GPU Parallelization of Gibbs Sampling – Abstractions, Results, and Lessons Learned

Alireza S Mahani
Scientific Computing Group
Sentrana Inc.

May 16, 2012

Objectives of This Talk

- **What This Talk Is About**

1. Useful parallelization abstractions for Gibbs sampling
2. CUDA implementation and speedup results for the above abstractions
3. Discussion of broader lessons for GPU programmers in particular, and scientific computing community in general

- **What This Talk Is NOT About**

1. Detailed discussion of Monte Carlo Markov Chain (MCMC) sampling theory
2. Application of GPU Performance optimization strategies to Gibbs sampling parallelization
3. Unequivocal endorsement of a particular hardware/software parallelization platform

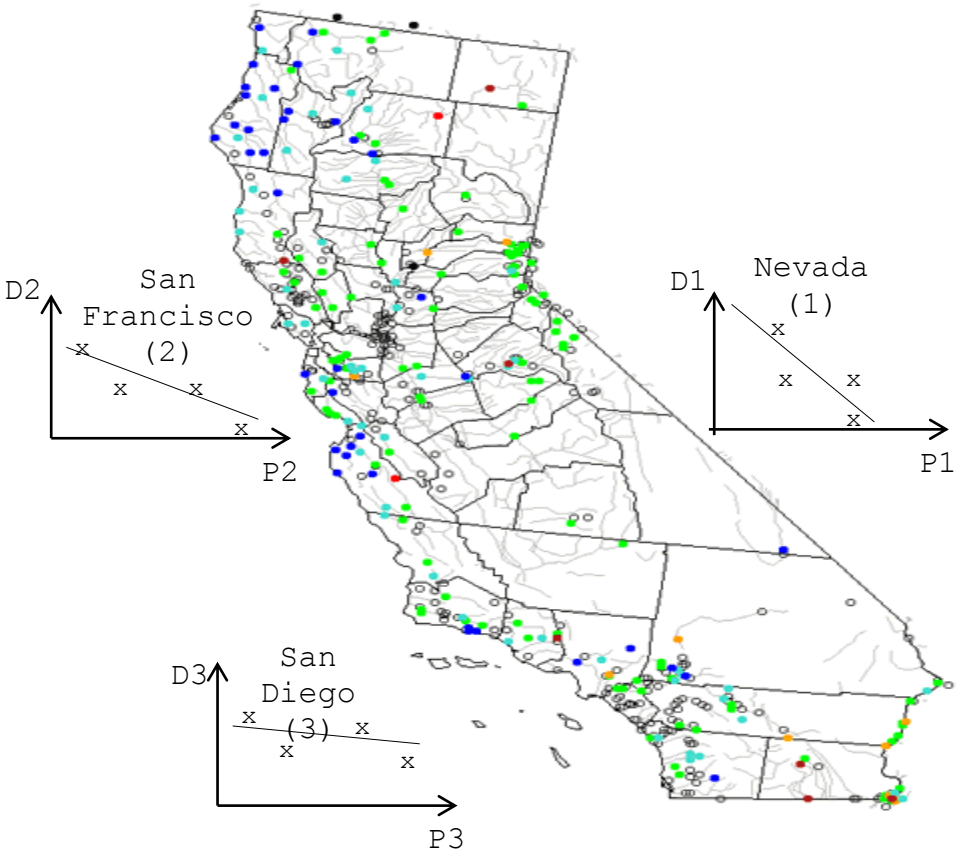
Table of Contents

Background

- **Academic/programming background:**
  - Ph.D. in Physics / Computational Neuroscience
  - Used MATLAB for thesis
  - Entered programming world 3 years ago, starting with C#
  - Gradually transitioned to C++ and C
  - First serious exposure to GPU programming at GTC 2010
  - Began CUDA programming in March 2011

- **My work:**
  - Our company (Sentrana) builds quantitative marketing software, with a focus on Foodservice industry
  - Our headcount is ~50, with ~25 business application software developers, 3 full-time modelers
  - Scientific computing is transitioning from a one-man show to a separate department

Hierarchical Bayesian Models in Quantitative Marketing



Why Hierarchical Bayesian models in Quantitative Marketing?

