Supplementary materials for this article are available online.
Please click the JCGS link at http://pubs.amstat.org.

Understanding GPU Programming for Statistical Computation: Studies in Massively Parallel Massive Mixtures

Marc A. SUCHARD, Quanli WANG, Cliburn CHAN, Jacob FRELINGER, Andrew CRON, and Mike WEST

This article describes advances in statistical computation for large-scale data analysis in structured Bayesian mixture models via graphics processing unit (GPU) programming. The developments are partly motivated by computational challenges arising in fitting models of increasing heterogeneity to increasingly large datasets. An example context concerns common biological studies using high-throughput technologies generating many, very large datasets and requiring increasingly high-dimensional mixture models with large numbers of mixture components. We outline important strategies and processes for GPU computation in Bayesian simulation and optimization approaches, give examples of the benefits of GPU implementations in terms of processing speed and scale-up in ability to analyze large datasets, and provide a detailed, tutorial-style exposition that will benefit readers interested in developing GPU-based approaches in other statistical models. Novel, GPU-oriented approaches to modifying existing algorithms software design can lead to vast speed-up and, critically, enable statistical analyses that presently will not be performed due to compute time limitations in traditional computational environments. Supplemental materials are provided with all source code, example data, and details that will enable readers to implement and explore the GPU approach in this mixture modeling context.

Key Words: Bayesian computation; Desktop parallel computing; Flow cytometry; Graphics processing unit programming; Large datasets; Mixture models.

Marc A. Suchard is Associate Professor, Departments of Biomathematics, Human Genetics and Biostatistics, University of California, Los Angeles, CA 90095 (E-mail: *msuchard@ucla.edu*). Quanli Wang is Senior Bioinformatician, Institute for Genome Sciences & Policy, Duke University, Durham, NC 27710 and Department of Statistical Science, Duke University, Durham, NC 27708 (E-mail: *quanli@stat.duke.edu*). Cliburn Chan is Assistant Professor, Department of Biostatistics & Bioinformatics, Duke University, Durham, NC 27710 (E-mail: *cliburn.chan@duke.edu*). Jacob Frelinger is Ph.D. Candidate, Program in Computational Biology & Bioinformatics, Duke University, Durham, NC 27710 (E-mail: *jacob.frelinger@duke.edu*). Andrew Cron is Ph.D. Candidate (E-mail: *ajc40@stat.duke.edu*) and Mike West is The Arts and Sciences Professor of Statistical Science (E-mail: *mw@stat.duke.edu*), Department of Statistical Science, Duke University, Durham, NC 27708.

Journal of Computational and Graphical Statistics, Volume 19, Number 2, Pages 419–438
DOI: 10.1198/jcgs.2010.10016

1. INTRODUCTION

Scientific computation using graphics processing units (GPUs) is increasingly attracting the attention of researchers in statistics as in other fields. A number of recent articles have detailed the "*what*" and the "*why*" of GPU computation; our work here assumes a reader aware of the foundations and interested now in the "*how?*".

We detail advances in statistical computing (programming perspectives and strategy, and how these guide algorithm implementation) and computation (experiences, results, and benchmarks with specific models, data, and hardware) for Bayesian analyses of structured multivariate mixtures. We are particularly concerned with mixture models with many mixture components and with large datasets—massive mixtures, where the massive parallelization offered by GPU machines has potential to define access to relevant statistical model-based methodology as a routine. The redevelopment of computation using GPUs enables scale-up on desktop personal computers that is simply not achievable using multithreaded CPU desktops and often impractical across distributed-memory computing clusters.

GPUs are dedicated numerical processors originally designed for rendering three-dimensional computer graphics. Current GPUs have hundreds of processor cores on a single chip and can be programmed to apply the same numerical operations simultaneously to each element of large data arrays under a single program, multiple data (SPMD) paradigm. As the same operations, called kernels, function simultaneously, GPUs can achieve extremely high arithmetic intensity if we can ensure swift data transfer to and from the processors.

General purpose computing on GPUs (GPGPU) is capturing the attention of researchers in many computational fields. Early adoption is growing for dynamic simulation in physics, signal and image processing, and visualization techniques (Owens et al. 2007). Computational statistics and statistical inference tools have yet to substantially embrace this new direction, though early forays are most encouraging (e.g., Charalambous, Trancoso, and Stamatakis 2005; Manavski and Valle 2008; Silberstein et al. 2008; Lee et al. 2009). Silberstein et al. (2008) first demonstrated the potential for GPGPU to impact the statistical fitting of simple Bayesian networks, and recent work, such as studies using novel GPU/CPU-based algorithms for MCMC fitting of highly structured Bayesian models of molecular sequence evolution (Suchard and Rambaut 2009a, 2009b), clearly exemplifies the opportunities; the latter example realized a greater than 100-fold reduction in run-time in very challenging and otherwise inaccessible computations.

Scientific computation using GPUs requires major advances in computing resources at the level of extensions to common programming languages (NVIDIA-CUDA 2008) and standard libraries (OpenCL: *www.khronos.org/opencl*); these are developing, and enabling processing in data-intensive problems *many orders of magnitude faster* than using conventional CPUs (Owens et al. 2007). The recent adoption of the OpenCL library for porting to popular statistical analysis packages reflects a future for algorithmic advances that are immediately available to the larger research community.

A barrier to adoption in the statistics community is the investment needed in developing programming skills and adopting new computing perspectives for GPUs; this requires

substantial time and energy to develop skills and expertise, and this challenges researchers for whom low-level programming has never been a focus. Part of our goal here is to use our context of Bayesian mixture modeling to convey not only the essence and results of our work in GPU implementations, but to also lay out a tutorial-style description and flow of the analysis so as to engage researchers who may be interested in developing in this direction in related or other classes of statistical models.

Our work is partly motivated by computational challenges in increasingly prevalent biological studies using high-throughput flow cytometry (FCM), generating many large datasets and requiring high-dimensional mixture models with large numbers of mixture components (Herzenberg et al. 2006). FCM assays provide the ability to measure multiple cellular markers for hundreds to thousands of cells per second, capturing cellular population data in a single, very high-throughput assay. FCM is relatively cheap and increasingly accessible, and will continue to grow dramatically as a preferred biotechnology as it provides data at the level of single cells—particularly critical in many areas of immunology, cancer, and emerging systems biology. The number of variables (markers) measured is currently in the 1–20 range, but technological advances are set to increase that substantially (Ornatsky et al. 2006).

