i)} = \frac{1}{N} \text{ for } i = 1, \ldots, N.$$

For the special case where L_{t-1} is the associated backwards kernel for K_t , ie.

$$\pi_t(x_t) L_{t-1}(x_t, x_{t-1}) = \pi_t(x_{t-1}) K_t(x_{t-1}, x_t).$$

the incremental importance weights simplify to

$$w_t(\mathbf{x}_t^{(i)}) \propto w_{t-1}(\mathbf{x}_{t-1}^{(i)}) \frac{\pi_t(\mathbf{x}_{t-1}^{(i)})}{\pi_{t-1}(\mathbf{x}_{t-1}^{(i)})}.$$

The normalization step is a reduction operation and a divide operation. The resampling step involves a parallel scan.

Implementation of Canonical Examples

To demonstrate the types of speed increase one can attain by utilizing GPUs, each method to a representative statistical problem is applied. Bayesian inference for a Gaussian mixture model is used as an application of the population-based MCMC and SMC samplers.

The applications are representative of the types of problems that these methods are commonly used to solve. In particular, while the distribution of mixture means given observations is only one example of a multimodal distribution, it can be thought of as a canonical distribution with multiple well-separated modes. Therefore, the ability to sample points from this distribution is indicative of the ability to sample points from a wide range of multimodal distributions.

Mixture Modeling

Finite mixture models are a very popular class of statistical models as they provide a flexible way to model heterogeneous data. Let $\mathbf{y} = y_{1:m}$ denote identically independent distribution (i.i.d) observations where $y_j \in R$ for $j \in \{1, \ldots, m\}$. A univariate Gaussian mixture model with k components states that each observation is distributed according to the mixture density

$$p(y_j|\mu_{1:k}, \sigma_{1:k}, w_{1:k-1}) = \sum_{i=1}^{k} w_i f(y_j|\mu_i, \sigma_i),$$

where f denotes the density of the univariate normal distribution. The density of $\boldsymbol{y}$ is then equal to

$$\prod_{j=1}^{m} p(y_j|\mu_{1:k}, \sigma_{1:k}, w_{1:k-1}).$$

For simplicity, assume that k, $w_{1:k-1}$ and $\sigma_{1:k}$ are known and that the prior distribution on μ is uniform on the k-dimensional hypercube $[-10, 10]^k$. $k = 4$, $\sigma_i = \sigma = 0.55$, $w_i = w = 1/k$ for $i \in \{1, \ldots, k\}$ are set. $m = 100$ observations are simulated for $\mu = \mu_{1:4} = (-3, 0, 3, 6)$. The resulting posterior distribution for μ is given by

$$p(\boldsymbol{\mu}|\mathbf{y}) \propto p(\mathbf{y}|\boldsymbol{\mu})\mathbb{I}(\boldsymbol{\mu} \in [-10, 10]^4).$$

The main computational challenge associated with Bayesian inference in finite mixture models is the nonidentifiability of the components. As exchangeable priors have been used for the parameters $\mu_{1:4}$, the posterior distribution $p(\mu|y)$ is invariant to permutations in the labeling of the parameters. Hence this posterior admits $k! = 24$ symmetric modes, which basic random-walk MCMC and importance sampling methods typically fail to characterize using practical amounts of computation. Generating samples from this type of posterior is a popular method for determining the ability of samplers to explore a high-dimensional space with multiple well-separated modes.

Population-Based Markov chain Monte Carlo

The auxiliary distributions $\pi_{1:M-1}$ following the parallel tempering methodology are selected, that is, $\pi_i(\mathbf{x}) \propto \pi(\mathbf{x})^{\beta_i}$ with $0 < \beta_1 < \ldots < \beta_M = 1$ and use $M = 200$. This class of auxiliary distributions is motivated by the fact that

MCMC converges more rapidly when the target distribution is flatter. For this problem, the cooling schedule $\beta_i = (i/M)^2$ and a standard N(0,Ik) random walk Metropolis-Hastings kernel are used for the first stage moves.