1. Graceful handling of *data sparseness*, without loss of *heterogeneity*

2. Ability to incorporate human knowledge into the model (*informative priors*)

Domain Implications:

- Need for fast sampling algorithms
- Model specifications are in flux
- Problem dimensions are unpredictable

Introduction to Gibbs Sampling (1/2)

- In a Bayesian framework, model estimation can be viewed as sampling from the posterior distribution:

$$P(\theta|X) \propto P(X|\theta)P(\theta)$$

posterior likelihood prior

X: observations (given)

θ: parameters (to be estimated)

- Sampling: Drawing a finite set of points from the posterior to approximate its moments:

$$E[f(\theta)] = \int f(\theta)P(\theta|X)d\theta \sim < f(\theta_i) >$$

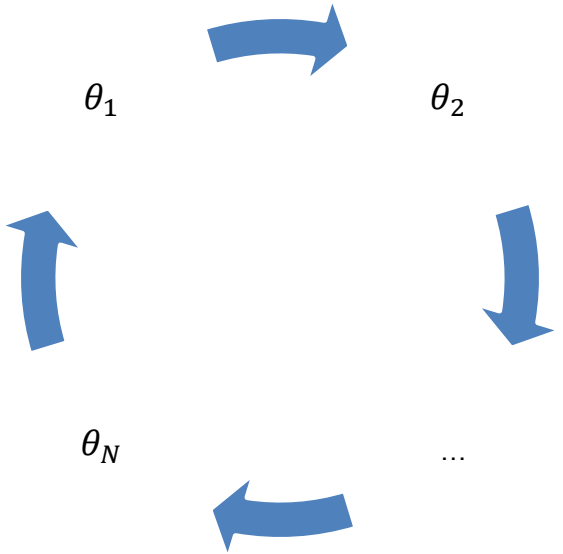
- Two challenges in sampling:
  - It is difficult to draw independent samples from complex posterior distributions (even in 1D)
  - In real-world problems, parameter space is high-dimensional; drawing samples from high-dimensional posteriors is even more challenging
- Gibbs sampling helps us turn a high-dimensional problem into a series of 1-D problems

Introduction to Gibbs Sampling (2/2)

- Gibbs sampling is a special case of Metropolis Hastings algorithm with acceptance rate of 100%
- It involves cycling through parameters, one at a time, and drawing samples from their posterior conditioned on current value of all remaining variables

Gibbs Sampling Algorithm*:

1. Initialize θ: $\theta^1 = (\theta_1^1, \theta_2^1, \dots, \theta_N^1)$
2. For $\tau = 1, \dots, T$:
   - Draw θ_1: $\theta_1^{\tau+1} \sim P(\theta_1 | \theta_2^\tau, \theta_3^\tau, \dots, \theta_N^\tau)$
   - Draw θ_2: $\theta_2^{\tau+1} \sim P(\theta_2 | \theta_1^{\tau+1}, \theta_3^\tau, \dots, \theta_N^\tau)$
   - …
   - Draw θ_j: $\theta_j^{\tau+1} \sim P\left(\theta_j | \theta_1^{\tau+1}, \dots, \theta_{j-1}^{\tau+1}, \theta_{j+1}^\tau, \dots, \theta_N^\tau\right)$
   - …
   - Draw θ_N: $\theta_N^{\tau+1} \sim P(\theta_N | \theta_1^{\tau+1}, \theta_2^{\tau+1}, \dots, \theta_{N-1}^\tau)$



- Gibbs sampling concept can be extended to variable blocks (i.e. when one or more of θ components are arrays themselves