Statistical analysis using various mixture modeling approaches is readily being adopted in FCM studies (Chan et al. 2008; Boedigheimer and Ferbas 2008; Lo, Brinkman, and Gottardo 2008; Pyne et al. 2009). The applied contexts involve very large sample sizes ($n = 10^4$–10^7 per assay) and, typically, quite heterogeneous distributions of the response markers reflecting multiple subtypes of cells and non-Gaussian structure. Further, standardization and routine application is pressing the need for efficient computation; typical studies can involve tens or hundreds of assays of such size (multiple treatments, time points, patients, etc.). Our developments aim, in part, to bring advanced Bayesian computation for FCM analysis into this frame. Laboratory practitioners and routine users of FCM do not and will not embrace institutional "computer center" cluster facilities, due to inaccessibility, expense, and other constraints; also, laboratory "culture" is heavily oriented around desktop computing. The availability of low-cost GPU technology on commodity desktops and workstations offers ease of access that is simply persuasive for many, hence promoting a focus on rethinking the computational implementations to exploit GPU parallelization for routine, automated use. It is with this perspective that we are interested in substantial computational advances for a class of Bayesian multivariate mixture models that provides the statistical setting.

2. STRUCTURE OF BAYESIAN MIXTURE MODELING

Many aspects of Bayesian computation can be cast in the "single instruction/program, multiple data" (SIMD/SPMD) framework to exploit the massive parallelism afforded by commodity GPUs, with potential speed-ups of orders of magnitude. A central case in point is Bayesian analysis of mixture models with sample sizes in the millions and hundreds of mixture components; these result in massively expensive computations for Markov chain Monte Carlo (MCMC) analysis and/or Bayesian EM (BEM) for local mode search and

optimization, but that can easily be cast in the SIMD/SPMD framework. We focus on multivariate normal mixture models under truncated Dirichlet process (TDP) priors (Ishwaran and James 2001), a variant of a model class very popular in density estimation and classification problems in applied statistics and machine learning (e.g., Escobar and West 1995; MacEachern and Müller 1998a, 1998b; Teh et al. 2006; Ji et al. 2009). Applied, substantive variants involving heavier-tailed or skewed mixture components, or hierarchical extensions in which component densities are nonnormal, themselves modeled as mixtures of normals, add detail to the computations but do not impact on the main themes and results of interest here.

A p-dimensional data vector $\mathbf{x}$ has density $g(\mathbf{x}|\boldsymbol{\Theta}) = \sum_{j=1:k} \pi_j N(\mathbf{x}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$ with parameters $\boldsymbol{\Theta} = \{\boldsymbol{\pi}_{1:k}, \boldsymbol{\mu}_{1:k}, \boldsymbol{\Sigma}_{1:k}\}$ for a fixed upper bound k. Analysis uses conditionally conjugate, parameterized priors of standard form for $(\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$, coupled with the specific class of priors arising in the TDP model for the mixture weights $\boldsymbol{\pi}_{1:k}$. Details are summarized in Appendix A. We use MCMC simulation methods and posterior mode search via BEM. The most computationally expensive aspect of the analysis lies in evaluation of posterior configuration probabilities

$$\pi_j(\mathbf{x}_i) \equiv \Pr(z_i = j|\mathbf{x}_i, \boldsymbol{\Theta}) = \pi_j N(\mathbf{x}_i|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)/g(\mathbf{x}_i|\boldsymbol{\Theta}) \qquad (j = 1:k).$$

The MCMC analysis then adds multinomial draws of configuration indicators $(z_i|\mathbf{x}_i, \boldsymbol{\Theta}) \sim \text{Mn}(1, \boldsymbol{\pi}_{1:k}(\mathbf{x}_i))$ $(i = 1:n)$. The $n \times k$ tasks imply billions of subsidiary calculations when n and k are large. Conditional independencies coupled with the relatively simple operations involved make this a prime target for GPU implementations. BEM calculations involve a complete scan over all n observations, weighted by their current configuration probabilities, for each of the k components at each iterate; MCMC involves component-specific calculations with the data subset currently configured into that component.

3. CODING FOR THE GPU

3.1 STRATEGIES AND PROCESS

Coding for the GPU is fundamentally an exercise in parallel programming, and the theoretical maximum speed-up is bounded by Amdahl's quantity $1/(1 - P + P/N)$ (Amdahl 1967) where P is the fraction of the program that can be parallelized and N is the number of processors. Clearly, the bottleneck is in the fraction of irreducibly serial computations in the code; if P is 90%, the maximum speed-up is only 10-fold even with an infinite number of processors. However, the fraction P is not necessarily fixed, and typically increases with problem size as first pointed out by Gustafson (1988). Hence, MCMC and related approaches that are often viewed as intrinsically *serial* algorithms can still benefit from tremendous parallelization speed-ups with the use of GPUs. The contexts in which this is attractive are those that involve intense, parallelizable computations *per iterate* of the MCMC. Since the basic algorithms for mixture modeling rely on iterative Gibbs samplers, the usual *coarse-grain* approach taken for "embarrassingly parallel" problems with message passing techniques (e.g., MPI on a Beowulf cluster) simply do not work well

due to the need for sequential updates of global model parameters. With increasingly large datasets and larger numbers of mixture components, however, each iteration becomes increasingly burdensome; hence the interest in opportunities to exploit *fine-grain* problem decompositions for parallelizing *within* iterations.

3.1.1 Problem Decomposition for MCMC

We illustrate the strategy of problem decomposition using the so-called block Gibbs sampling approach in TDP models. The size of a given problem is determined by the three integers (n, p, k), respectively the data sample size, variable dimension, and maximum number of mixture components. In flow cytometry applications, p is usually modest (5–20), while n can range from 10^5 to 10^7 and k typically ranges from 10^1 to 10^3. These quantities suggest that any sampling steps involving n and/or k are potential candidates for parallelization.

Each MCMC iteration (Appendix A) involves four sequential steps to block-update model parameters, rephrased here with time complexity estimates in big-O notation:

1. Resample configuration indicators. This step draws a new indicator for each data point over the k components independently $[O(nkp^2)]$.
2. Resample mean and covariance matrices. This step involves updating group sample means and covariance matrices for each component independently based on the new indicators from step 1, and then drawing new parameters $[O((n+k)p^2)]$.
3. Resample the mixture weights $[O(k)]$.
4. Resample the Dirichlet process precision parameter $[O(k)]$.

Step 1 and, to a much lesser extent, step 2 represent key computational bottlenecks for problems in which both n and k are large—the massive mixture contexts. For example, for a serial version of MCMC for the TDP, when $n = 10^6$ and $k = 256$, step 1 accounts for more than 99% of overall compute time, making step 1 the prime target for parallelization. This number only increases as n and k become even larger.

3.1.2 Potential Gains

The analysis in Section 3.1.1 provides potential targets for parallelization. However, we still need to identify the proportion of remaining code that is serial so as to understand the potential speed-up and assess just how worthwhile it might be to invest the significant effort needed for effective parallelization. In our TDP example, the serial components (including steps 2–4 that are not immediate targets for parallelization currently) comprise a very low percentage of the total compute burden. Focusing on step 1 alone can yield theoretical speed-up greater than 100-fold, so is well worth the effort to parallelize.