For the second stage moves, the basic exchange move are used, chains i and j swap their values with probability min{1, α_{ij}} where

$$\alpha_{ij} = \frac{\pi_i(\mathbf{x}_j)\pi_j(\mathbf{x}_i)}{\pi_i(\mathbf{x}_i)\pi_j(\mathbf{x}_j)}.$$

Further, the exchanges to take place only between adjacent chains are allowed so that all moves can be done in parallel. $R = M/2$ and I1:R is either {{1, 2}, {3, 4}, … , {M − 1, M}} or {{2, 3}, {4, 5}, … , {M – 2, M − 1}, {M, 1}}, each with probability half are used. Emphasize that all first stage MCMC moves are executed in parallel on the GPU, followed by all the exchange moves being executed in parallel. The following code segments are to get compute value function properties for MCMC.

```
void mcmc(int M, int nb, int nt)
{
        generate_mix_data(k, sigma, mus, data_array, L);
        compute_ci1_ci2(sigma, 1.0f / k, c1, c2);
        populate_rand_d(d_array_init, numChains * k);
        multiply(numChains * k, d_array_init, d_array_init, 20, nb, nt);
        add(numChains * k, d_array_init, d_array_init, -10, nb, nt);
}
```

To test the computational time required by the algorithms the number of chains are allowed to vary but fix the number of points which wishing to sample from the marginal density $\pi_M = \pi$ at 8192. As such, an increase in the number of chains leads to a proportional increase in the total number of points sampled.

Sequential Monte Carlo Sampler

As with population-based MCMC, a tempering approach and the same cooling schedule are used, this is, $\pi_t(\mathrm{x}) \propto \pi(\mathrm{x})^{\beta_t}$ with $\beta_t = (t/M)^2$ and $M = 200$. The uniform prior on the hypercube are used to generate the samples $\{\mathrm{x}_0^{(1:N)}\}$ and perform 10 MCMC steps with the standard $N(0,I_k)$ random walk Metropolis-Hastings kernel at every time step. The generic backward kernel is

used for the case where each kernel is π_t -stationary so that the unnormalized incremental importance weights are of the form $\pi_t(\mathbf{x}_{t-1})/\pi_{t-1}(\mathbf{x}_{t-1})$. The following code segments are to compute value function properties for SMCS.

```
void testMG(int N, int nb, int nt)
 {
      generate_mix_data(k, sigma, mus, data_array, L);
      compute_ci1_ci2(sigma, 1.0f / k, c1, c2);
      populate_rand_d(d_array_init, N * k);
      multiply(N * k, d_array_init, d_array_init, 20, nb, nt);
      add(N * k, d_array_init, d_array_init, -10, nb, nt);
      testMG(N, k, T, numSteps, d_array_init, temps, h_args_t1, h_args_t2, nb, nt);
      testMG_host(N, k, T, numSteps, array_init, temps, h_args_t1, h_args_t2);
 }
```

Results and Discussion

The parallel code is run on a computer equipped with an NVIDIA GT 750M GPU, and the reference single-threaded code is run on a Intel (R)core(TM)i7 4500U CPU 1.80GHz processor. The resulting processing times and speedups are given in Tables 1–2.

Population-Based Markov chain Monte Carlo Results

Table 1: Running times for the Population-Based MCMC Sampler for various numbers of chains M.

N = 8192 points are sampled from chain M.

M	CPU(secs)	GT 750 M(secs)	Speedup
(1) 8	1.33	0.93	1
(2) 32	5.32	1.03	5
(3) 128	20.00	1.89	11
(4) 512	62.40	1.24	50
(5)2048	249.64	1.43	175
(6)8192	998.42	2.32	430
(7)32768	4002.00	7.73	518
(8)131072	16218.00	28.35	572

