Statistical computation on GPGPUs

Jarad Niemi
Matthew Wheeler

Iowa State University

28 Sep 2011

Outline

- Results
- Parallelism
  - General
  - Statistics
    - Optimization (MLEs)
    - Integration (Posteriors)
- GPGPUs
  - What?
  - How?
- An example - stochastic chemical kinetic models (skip?)
- Wrap-up
  - Parallelism coming to the end user?

On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods

Anthony LEE, Christopher YAU, Michael B. GILES,
Arnaud DOUCET, and Christopher C. HOLMES

Table 1. Running times for the population-based MCMC sampler for various numbers of chains M.

M	CPU (min)	8800 GT (sec)	Speedup	GTX 280 (sec)	Speedup
8	0.0166	0.887	1.1	1.083	0.9
32	0.0656	0.904	4	1.098	4
128	0.262	0.923	17	1.100	14
512	1.04	1.041	60	1.235	51
2048	4.16	1.485	168	1.427	175
8192	16.64	4.325	230	2.323	430
32,768	66.7	14.957	268	7.729	527
131,072	270.3	58.226	279	28.349	572

On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods

Anthony LEE, Christopher YAU, Michael B. GILES,
Arnaud DOUCET, and Christopher C. HOLMES

Table 2. Running times for the sequential Monte Carlo sampler for various values of N.

N	CPU (min)	8800 GT (sec)	Speedup	GTX 280 (sec)	Speedup
8192	4.44	1.192	223.5	0.597	446
16,384	8.82	2.127	249	1.114	475
32,768	17.7	3.995	266	2.114	502
65,536	35.3	7.889	268	4.270	496
131,072	70.6	15.671	270	8.075	525
262,144	141	31.218	271	16.219	522

On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods

Anthony LEE, Christopher YAU, Michael B. GILES,
Arnaud DOUCET, and Christopher C. HOLMES

Table 3. Running time (in seconds) for the sequential Monte Carlo method for various values of N.

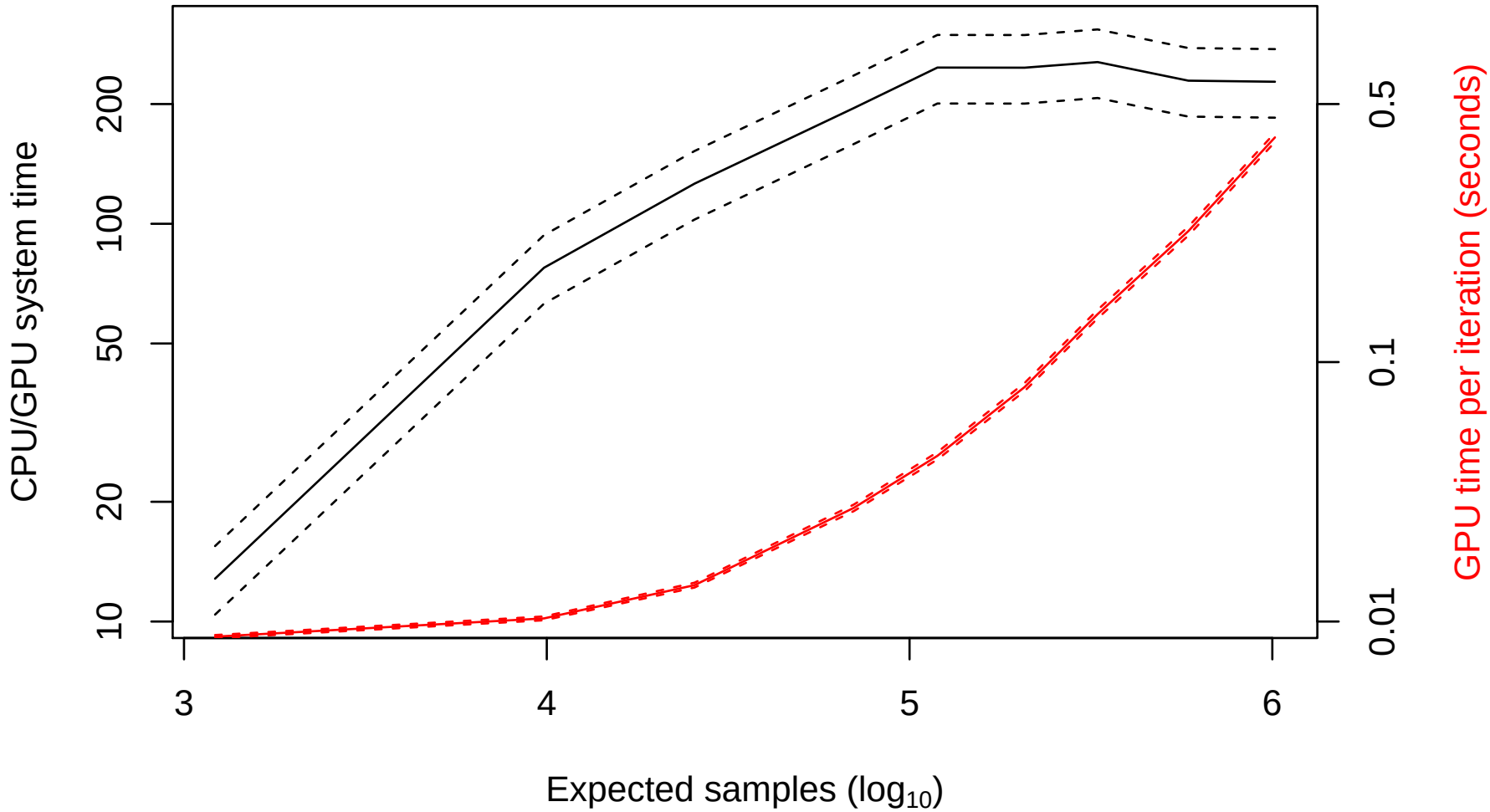
N	CPU	8800 GT	Speedup	GTX 280	Speedup
8192	2.167	0.263	8	0.082	26
16,384	4.325	0.493	9	0.144	30
32,768	8.543	0.921	9	0.249	34
65,536	17.425	1.775	10	0.465	37
131,072	34.8	3.486	10	0.929	37