In practice, of course, the theoretically achievable speed-up is never realized due to the finite number of parallel devices (e.g., GPU cores, CPU cores, processing nodes in a Beowulf cluster) available, and also because there are latency costs involved in moving data

to, from, and between these devices. Suppose that we have S devices, and that each can be programmed to perform the same computational task in the same amount of time. If a parallelizable task with overall execution time c_0 can be evenly distributed, then an ideal S-fold speed-up will yield actual time of $c_0/S + c_1$ where c_1 is an incurred access/utilization time. To make things more complicated, if we have to divide tasks into even smaller pieces (such as assigning m tasks per parallel device) for some legitimate reason, the overhead parameter c_1 will then be multiplied by m. Any strategy to define a parallel algorithm has to take these issues into account in assessing potential speed-up benefits.

3.1.3 Coarse Grain Parallelization (MPI/Multi-Core Computing)

The most familiar form of parallel computing is MPI-based computing on Beowulf clusters of inexpensive CPU machines, widely employed in academic institutions. Clusters are usually used in the "embarrassingly parallel" mode; more delicately, in the "master/slave" model for scientific computing. Clusters can scale up to large numbers of processing nodes with typical research institute clusters having thousands of nodes. However, since most of these clusters are rack mounted computers with multi-core CPUs linked loosely through an Ethernet network, sharing of data across nodes has high latency, making the parallelization overhead parameter c_1 relatively large. Because of the high latency, coarse-grain decompositions with larger and relatively independent tasks assigned to each computing node are optimal and result in minimizing the parallelization overhead mc_1 for m tasks per node.

Another common complication is cluster node heterogeneity. As large computer clusters evolve, hardware updates lead to mixtures of architectures, CPU speed, and memory available. With coarse-graining, the time for each iteration is held back by the slowest node, dragging down overall performance. Hence there is usually some trial-and-error to determine the optimal granularity and load-balancing of task decomposition for the specific cluster available. In our experiences running MCMC for TDP models on the large Duke University high-performance cluster, a granularity in the vicinity of 5000 data points per compute node works best. Even then the resulting speed-ups, while significant, are disappointing due to the high latency of data transport. For example, with $n = 5 \times 10^6$, $k = 256$, and $p = 14$, speed-up of approximately only 20-fold is a good benchmark on a cluster of 100 nodes.

As an alternative to MPI-based clusters, modern desktop and laptop computers usually come with two to four homogeneous CPU cores that can be effectively used for parallel computing. Using either OpenMP (*http://openmp.org/wp/*) or various multi-threading libraries (e.g., Intel Thread Building Block (TBB), Boost Thread library), we can write efficient parallel code that uses these cores. The homogeneity of CPU cores and the fact that all memory is shared by all cores make communication between threads (tasks) much faster, thus incurring much smaller overhead costs (mc_1) than MPI-based cluster solutions. For example, using the Intel TBB (Thread Building Block) library, we have implemented a multi-threading version of MCMC for the TDP; this achieves a fivefold speed-up on an 8-core (dual quad-core) machine. Of course, this approach is severely limited in scalability by the number of cores available, and is useful only for problems of modest size.

3.1.4 Fine-Grain Parallelization on the GPU

In the last couple of years, GPUs have begun to attract attention in scientific computational communities due to their very low expense and high performance. Current commodity GPUs can have an extremely large number of relatively simple compute cores (240–480), each being capable of executing fast arithmetic and logical instructions. Data on a GPU can be in *device* memory that is accessible to all cores, *shared* memory that is common to a *block* of cores, and also in *registers* specific to individual cores. Unlike CPU cores, cores in the GPU have very limited numbers of registers and small shared memory resources, and completely lack control logic for branch prediction and other aggressive speculative execution techniques. Therefore, an understanding of GPU architectural limitations is critical to maximally exploiting GPUs. The architecture in a GPU has the following distinctive features:

1. communication between CPU and GPU is only possible via device memory;
2. device memory on board can be accessed by all cores but data access is much slower than for shared memory;
3. barring memory bank conflicts, shared memory is as efficient as the use of registers (1 clock cycle);
4. the shared memory within a core block allows extremely fast communication/collaboration among cores;
5. the creation of GPU threads is significantly faster than that of CPU threads.

The limited number of registers and small shared memory essentially mandate a very fine-grain decomposition of parallelized algorithms for optimal performance. Such high parallelism critically hides memory latency, as different threads continue to execute while others wait on memory transactions. Yet, because of the low cost of thread creation and ultra-efficient data access from registers and shared memory, dramatic speed-ups are possible for algorithms such as those that arise in massive mixture context.

In contrast to the coarse grain size of the CPU/multi-threading approach—in which each parallel CPU task thread tries to work on as many data points as possible to reduce parallelization overhead costs—the GPU implementation goes to the opposite extreme and makes multiple threads work on a *single* data point. More specifically, a block of threads (limited to 512 threads in current GPU architectures) first loads the necessary data concurrently from device memory to much faster shared memory. Then each thread evaluates only one data point on only one component density and writes the result back to device memory. The function implementing this is called a *kernel function*; kernel functions are typically very simple with little or no code branching. In the MCMC mixture analysis with computation of the component configuration indicators, each data point is evaluated against each component density, following which another kernel function reads these results back into shared memory to calculate cumulative weights and sample new indicators for each data point.

This approach proves to be very successful. With the same dataset, we can achieve more than 120-fold speed-up for the overall program using one standard, cheap consumer

GPU (NVIDIA GeForce GTX285 with 240 cores). This impressive factor results from parallelizing step 1 alone. Detailed discussion of speed-up across various datasets and using different GPUs and machines appears below.

For a comparable cluster-based approach, if we ignore communication latency between nodes a 120-fold speed-up requires (at a severe underestimate, since we only achieved a 20-fold speed-up with 100 CPU cores) at least 119 additional CPU cores. Assuming high-end, quad-core CPUs would require about 30 nodes—∼U.S. $50,000 at today's costs. This estimate does not include maintenance, permanent personnel support, space, or air-conditioning for such a large system. In any case, the cluster approach compares quite poorly with the approximate $400 cost of a GTX285 card currently.

3.2 CUDA TUTORIAL FOR BAYESIAN SIMULATION AND OPTIMIZATION

Once a strategy for fine-grain decomposition of a problem has been decided, it only remains for it to be written and compiled for the GPU. For NVIDIA hardware, this currently can be done either with the CUDA SDK (*http://www.nvidia.com/object/cuda_get.html*) or with the OpenCL library (*http://www.khronos.org/registry/cl/*). We describe the approach using CUDA but the OpenCL approach is broadly similar. We detail implementation and the evolution of the MCMC code from serial to MPI/multi-threaded to GPU code that results in over two orders of magnitude speed-up. BEM code highlights differences in coding simulation and optimization routines.