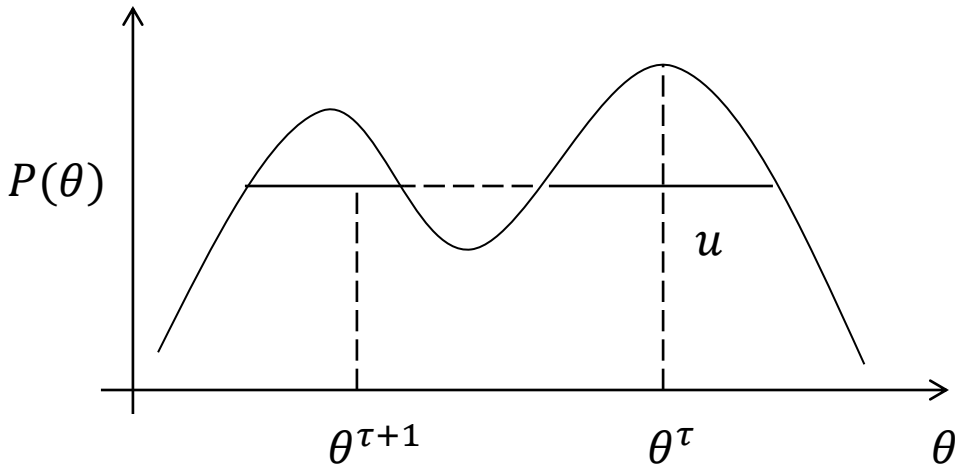
* Adapted from *Pattern Recognition and Machine Learning* by Bishop, C.M.

Introduction to Univariate Slice Sampler

- Since Gibbs sampling reduces a high-D sampling problem to a series of 1-D sampling problems, having a high-quality, versatile univariate sampler becomes essential to our HB estimation toolkit
- Univariate *slice sampler** is a generic, robust, and near-automatic algorithm

Univariate Slice Sampler:

1. Draw u from $\text{unif}[0, P(\theta^{\tau})]$
2. Define horizontal slice $S = \{\theta^{*} | P(\theta^{*}) \geq u\}$
3. Draw $\theta^{\tau+1}$ from a uniform distribution on S



- As with other sampling (and optimization) algorithms, the majority of computation time during slice sampling is devoted to function evaluation

* Neal R.M. (2003). *Annals of Statistics*, **31**, 705-767.

Single-Chain Parallelization of Gibbs Sampling

- There are several parallelization angles not considered here*, mainly the use of multiple chains; here we focus in two single-chain strategies:

1. **Block-level Parallelization**

   — Available when multi-D conditional posterior for a parameter block is separable over individual elements:

$$P(\theta|-) = \prod_i P(\theta_i|-)$$

   — In this case, each parameter θ_i can be sampled independently of other parameters within the block, $\theta_i, j \neq \mathrm{i}$.

   — Block-level parallelization is relatively coarse, and hence more efficient; however, in many cases, it simply isn't available

   — Historically, Gibbs sampling was introduced into the statistical community as 'parallel Gibbs', wherein it was mistakenly assumed that block-level parallelization works in general

2. **Log-Posterior Parallelization**

   — When observations are i.i.d. (independently and identically distributed), log-likelihood consists of sum of contributions from each observation point:

$$X = \{X_1, X_2, \ldots, X_M\}$$
$$\log P(\theta|X) = \log P(X|\theta) + \log P(\theta)$$
$$\log P(\theta|X) = \sum_i \log P(X_i|\theta) + \log P(\theta)$$

   — This parallelization is finer-grained relative to the first type and hence less efficient, but tends to be more prevalent in statistical modeling

* For a complete discussion, see *Parallel Bayesian Computation* by Wilkinson, D.J. (Chapter 16 in *Handbook of Parallel Computing and Statistics* (editor Kontoghiorghes E.J.)