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

Marc A. Suchard[1], Quanli Wang[2,5], Cliburn Chan[3], Jacob Frelinger[4],
Andrew Cron[5] & Mike West[5]

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	1,907.0	169.7	2,758.0
10^6	192.0	119.5	224.6	3,797.0	19,048.0	1,118.1	27,529.0
5×10^6	954.0	591.0	1,116.0	17,667.0	95,283.0	5,785.8	141,191.0

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


CPU/GPU system time
10
20
50
100
200
GPU time per iteration (seconds)
0.01
0.1
0.5
3
4
5
6
Expected samples ($\log_{10}$)

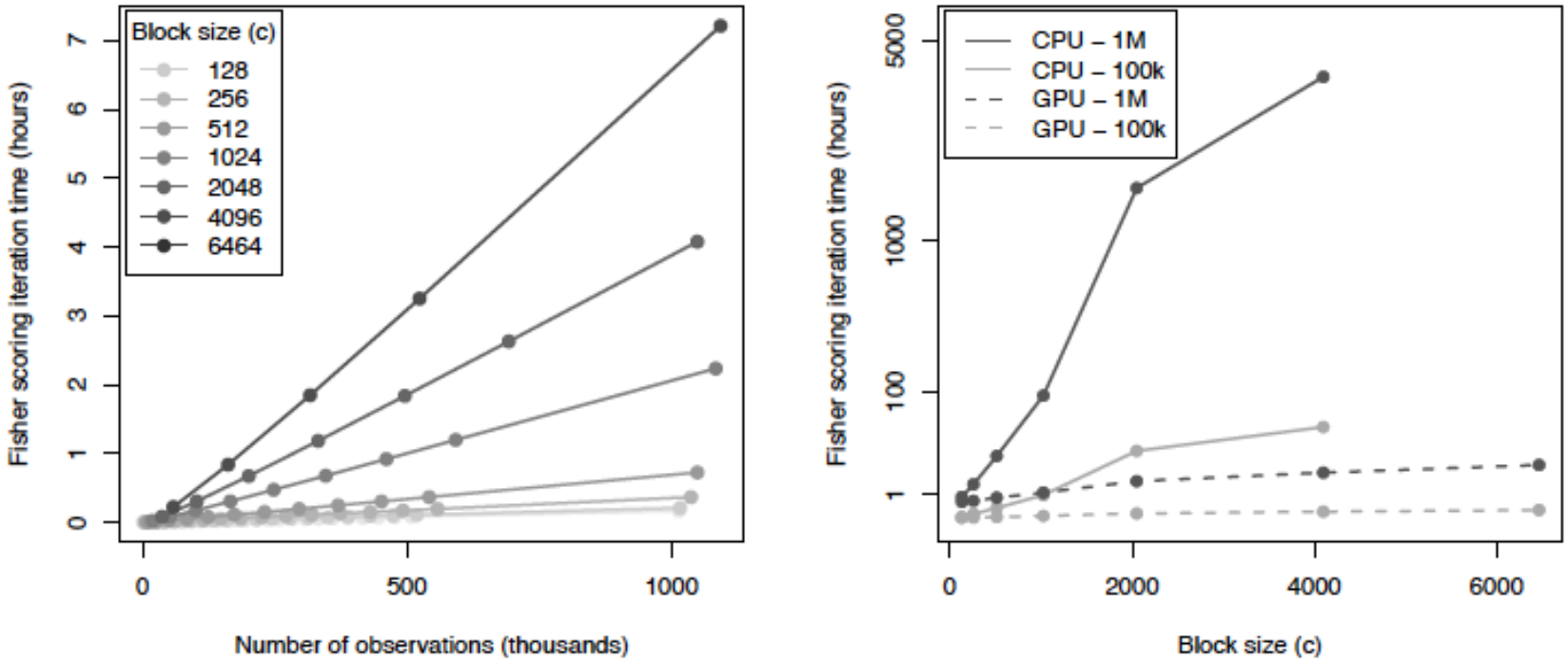


Fig. 4. (a) Computation times for a single Fisher scoring iteration on a NVIDIA C2050 GPU as a function of data size for a variety of block sizes. (b) Computation times, plotted on a cube-root scale, on both the CPU and GPU for a single Fisher scoring iteration as a function of block size for specific data sizes.

The speedup was 1.4-fold at c = 128, 13-fold at c = 512, 29-fold at c = 1024, and 112-fold at c = 4096.

Theoretical maximum speedup, Amdahl's quantity:

$$\frac{1}{1 - P + \frac{P}{N}}$$