3.2.1 MCMC: Serial, Multi-Threaded, and CUDA

The resampling configuration step,

1 **for** each $i = 1:n$, compute $\boldsymbol{\pi}_{1:k}(\mathbf{x}_i)$, then
2 draw z_i independently from $\mathrm{Mn}(1, \boldsymbol{\pi}_{1:k}(\mathbf{x}_i))$,

is the main bottleneck in the MCMC routine. This typically accounts for over 99% of the execution time with large datasets, so is the primary target for parallelization.

3.2.2 Serial Version

Algorithm 1 gives the C-style pseudo-code for this sampler in a standard serial CPU implementation.

The function `sample` here simply normalizes the weights and then performs the multinomial draw. A number of optimization considerations can be found in our source code, which is essentially fully optimized for serial implementation.

3.2.3 MPI/Multi-Threaded Version

The strategy for parallelizing this code snippet with large n, k is obvious in view of the nested loops over samples and components. Any large grain size parallelization will allow each "slave" CPU to compute the configuration probabilities and sample indicators for a

Algorithm 1 Serial CPU version.

```
for (j = 1; j <= k; j++)
    L_j = Cholesky(Σ_j);
for (i = 1; i <= n; i++) {
    for (j = 1; j <= k; j++)
        π_j(x_i) = π_j mvnormpdf(x_i, μ_j, L_j);
    z_i = sample(π_{1:k}(x_i));
}
```

subset of the data, returning the indicators alone to the "master" computer node for the next step computations. Algorithm 2 outlines the pseudo-code.

Compared to serial Algorithm 1, this allows multiple CPUs to work in parallel to resample component indicators. GrainSize will be chosen based on the numbers of nodes in the cluster and other aspects of the parallel computing architectures.

3.2.4 CUDA Version for GPUs

Parallel programming for GPU devices takes a quite different approach due to differences in hardware (limited fast processing registers and shared memory resources) as well as in the use of threads in scheduling tasks. We start with a simple framework for CUDA programming, then extend it to describe the mixture analysis algorithms.

CUDA is implemented as a set of extensions to a subset of the C language. The NVIDIA nvcc compiler splits and runs two code components: one on CPU(s) and one on GPU(s)

Algorithm 2 MPI/Multi-threaded version.

```
for (j = 1; j <= k; j++)
    L_j = Cholesky(Σ_j);
GrainSize = FindOptimalGrainSizeBasedOnSystemResources();
NumberOfTasks = n/GrainSize;
for (task=1; task<=NumberOfTasks; task++) {
    CurrentNode = FindTheNextAvailableParallelDevice();
    SampleConfigIndicators(task*GrainSize:
        (task+1)*GrainSize);
}
function SampleConfigIndicators(start_data_point,
    end_data_point) {
    for (i=start_data_point; i<=end_data_point; i++) {
        for (j = 1; j <= k; j++)
            π_j(x_i) = π_j mvnormpdf(x_i, μ_j, L_j);
        z_i = sample(π_{1:k}(x_i));
    }
}
```

as CUDA *kernels*. CUDA refers to a CPU as the *host* and a GPU as the *device*. CUDA extends C with declaration specifications for functions that determine where the function call can originate and where it is executed. The default `__host__` declaration is a regular C function, called from and executed on the host CPU. In contrast, the `__device__` declaration describes a function called from and executed on the GPU device. The CUDA kernel is key: it is there that the program transfers execution from CPU to GPU. This can be done asynchronously, allowing simultaneous CPU and GPU computations. Kernels are executed in parallel and threads are organized into *blocks*. Threads within a block can communicate using extremely low-latency *shared memory*; optimizing the use of this shared memory is fundamental to GPU performance. There are at most 512 threads in a block, and blocks are organized into *grids*. A block can be specified as a one- or two-dimensional array; a grid can be specified as a one-, two-, or three-dimensional array, with the dimensionality chosen to match the problem being solved. Thread organization is specified with a `<<<gridSize,blockSize,sharedMemSize>>>` syntax between the kernel function name and its argument list. The parameters `gridSize` and `blockSize` are defined using the predefined type `dim3` (e.g., `dim3 gridSize(4, 2, 1)`) and `sharedMemSize` specifies the number of bytes in shared memory to be dynamically allocated per block for this kernel function call.

Pseudo-code in Algorithm 3A outlines a generic CUDA program that defines the following basic steps:

lines 2, 3: Allocate memory for data storage on both CPU (host) and GPU (device).

line 4: Data on host is transferred to device.

line 5: A parallel execution plan is designed and a kernel function is called. The CUDA driver schedules the parallel execution of thread blocks on the device cores. Typically, the kernel function calls a thread synchronization function to ensure that all threads are completed before exiting and returning control to the host CPU.

line 6: Results transferred back to host from device and CPU takes over.

line 7: Device memory is cleared and the program ends.

Algorithm 3A GPU CUDA program template.

```
1  int cudatemplate() {
2    AllocateMemoryOnHost();
3    AllocateMemoryOnDevice();
4    CopyHostMemoryToDevice();
5    InvokeKernel<<<gridSize, blockSize,
         sharedMemSize>>>(parameters);
6    CopyDeviceMemoryToHost();
7    FreeDeviceMemory();
8  }
```

Algorithm 3B GPU MCMC template version.

```
1  int cudatemplate() {
2    AllocateMemoryOnHost();
3    AllocateMemoryOnDevice();
4    for(iter=1; iter<=nBurn+nIter; iter++) {
5      CopyHostMemoryToDevice();
6      EvalPDF();
7      SynchronizeThreads();
8      SampleIndicators();
9      SynchronizeThreads();
10     CopyDeviceMemoryToHost();
11   }
12   FreeDeviceMemory();
13 }
```

This generic framework is slightly modified for the MCMC mixture algorithms as we execute two kernel functions sequentially over many iterations: one to evaluate the multivariate normal pdf for each data point on each mixture component (`EvalPDF`), and another for the follow-on multinomial sampling probabilities (`SampleIndicators`); see Algorithm 3B. From a programming strategy perspective, the essential differences between the MCMC algorithm for the GPU and the MPI/multi-threaded parallel versions are:

1. We have very fine grain size for parallelization: each thread (task) calculates for only one data point $\mathbf{x}_i$ against one normal density $N(\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$.
2. A dedicated execution plan is needed to decide how to maximize performance by organizing threads into blocks so they can cooperate with each other and optimize the use of low-latency shared memory and thread registers.
3. GPU computations are typically done in single-precision, since double-precision operations are up to 10 times slower with the current generation of GPUs. This is key with current technology, but will change as substantially faster double-precision GPUs emerge in future generations.