Taxonomy of Parameter Blocks

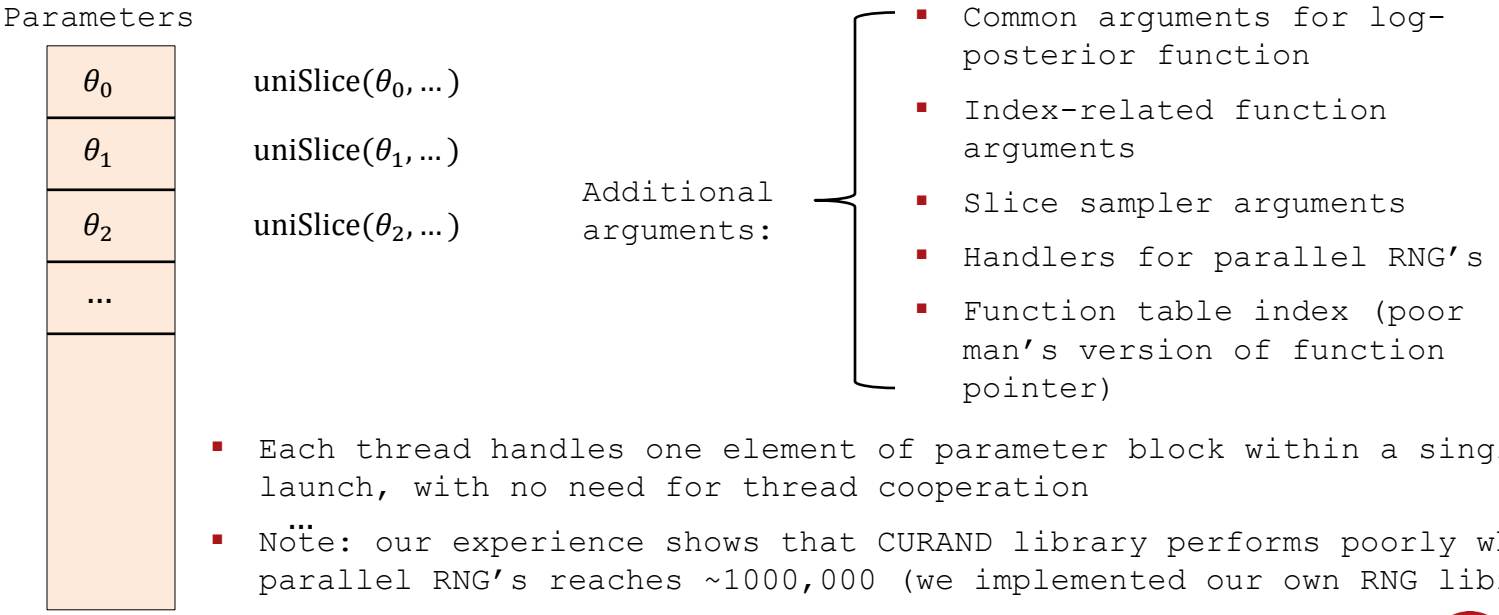
- All permutations of block-level and log-posterior parallelization are possible, leading to four types of parameter blocks:
  1. *Block-serial, log-posterior-serial (BS-LPS)* → No parallelization opportunity
  2. *Block-parallel, log-posterior-serial (BP-LPS)* → Single-faceted, easy to implement, high modular speedup, low overall impact
  3. *Block-serial, log-posterior-parallel (BS-LPP)* → Single-faceted, modestly difficult to implement, modest modular speedup, good overall impact
  4. *Block-parallel, log-posterior-parallel (BP-LPP)* → dual-faceted, difficult to implement, high modular speedup, high overall impact

- Organization of GPU into blocks and threads naturally maps to dual-faceted parallelization for BP-LPP parameter blocks

- However, taking full advantage of the available silicon on the GPU requires significant code reorganization

GPU Parallelization of BP-LPS Parameters

- Implement univariate slice sampler as device function
- Implement log-posterior functions as device functions as well
- Use a function table to pass log-posterior function to slice sampler (crude substitute for function pointers in C)
- A kernel launches multiple instances of slice sampler, one for each parameter in the block



- Each thread handles one element of parameter block within a single kernel launch, with no need for thread cooperation
- Note: our experience shows that CURAND library performs poorly when number of parallel RNG's reaches ~1000,000 (we implemented our own RNG library)