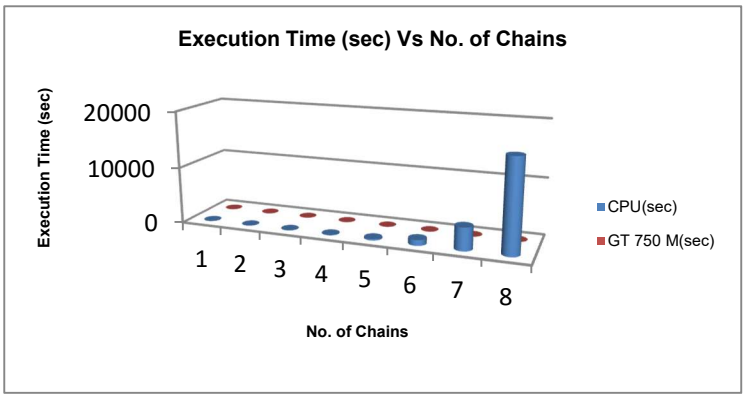

Figure 2: The relation of execution time and number of chains

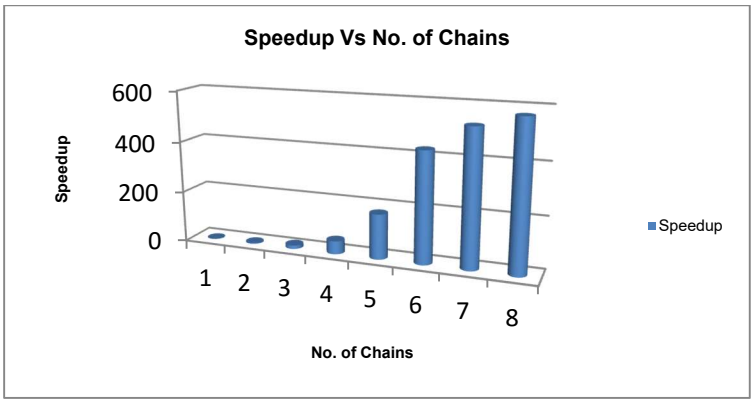

Figure 3: The relation of speedup and number of chains

Processing times for MCMC code are given in Table 1, in which one can see that using 131072 chains is impractical on the CPU but entirely reasonable using the GPU. Figure 2 shows that GPU time is faster than CPU time. Figure 3 shows that speedup goes faster with increasing the number of chains. So it can be observed that parallel computing is more suitable for enormous data.

Sequential Monte Carlo Sampler Results

Table 2: Running times for the Sequential Monte Carlo Sampler for various values of N.

N	CPU(secs)	GT 750 M (secs)	Speedup
(1)8192	266.40	0.60	444
(2)16384	529.20	1.11	477
(3)32768	1062.00	2.19	485
(4)65536	2118.00	4.50	471
(5)131072	4236.00	8.08	524
(6)262144	8460.00	16.22	522

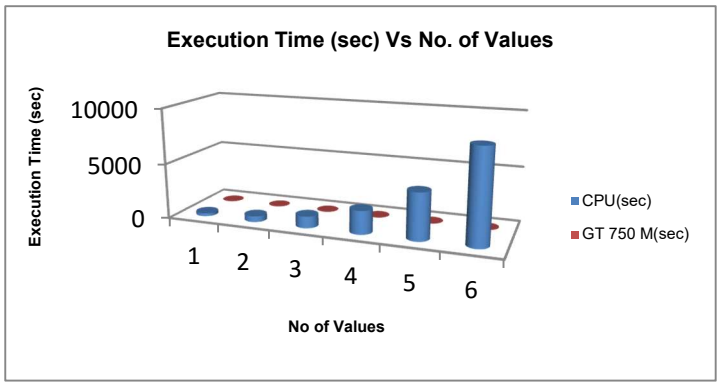

Figure 4: The relation of execution time and number of values

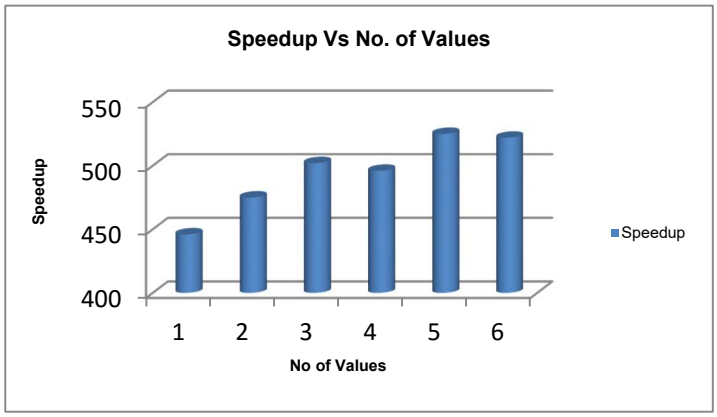

Figure 5: The relation of speedup and number of values

Processing times for SMCS code are given in Table 2. GPU execution time is faster than CPU execution time in SMC sampler that shown in Figure 4. Figure 5 shows that speedup goes faster with increasing the number of values.

Discussion

The speedup for the population-based MCMC algorithm and the SMC sampler is tremendous. In particular, the evaluation of $p(y|\mu)$ for the mixture-modelling application has high arithmetic intensity since it consists of a product-sum operation with 400 Gaussian log-likelihood evaluations involving only 104 values. In fact, because of the low register and memory requirements, so many threads can be run concurrently that SIMD calculation of this likelihood can be sped up by 500 times on the GT 750M. Estimation of static parameters in continuous state-space models or the use of SMC proposals within MCMC can require thousands of runs, so a speedup of this scale can substantially reduce the computation time of such approaches. Speedups can be expected in the vicinity of 500 with SMC if few resampling steps are required and each weighting step has small space complexity and moderate time complexity.