The following sections detail each of the GPU program steps, with emphasis on coding techniques to optimize performance.

3.2.5 Allocate Memory on CPU and GPU and Move Data From CPU to GPU

The host (CPU) and device (GPU) have separate memory spaces. CUDA supports three types of data storage: global, constant, and shared memory, with vastly different access methods and speed. Global memory is large (up to 4G in the NVIDIA Tesla chips in 2009) and the only memory accessible to the host, so all data moving from host to device or vice versa needs to go through global memory. Constant memory is basically global memory that is textured for faster access. Within a thread block, threads can communicate using

shared memory whose latency is more than 100 times faster than global memory, but the capacity is relatively small (16K in current Tesla chips, although the next generation Fermi chips in 2010 will have up to 64K). Individual threads also have access to a limited number of registers.

Efficient CUDA kernel code usually takes advantage of fast shared memory and is optimized in moving data from global memory to shared memory to minimize latency. To maximize data-throughput, the GPU hardware combines—or *coalesces*—memory read/write operations of 16 consecutive threads into one single wide memory transaction. Without coalescence of global memory transactions, separate memory transactions occur for each thread, resulting in high latency. Our algorithm addresses this issue by padding the specified variable dimension p so as to read/write only multiples of 16 values at a time. The following code snippet illustrates padding to optimize read/write operations:

```
1  REAL* d_ix;
2  int DATA_PADDED_DIM=16;
3  while (p>DATA_PADDED_DIM) {
4    DATA_PADDED_DIM+=16;
5  }
6  unsigned int mem_size = n*DATA_PADDED_DIM*sizeof(REAL);
7  cutilSafeCall(cudaMalloc((void**) &d_ix, mem_size));
8  cutilSafeCall(cudaMemcpy(d_ix, h_ix, mem_size,
       cudaMemcpyHostToDevice));
```

Here we allocate $n \times$ `DATA_PADDED_DIM` floating point numbers in the GPU global memory and transfer the corresponding data on the CPU to the GPU.

3.2.6 Shared Memory and Registers

Up to 512 threads can be grouped into a thread block, and these threads have shared access to 16KB of shared memory that performs 100–150 times faster than even coalesced global memory transactions. However, 16KB is very small and only holds 4096 single-precision values. Individual threads also have private access to very fast but scarce registers. All threads in a block share a 16KB register file that holds all the register values. If a block contains the maximum of 512 threads, each thread will only be allocated $(16 \times 1024)/(512 * 4)$ single-precision registers. If more registers are requested than are available, the excess data are stored in *global* memory, resulting in a devastating drop in performance.

Efficient caching is thus simply critical to optimizing performance. As a result, besides using data padding for the input data to enforce coalescing shared memory transactions, we further combine all the parameters $\{\pi_{1:k}, \boldsymbol{\mu}_{1:k}, \boldsymbol{\Sigma}_{1:k}\}$ into a $k \times$ `PACK_DIM` matrix; each row holds all the information for one mixture component, padded with zeros at the end of the row such that `PACK_DIM` is a multiple of 16. We attempt to cooperatively prefetch the largest possible chunks of data sitting in global memory using coalesced transactions and cache these values in shared memory, in an order that maximizes their reuse across the threads in block.

The code snippet in Appendix B (in the Supplemental Material) is the kernel function that combines these strategies and steps. The implementation is relatively direct, with typical consideration for shared memory reuse and coalesced memory transactions. Yet the level of parallelization it achieves allows the density evaluation to be more than 150 times faster than its serial peer. Together with another relatively simple kernel function for sampling the configuration indicators, the overall performance improvement is about 120-fold in total run-time for large datasets.

3.2.7 Blocking for Large Dataset

While the performance enhancement is impressive, this implementation requires more memory than the serial and MPI/multi-threading versions. In non-GPU versions, the parallel task grain size is relatively large and each task can deal with multiple data points without saving density values for each combination of (i, j), as shown in Algorithms 1 and 2. This is not the case for Algorithm 3. The particular, efficient design of thread blocks allows for densities to be evaluated in parallel. However, the efficiency comes with the cost of having to save all these densities in global memory before applying another kernel to sample from them. This will become a problem for really huge datasets on commonly used GPU devices such as GTX285 with just 1G or 2G device memory. We address this by chunking data into subsets, and launching the kernel functions multiple times, once for each subset. The overheads of this approach are minimal since it is relatively cheap to launch kernel functions multiple times (a full implementation can be found in the source code).

3.2.8 Using Multiple GPUs

Practical implementations typically exploit multiple GPUs as well as multiple CPUs. By chunking data into subsets and launching multi-threads on multiple CPUs, with each CPU controlling one GPU, we gain substantial further speed-up; see benchmarks report in Section 4. Additional code illustrating this, and allowing interested researchers to exploit multiple processors, is available in our implementations at the web site (see URL below).

3.2.9 Bayesian EM for TDP Mixtures—Serial and CUDA Algorithms

Posterior mode search using EM (Bayesian EM, or BEM) and other optimization algorithms can also greatly benefit from massive parallelization even though they contain no random sampling. These algorithms can ultimately be reduced to simple sums and matrix manipulations, and in the large data context, these basic functions are extremely cumbersome and are prime candidates for parallelization. The dominant computations in BEM in our mixture models (see details in Appendix A) are as follows:

1 **for** each $i = 1:n$, compute $\boldsymbol{\pi}_{1:k}(\mathbf{x}_i)$;
2 **for** each $j = 1:k$, calculate $\mathbf{m}_j = \sum_{i=1}^{n} \pi_j(\mathbf{x}_i)\mathbf{x}_i$ and $\mathbf{S}_j = \sum_{i=1}^{n} \pi_j(\mathbf{x}_i)(\mathbf{x}_i - \mathbf{m}_j)(\mathbf{x}_i - \mathbf{m}_j)'$;

The CPU version of this algorithm parallels that for the MCMC, differing in the evaluations of probability weighted sample means and covariance matrices that scan across the entire dataset rather than just a subsample. This latter element imposes additional storage as well as computational requirements, but does not modify the overall program flow.

3.2.10 CUDA Version

Let $\mathbf{\Pi}$ be the $n \times k$ matrix of configuration probabilities $\pi_j(\mathbf{x}_i)$ $(i = 1:n, j = 1:k)$, $\mathbf{M}$ the $p \times k$ matrix whose columns are the p-vector weighted means $\mathbf{m}_j$ $(j = 1:k)$, and $\mathbf{X}$ the $n \times p$ data matrix. BEM involves multiplication $\mathbf{M} = \mathbf{X}'\mathbf{\Pi}$ of two very large matrices. In standard CPU implementations we can use highly optimized libraries, such as BLAS. The analogous library CUBLAS (NVIDIA-CUBLAS 2008) in the CUDA toolkit is a GPU implementation of basic BLAS functions. As with BLAS, CUBLAS reads and writes matrices in column-major format, while the C convention is row-major, so all results will be implicitly transposed. Algorithm 5A uses CUBLAS in our GPU template.