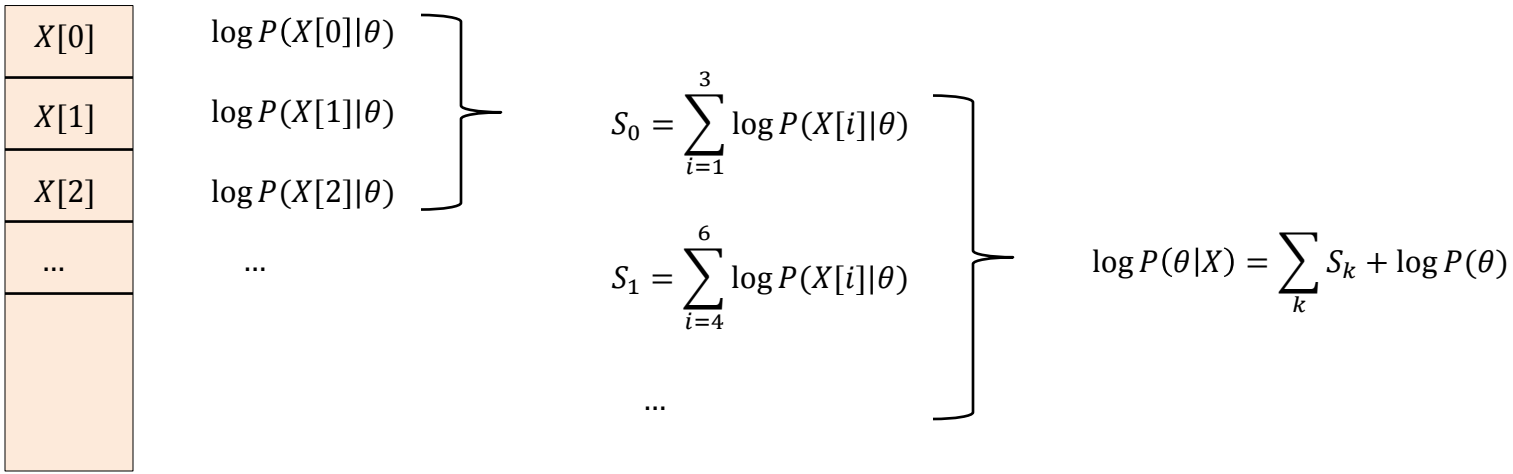
GPU Parallelization of BS-LPP Parameters

- Sampling of elements within the block must happen serially
- Parallelization occurs while evaluating log-posterior function for a single element (due to i.i.d. nature of observed data)
- We employ a two-stage scatter-gather approach for GPU parallelization

Step 1: Scatter + Partial Gather
(executed on device)

Step 2: Final Gather
(some or all executed on host)

Observations



GPU Parallelization of BP-LPP Parameters (1/4) – Vectorization of Slice Sampler

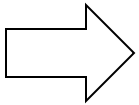
- We can parallelize at block level as well as log-posterior level
- CUDA implementation is more involved than first two cases

Step 1: Create a vectorized version of slice sampler

Standard Slice Sampler

```
typedef double (*Func)(
            double x
            , const void *args
            );

double uniSlice(
            Func f
            , double x0
            , const void *args
            , SliceArg &sliceArgs
            );
```



Vectorized Slice Sampler

```
typedef void (*FuncPar)(
            int n,
            const double *xVec
            , const void **argsVec
            , const bool *selectVec
            , double *fVec
            );

void uniSlicePar(
            FuncPar f
            , int n
            , const double *x0Vec
            , const void **argsVec
            , SliceArg &sliceArgs
            , double *xVec
            );
```

- Vectorized slice sampler takes advantage of a parallel function evaluator (FuncPar), with each of *n* elements corresponding to one conditionally-independent parameter in the block
- Since each parameter may need a different number of iterations before acceptance, we can keep track by using *selectVec* and use parallel hardware more efficiently

GPU Parallelization of BP-LPP Parameters (2/4) – Parallel Function Evaluation

Step 2: Implement a kernel for 2-level parallelization of log-posterior evaluation for a parameter block

- Each parameter is assigned to a GPU block; conditional independence means there is no need for inter-block communication → no need for multiple kernel launches
- Within each block, each thread handles one observation point; thread cooperation allows these contributions to be summed up

Block i, thread j:

$$q_{ij} = \log P(X_j = x_j | \theta_i = \theta_i^{\tau}, \theta_{-i} = \theta_{-i}^{\tau-1}, \ldots)$$

Within-block summation:

$$Q_i = \sum_j q_{ij} + \log P(\theta_i = \theta_i^{\tau})$$

- $Q_i's$ are returned to the host as elements of *fVec* (see previous slide)