While CUDA have been used to implement the parallel components of algorithms, the results are not necessarily specific to this framework or to GPUs. It is expected that the many-core processor market will grow and there will be a variety of different devices and architectures to take advantage of. However, the SIMD architecture and the sacrifice of caching and flow control for arithmetic processing is likely to remain since when it is well-suited to a problem it will nearly always deliver considerable speedup. For users who would like to see moderate speedup with very little effort, there is work being done to develop libraries that will take existing code and automatically generate code that will run on a GPU.

The speedups attainable with many-core architectures have broad implications in the design, analysis, and application of SMC and population-based MCMC methods. In application, this does not occur until one have around 4096 auxiliary distributions. One might notice that this number is far larger than the number of processors on the GPU. This is due to the fact that

even with many processors, significant speedup can be attained by having a full pipeline of instructions on each processor to hide the relatively slow memory reads and writes. In both SMC and MCMC, it is also clear from this case study that it is beneficial for each thread to use as few registers as possible since this determines the number of threads that can be run simultaneously. This may be of interest to the methodology community since it creates a space-time trade-off that might be exploited in some applications.

A consequence of the space-time trade-off mentioned above is that methods which require large numbers of registers per thread are not necessarily suitable for parallelization using GPUs. For example, operations on large, dense matrices that are unique to each thread can restrict the number of threads that can run in parallel and hence dramatically affect potential speedup. In cases where data are shared across threads, however, this is not an issue. In principle, it is not the size of the data that matters but the space complexity of the algorithm in each thread that dictates how scalable the parallelization is.

Conclusion

The potential of parallel processing to aid in statistical computing is well documented. Graphics cards for certain generic types of computation offer parallel processing speedups with advantages. They are Cost: graphics cards are relatively cheap, being commodity products. Accessibility: graphics cards are readily obtainable from consumer-level computer stores or over the internet. Maintenance: the devices are self-contained and can be hosted on conventional desktop and laptop computers. Speed: in line with multi-core CPU clusters, graphics cards offer significant speedup, albeit for a restricted class of scientific computing algorithms. Power: GPUs are low energy consumption devices compared to clusters of traditional computers, with a graphics card requiring around 200 Watts. While improvements in energy efficiency are application-specific, it is reasonable in many situations to expect a GPU to use around 10 per cent of the energy to that of an equivalent CPU cluster. Dedicated and local: the graphics cards slot into conventional computers offering the user ownership without the need to transport data externally.

The parallelization of the advanced Monte Carlo methods described here opens up challenges both for practitioners and for algorithm designers. There are already an abundance of statistical problems that are being solved computationally and technological advances, if taken advantage of by the community, can serve to make previously impractical solutions eminently reasonable and motivate the development of new methods.

The speedups have practical significance. Arithmetic intensity is important. There is a roughly linear penalty for the space complexity of each thread. Emerging many-core technology is likely to have the same kinds of restrictions. There is a need for methodological attention to this model of computation. For example, SMC sampler methodology can be more suitable to parallelization when the number of auxiliary distributions one wants to introduce is not very large. There are many other algorithms that will benefit from this technology.

Acknowledgements

I would like to express my sincere grateful very much to Professor Dr Soe Mya Mya Aye, Head of Department of Computer Studies, Yangon University for her kind permission to carry out this research. My thanks to my gratitude to Dr. Pho Kaung, Rector, University of Yangon, who has given me invaluable advice and patient guidance that helped my research work to accomplish.

References

1. Andrieu C., Doucet A., and Holenstein R., (2010) "Particle Markov Chain Monte Carlo." *Journal of the Royal Statistical Society, Ser. B*, vol. 72, pp.269–342.

2. Brockwell A. E., (2006) "Parallel Processing in Markov Chain Monte Carlo Simulation by Pre-Fetching." *Journal of Computational and Graphical Statistics*, vol.15,no.1, pp.246–261.

3. Celeux, G., Hurn, M., and Robert, C. P., (2000) "Computational and Inferential Difficulties with Mixture Posterior Distributions." *Journal of the American Statistical Association*, vol.95, pp.957–970.

4. Del Moral P., Doucet, A., and Jasra, A.,(2006) "Sequential Monte Carlo Samplers." *Journal of the Royal Statistical Society, Ser. B*, vol. 68,no.3, pp.411–436.

5. Doucet A., and Johansen A. M.,(2010) *A Tutorial on Particle Filtering and Smoothing: Fifteen Years Later*, in *Handbook of Nonlinear Filtering*, eds., Oxford University Press.