CUBLAS handles all kernel executions inside the `cublasSgemm` function and parameter inputs follow the BLAS library; see NVIDIA-CUBLAS (2008) for detailed information. CUBLAS performs well for sufficiently large matrices and can be a powerful component of CUDA code leading to impressive GPU speed-up with limited work involved.

Calculating the weighted sample covariance is more specialized and requires a custom kernel. There are several approaches but, due to the limited shared memory (Section 3.2.6), we are led to calculating the $\mathbf{S}_j$ element-wise, reducing to a problem of maximizing memory bandwidth by reading numbers in a coalesced fashion (Harris 2008), as in the kernel detailed in Appendix C (in the Supplemental Material). Implementing this simple kernel and prepackaged CUBLAS libraries has produced speed-ups as high as 80-fold as noted in the following section.

Algorithm 5A GPU BEM template version with CUBLAS.

```
1  int main() {
2     cublasInit(); // Required before any CUBLAS functions are called.
3     AllocateMemoryOnHost();
4     AllocateMemoryOnDevice();
5     for (iter=1; iter<=nBurn+nIter; iter++) {
6          CopyHostMemoryToDevice();
7          EvalPDF();
8          SynchronizeThreads();
9          // tM = M' input
10         cublasSgemm('n','t', k,p,n,1.0,Π,n,X,n,0.0,tM,k);
11         SynchronizeThreads();
12         CopyDeviceMemoryToHost();
13    }
14    FreeDeviceMemory();
15 }
```

4. BENCHMARKS

Table 1 compares compute times in running the MCMC for the TDP model for sample sizes that range up the levels common in flow cytometry studies, and for $p = 14$ to reflect the current ranges of variables of interest in such studies. These results were developed using subsamples of a flow cytometry dataset in $p = 14$ dimensions with a mixture model having $p = 256$ components. Comparisons show running times for 100 MCMC iterates, comparing analyses with various sample sizes n (rows) and with all other aspects of the model and MCMC specification common. The first set of analyses (first five columns of the tabulated times) were all run on the same machine, and differ in utilizing different processors: gpu 1, a single NVIDIA GeForce GTX285 with 240 cores; tesla, a single Tesla C1060; gpu 3, running two GTX285 processors and one Tesla C1060 together; cpu 1, an Intel Core 2 Quad 3.33 GHz CPU; and cpu 8, multi-threaded parallelization on two Intel Core 2 Quad 3.33 GHz CPUs. This standard desktop compute and gaming machine currently costs in the ∼U.S. $5000 range. The final two columns give times from running on a standard laptop, a MacBook with Intel Core 2 Duo 2.66 GHz CPU, 4GB (1076 MHz) RAM and running the GPU component on one of two installed graphics cards, the 32 core NVIDIA GeForce 9600M GT with 512 MB memory. Performance on this latter computer is particularly relevant as it reflects scale-up ability on routinely configured, cheap "everyday" laptops (∼U.S. $1500 currently).

Note that the Tesla C1060 is about 20% slower than each GTX285 due to slower memory reading/writing and, as a result, the performance is downgraded a little on the desktop running on the Tesla GPU. The MacBook is naturally slower than the desktop given the substantial differences in processor and memory speed, but note that the speed-up using a single GPU compared to the CPU is comparable and substantial, enabling 16- to 25-fold faster computations with medium to larger datasets.

Additional benchmarks for the BEM computations show comparable computational improvements although the picture is less clear than for MCMC; see Table 2. The BEM algorithm is less efficient for small n because it requires a complete scan and calculations on all n observations and is thus naturally less efficient in parallelized mode than the MCMC that splits into subsets of the data allocated across threads. However, when n is moderately large, the GPU implementation is vastly more efficient than its CPU counterpart. In general, moving to the GPU requires a good deal of overhead that is partly offset in the MCMC

Table 1. Running times (in seconds) for 100 iterations of the MCMC analysis of TDP model.

n	gpu 1	gpu 3	tesla	cpu 8	cpu 1	mac gpu	mac cpu
10^2	1.225	1.243	1.226	3.0	3.0	2.119	5.0
10^3	1.42	1.36	1.45	20.0	20.0	3.654	30.0
10^4	3.18	2.46	3.49	94.0	191.0	18.78	277.0
10^5	20.4	13.1	23.7	386.0	1907.0	169.7	2758.0
10^6	192.0	119.5	224.6	3797.0	19,048.0	1118.1	27,529.0
5×10^6	954.0	591.0	1116.0	17,667.0	95,283.0	5785.8	141,191.0

Table 2. Running times (in seconds) for 100 iterations of the BEM analysis of TDP model.

n	gpu 1	cpu 1	mac gpu	mac cpu
10^2	71.1	4.0	123.1	8.0
10^3	81.3	40.0	125.2	63.0
10^4	91.2	607.0	159.5	1362.0
10^5	129.6	7793.0	439.0	16,165.0
10^6	680.6	78,765.0	5032.5	165,973.0
5×10^6	4436.1	—	—	—

analysis by exploiting this greater parallelization opportunity. Also note that 100 iterations of BEM is generally slower than 100 iterations of MCMC. This is because in BEM the component means and covariance matrices are computed based on all of the data while the MCMC components are based on a random subset.

5. ADDITIONAL COMMENTS

In August 2009, the *New York Times* ran a front-page article on statistics, "big data," and the challenges and opportunities in data-rich, 21st century science (Aug. 6, 2009 NYT: "*For Today's Graduate, Just One Word: Statistics*"). Noting the core challenges to "*. . .the ability of humans to use, analyze and make sense of. . .*" increasingly complex, massive amounts of data in many fields, the article says, "*The new breed of statisticians tackle that problem. They use powerful computers and sophisticated mathematical models to hunt for meaningful patterns and insights in vast troves of data*." Though historically such problems tend to be tackled more readily by computer scientists, physicists, or engineers who more aggressively embrace new technology, the opportunities for statisticians, and for conceptual innovation in computational modeling in statistics that responds to increasingly complex, high-dimensional problems, are open and waiting. Statistical thinking must be partially customized to, motivated by, and evolving with rapidly changing modes of computation. Distributed computation on increasingly large (and expensive) institutional clusters as well as multi-core desktops is fairly common in statistical work, but GPU hardware offers opportunities for vastly increasing the use, reach, and impact of statistical modeling on everyday desktops.

Bayesian computation in highly structured, multilayered models—that lie at the heart of many significant data mining, prediction, and discovery contexts in "data-rich studies"—is a key example of an arena seeing initial developments in the use of GPUs, and where the technology is opening up simply major new opportunities for orders of magnitude advances in the ability to compute. Our studies and innovations in computing and computation for structured mixture models represent just one area in which such advances are being made, and we hope that the detailed discussion of this work will aid others in making advances with other model classes.