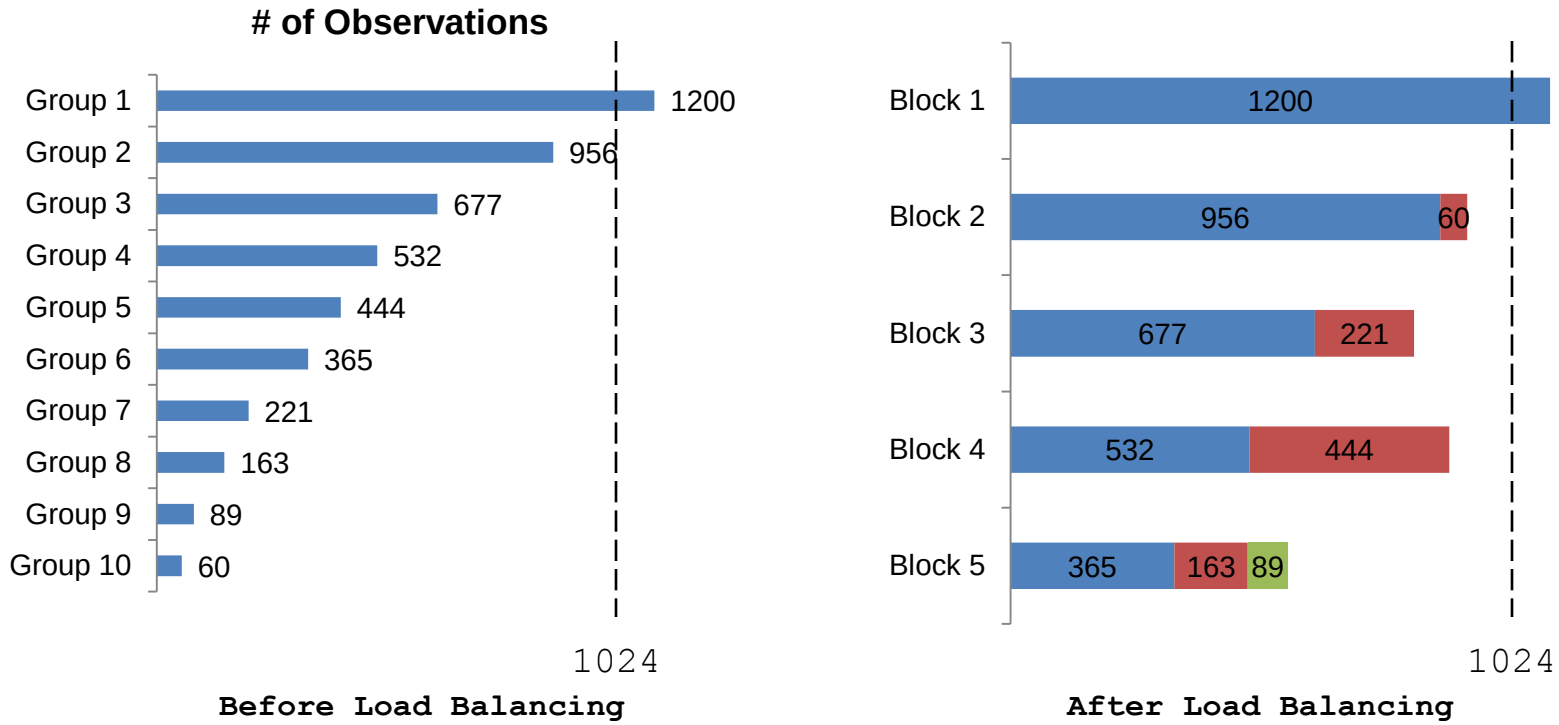
Step 3: Plumbing … lots and lots!

- We can use function tables and void pointers to abstract the algorithm as much as possible, but it won't look as elegant as the host-side code
- Need for efficiency means some arrays have to be pre-allocated outside a function, despite the fact that they have no use on the outside

GPU Parallelization of BP-LPP Parameters (3/4) – Non-uniform Blocks

- In many statistical applications, conditionally-independent parameter blocks correspond to non-uniform regression groups, i.e. groups with different number of observation points
- In fact, handling data sparseness, i.e. small groups, by borrowing information from larger groups is a primary focus of HB modeling
- Our naïve GPU implementation of BP-LPP parameters, while mapping naturally to GPU structure, is quite inefficient:
  - If # of threads per block is set according to largest group, silicon is wasted for small groups
  - If # of threads per block is set according to smallest group, large groups end up with serial calculation (need to fold the data over)
- We therefore need 'load balancing' on the GPU, with two objectives:
  - Rearrange calculations across blocks to make them as evenly distributed as possible
  - Avoid the need for inter-block communication to avoid the need for multiple kernel launches
- Solution:
  1. Select N, number of threads per block (typically at its maximum of 1024 for 2.x devices due to step 2)
  2. For groups that have more observations than N, fold them over
  3. For groups of size less than N, perform simple load balancing to arrange them into blocks such that their total # of observations is less than N
  4. Assign the blocks created in 3 onto GPU blocks

GPU Parallelization of BP-LPP Parameters (4/4) – Non-uniform Blocks (cont.)