To aid in this, we have made all code available, together with links and resources related to libraries of interest to statisticians working with, or beginning to engage in, GPU-based

computational methods in statistics; see details in the Supplemental Materials section below.

The potential for further growth in GPU computing is great; often quoted in 2009, Nvidia CEO Jen-Hsun Huang predicted that GPU compute capabilities are likely to increase by 570-fold over the next six years (Huang 2009), greatly outpacing CPU development. The computer gaming industry is now rapidly broadening its interests to address increasing attention from scientific arenas, and the next couple of years will see substantial advances in the ability to more easily access and program with GPUs as a result. There is also increasing interest in developing hybrid GPU-CPU technologies that have the potential to bring massive parallelization to the desktop faster and more forcefully than to date.

A. APPENDIX: ASPECTS OF BAYESIAN TDP MIXTURE MODELS

The TDP analysis uses normal, inverse Wishart priors for normal component parameters, $(\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \sim N(\boldsymbol{\mu}_j|\mathbf{m}, \gamma\boldsymbol{\Sigma}_j)\,\mathrm{IW}(\boldsymbol{\Sigma}_j|\nu+2, \nu\boldsymbol{\Phi})$ independently over $j = 1:k$. This is coupled with the implicit priors over mixture probabilities arising from the underlying DP model, that is, $\pi_1 = v_1$, $\pi_j = v_j\prod_{r=1}^{j-1}(1-v_j)$ for $j = 2:k-1$ and $\pi_k = 1$, where $v_j \sim \mathrm{Be}(1,\alpha)$ for $j = 1:k-1$. The DP precision parameter α has a conditionally conjugate gamma prior. Analysis includes learning on hyperparameters $\{\mathbf{m}, \gamma, \nu, \boldsymbol{\Phi}\}$; this adds compute burden, but does not materially impact on the general results and ideas of this article.

Based on data $\mathbf{x}_{1:n}$ of sample size n, the block MCMC algorithm successively resamples values of the parameters $\boldsymbol{\Theta} = (\boldsymbol{\pi}_{1:k}, \boldsymbol{\mu}_{1:k}, \boldsymbol{\Sigma}_{1:k})$ of the k normal components. Optimization computations to find posterior modes follow related steps, and in practice typically involve running multiple Bayesian EM (BEM) searches from multiple different starting points to couple with MCMC analyses. Each relies heavily on repeatedly recomputed values of the key *conditional configuration probabilities*

$$\pi_j(\mathbf{x}_i) \equiv \Pr(z_i = j|\mathbf{x}_i, \boldsymbol{\Theta}) = \pi_j N(\mathbf{x}_i|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)/g(\mathbf{x}_i|\boldsymbol{\Theta}) \qquad (j = 1:k), \tag{A.1}$$

independently over data points $i = 1:n$, at any parameter value $\boldsymbol{\Theta}$. These are the theoretical classification probabilities for the implicit *configuration indicators* $z_{1:n}$ where $z_i \in 1:k$ is such that $(\mathbf{x}_i|z_i = j, \boldsymbol{\Theta}) \sim N(\mathbf{x}_i|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$.

MCMC steps: Each MCMC iterate has the following steps.

(a) *Recompute configuration probabilities and resample configuration indicators:* Calculate $\pi_j(\mathbf{x}_i)$ for each $i = 1:n$, $j = 1:k$. Then make n independent and individual multinomial draws of size 1, $(z_i|\mathbf{x}_i, \boldsymbol{\Theta}) \sim \mathrm{Mn}(1, \boldsymbol{\pi}_{1:k}(\mathbf{x}_i))$. This reconfigures the n points independently among the k components, and delivers counts $n_j = \#\{z_i = j, i = 1:n\}$ for $j = 1:k$. Note that some components may be empty, with $n_j = 0$ for some j.

(b) *Resample means and covariance matrices:* Draw new $(\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$ independently over $j = 1 : k$ from implied conditional normal, inverse Wishart distributions. Component j has the conjugate posterior based on the currently allocated data, those samples such that $z_i = j$ (and components revert to the prior for cases with $n_j = 0$).

(c) *Resample mixture weights:* For $j = 1 : k - 1$, draw independent beta variates $v_j \sim \text{Be}(1 + n_j, a_j)$ where $a_j = \alpha + \sum_{r=j+1}^{k} n_r$; set $\pi_1 = v_1$ and $\pi_j = v_j \prod_{r=1}^{k-1}(1 - v_r)$ for $j = 2 : k - 1$.

(d) *Resample DP precision:* Draw α from its conditional gamma posterior $p(\alpha|n, \Theta)$ (Ishwaran and James 2001).

Additional, subsidiary computations address model identifiability, a thorny and challenging issue in mixture analysis when full posterior inferences via MCMC are desired. These are secondary issues in the context of the contributions of this article, but we do note that we use novel mixture component relabeling strategies to address this in our applied work (Lin, Chan, and West 2010). This adds to the overall computational burden but is, again, a strategy involving parallelizable steps also ideally suited to GPU computing.

BEM steps: Each BEM iterate has the following steps, involving averaging with respect to the configuration probabilities. Full details are given by Lin, Chan, and West (2010).

(a) *Recompute configuration probabilities:* $\pi_j(\mathbf{x}_i)$ for each $i = 1 : n$, $j = 1 : k$.

(b) *Compute conditional posterior parameters with weighted samples:* For each $j = 1 : k$, compute the parameters of the conditional normal, inverse Wishart posterior for $(\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$ obtained from all n observations using the likelihood function derived from a weighted sampling model with $\mathbf{x}_i \sim N(\mathbf{x}_i|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j/w_{ij})$ and where the weights have "plugged-in" values $w_{ij} = \pi_j(\mathbf{x}_i)$. The posterior means of the resulting normal, inverse Wishart distribution give the values $(\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$.

(c) *Recompute mixture weights:* For $j = 1 : k - 1$, compute

$$v_j = \min\left\{1, \max\left\{0, p_j \Big/ \left(\alpha - 1 + \sum_{r=j+1}^{k} p_r\right)\right\}\right\},$$

where $p_r = \sum_{i=1}^{n} \pi_r(\mathbf{x}_i)$. Set $\pi_1 = v_1$ and $\pi_j = v_j \prod_{r=1}^{k-1}(1 - v_r)$ for $j = 2 : k - 1$. Note, incidentally, that some of the v_j can be zero, reflecting the ability of modal estimates of the model to "cut-back" to fewer than the upper bound k of components.

(d) *Recompute DP precision:* Set α to the mode of the conditional gamma posterior $p(\alpha|n, \boldsymbol{\Theta})$.

SUPPLEMENTAL MATERIALS[1]

Additional Appendices: Document with additional appendix material referred to in the article. This includes *Appendix B, CUDA Storage and Padding for Optimization* and *Appendix C, GPU Parallel Reduction for BEM.* (GPUmixAppendixB+C.pdf)

Code and Data: Containing all source code, with detailed instructions for compilation, linking libraries, and running the analyses. Data from the benchmark analyses in the article, together with details of the parameter input files and resulting analysis output. (GPUmixCodeExamples.tar.gz)

ACKNOWLEDGMENT

We are grateful to the editor and an associate editor for their review and recommendations on the article. Theory underlying our BEM implementation in TDP models is due to Lin Lin of Duke University. Fernando Bonassi of Duke University assisted with web page development for the site *http://www.stat.duke.edu/gpustatsci/*. Research reported here was partially supported by grants from the U.S. National Science Foundation (DMS-0342172) and National Institutes of Health (U54-CA-112952, P50-GM081883, RC1-AI086032, and R01-GM086887). Any opinions, findings and conclusions, or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the NSF.

[Received February 2010. Revised February 2010.]

REFERENCES

Amdahl, G. (1967), "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," *AFIPS Conference Proceedings*, 30, 483–485. [422]

Boedigheimer, M. J., and Ferbas, J. (2008), "Mixture Modeling Approach to Flow Cytometry Data," *Cytometry A*, 73 (5), 421–429. [421]

Chan, C., Feng, F., West, M., and Kepler, T. (2008), "Statistical Mixture Modeling for Cell Subtype Identification in Flow Cytometry," *Cytometry A*, 73, 693–701. [421]

Charalambous, M., Trancoso, P., and Stamatakis, A. (2005), "Initial Experiences Porting a Bioinformatics Application to a Graphics Processor," in *Proceedings of 10th Panhellenic Conference on Informatics (PCI2005). Lecture Notes in Computer Science*, Vol. 3746, New York: Springer-Verlag, pp. 415–425. [420]

Escobar, M., and West, M. (1995), "Bayesian Density Estimation and Inference Using Mixtures," *Journal of the American Statistical Association*, 90, 577–588. [422]

Gustafson, J. L. (1988), "Reevaluating Amdahl's Law," *Communications of the ACM*, 31, 532–533. [422]

Harris, M. (2008), "Optimizing Parallel Reduction in CUDA," available at *http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/reduction/doc/reduction.pdf*. [Accessed on 1/25/2010.] [432]

Herzenberg, L. A., Tung, J., Moore, W. A., Herzenberg, L. A., and Parks, D. R. (2006), "Interpreting Flow Cytometry Data: A Guide for the Perplexed," *Nature Immunology*, 7 (7), 681–685. [421]

Huang, J. (2009), "Keynote Address," in *Hot Chips: A Symposium on High Performance Chips 21*, Stanford, CA: IEEE Technical Committee on Microprocessors and Microcomputers. Available at *http://blogs.nvidia.com/ntersect/2009/08/hot-chips-2009-keynote-by-jen-hsun-huang.html*. [435]

[1]Interested readers can also download the Supplemental Material from *http://www.stat.duke.edu/gpustatsci/* under the *Papers* tab. This web site also contains additional software, including extended versions of the original code with Matlab and R wrappers that create the input files and retrieve the data automatically.

Ishwaran, H., and James, L. (2001), "Gibbs Sampling Methods for Stick-Breaking Priors," *Journal of the American Statistical Association*, 96, 161–173. [422,436]

Ji, C., Merl, D., Kepler, T., and West, M. (2009), "Spatial Mixture Modelling for Unobserved Point Processes: Application to Immunofluorescence Histology," *Bayesian Analysis*, 4, 297–316. [422]

Lee, A., Yan, 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," technical report, Oxford University, Dept. of Statistics. [420]

Lin, L., Chan, C., and West, M. (2010), "Discriminative Information Analysis for Variable Selection in Mixture Modelling," technical report, Duke University, Dept. of Statistical Science. [436]

Lo, K., Brinkman, R. R., and Gottardo, R. (2008), "Automated Gating of Flow Cytometry Data via Robust Model-Based Clustering," *Cytometry A*, 73 (4), 321–332. [421]

MacEachern, S., and Müller, P. (1998a), "Computational Methods for Mixture of Dirichlet Process Models," in *Practical Nonparametric and Semiparametric Bayesian Statistics*, eds. D. Dey, P. Müller, and D. Sinha, New York: Springer-Verlag, pp. 23–44. [422]

——— (1998b), "Estimating Mixture of Dirichlet Process Models," *Journal of Computational and Graphical Statistics*, 7, 223–238. [422]

Manavski, S. A., and Valle, G. (2008), "CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith–Waterman Sequence Alignment," *BMC Bioinformatics*, 9, S10. [420]

NVIDIA-CUBLAS (2008), "NVIDIA CUDA CUBLAS Library 2.0," available at *http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf*. [432]

NVIDIA-CUDA (2008), "NVIDIA CUDA Compute Unified Device Architecture: Programming Guide Version 2.0," available at *http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf*. [420]

Ornatsky, O., Baranov, V., Bandura, D., Tanner, S., and Dick, J. (2006), "Multiple Cellular Antigen Detection by ICP-MS," *Journal of Immunological Methods*, 308 (1–2), 68–76. [421]

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. (2007), "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, 26, 80–113. [420]

Pyne, S., Hu, X., Wang, K., Rossin, E., Lin, T.-I., Maier, L. M., Baecher-Allan, C., McLachlan, G. J., Tamayo, P., Hafler, D. A., Jager, P. L. D., and Mesirov, J. P. (2009), "Automated High-Dimensional Flow Cytometric Data Analysis," *Proceedings of the National Academy of Sciences of the USA*, 106 (21), 8519–8524. Available at *http://www.pnas.org/content/106/21/8519.long*. [421]

Silberstein, M., Schuster, A., Geiger, D., Patney, A., and Owens, J. D. (2008), "Efficient Computation of Sum-Products on GPUs Through Software-Managed Cache," in *Proceedings of the 22nd Annual International Conference on Supercomputing*, ed. P. Zhou, New York: ACM, pp. 309–318. [420]

Suchard, M. A., and Rambaut, A. (2009a), "BEAGLE: Broad-Platform Evolutionary Analysis General Likelihood Evaluator," available at *http://beagle-lib.googlecode.com*. [420]

——— (2009b), "Many-Core Algorithms for Statistical Phylogenetics," *Bioinformatics*, 25, 1370–1376. [420]

Teh, Y., Jordan, M., Beal, M., and Blei, D. (2006), "Hierarchical Dirichlet Processes," *Journal of the American Statistical Association*, 101, 1566–1581. [422]