- For our data sets, load balancing increased speed by a factor of 2-3
- All code optimization routes were not explored, as there is a tradeoff between performance and code development and maintenance costs

Speedup Results

Parameter Block	Problem Spec	Serial-CPU	OpenMP (8 cores)	CUDA
BP-LPS (block parallelization)	▪ Block size: ~4 million ▪ CPU execution time for parallelized task: ~7sec	1.0x	~3x	~100x
BS-LPP (log-posterior parallelization)	▪ Number of observations: 6390 ▪ CPU execution time for parallelized task: ~7msec	1.0x	~5x	~23x
BP-LPP (dual parallelization)	▪ Number of groups: 500 ▪ Number of observations per group: 500 ▪ CPU execution time for parallelized task: ~2sec	1.0x	~6x	~100x

- Details:
  - CPU: Intel Xeon W3680, 3.33GHz; 12 cores; 2GB/core RAM (8 cores used for OpenMP since more than 8 reduced performance); code compiled with –O2 flag
  - GPU: GeForce GTX 580 (*all GPU code was double-precision; fast math option turned on*)
  - Measurements include: per sampling (recurring) data transfer between host and device
  - Measurements exclude: initial (one-time) data transfer between host and device
- Other tests:
  - BP-LPP MPI-parallelization was tested on a commercial cluster; we saw speedups of ~50-60x relative to single-threaded code (on same cluster), saturating at ~16 nodes / 128 cores

Total Cost Analysis

Total Model Cost = Development Cost + Execution Cost

- Development cost
  - Time and expertise needed to specify HB model (hierarchy definition, variable selection / transformation, etc) → iterative, requires building, running, and inspecting early versions
  - Time and expertise needed to convert high-level, low-performance code (e.g. in R) to low-level, high-performance code (C/C++/CUDA); includes debugging/testing
  - Code maintenance
- Execution cost
  - Wall-time needed to estimate model coefficients (i.e. run Gibbs sampling)
  - Total core-hours

- How does CUDA stack up against alternatives on these two metrics?
  - Development cost
    - By reducing execution time, it allows modelers to iterate on models faster → trial-and-error (to some degree) replaces mathematical expertise
    - However, it shifts the burden from mathematical and statistical expertise to programming time and expertise (C is harder than R; CUDA is harder than C)
  - Execution cost
    - Not only does it reduce wall-time, but it also is cheaper (our calculations showed that porting our code from CPU to GPU leads to savings of ~5-10x)
    - On the other hand, maintaining in-house GPU servers adds to required skill set for system administrators
    - In comparison to MPI and Intel's Vector Math Library, CUDA requires about as much code modification (if not less) and the return on investment is higher

Lessons Learned

- Suggestions for Modelers / Developers / Rookie GPU Programmers
  1. Be goal-oriented and means-agnostic (too much expertise might create bias)
  2. Don't fall in love with performance (poorly-written, super-fast code is doomed for failure)
  3. Seek opportunities for abstraction (create libraries, design abstract interfaces, etc) proportional to model volatility
  4. Consider a software development path (e.g. we use R->C++->OpenMP->CUDA) that facilitates testing/debugging
  5. Learn about available tools (debuggers, profilers, memory checks, etc.)
  6. Be prepared to learn about hardware as needed
  7. Avoid common CUDA pitfalls
     - Attempting/assuming block cooperation within kernels
     - Synchronization within divergent threads
     - Missing thread synchronization
     - Invalid memory access (e.g. negative array indexes)

- Suggestions for Nvidia
  1. Publish/advertise a list of common CUDA programming mistakes
  2. Improve abstraction capabilities, e.g. full support for function pointers
  3. Fix double-precision volatility of GTX cards!!