P: fraction of the program that can be parallelized

N: number of parallel processors

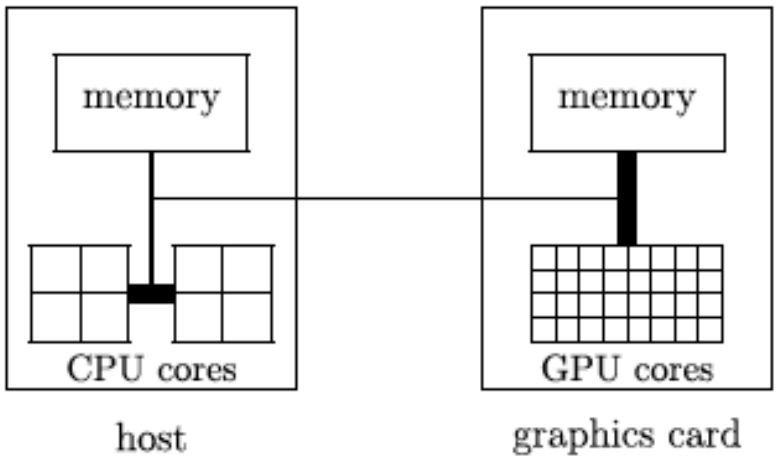
As $N \to \infty$:

$$\frac{1}{1 - P}$$

So if 99% of the program can be parallelized, theoretically we could have a 100-fold speedup.

What's N?

- CPU
  - Clusters: multiple CPUs/memory networked (CyBlue: N=1-2k)
  - Multicores: 2-6 CPUs with shared memory (N=2-6)
- GPU
  - Single card: many-core with own, shared memory attached to CPU (N=448)
  - Cluster: multiple GPUs with CPU(s) (N=4x448)



From Lee et. al.

FLOPS, a measure of performance: CyBlue: 5.7TF, Tesla C2050: 1.0TF

What's P?

Flynn's taxonomy

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

- SISD: no parallelism
- SIMD: good for CPU/GPU parallelism
- MISD: fault tolerance, e.g. Space Shuttle flight controller
- MIMD: possible for CPU/GPU parallelism (edge to CPUs)

From Wikipedia: `http://en.wikipedia.org/wiki/Flynn's_taxonomy`

Multiplying matrices

Let

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

and you are interested in $C = AB$.

The SISD calculation:

1. $c_{11} = a_{11}b_{11} + a_{12}b_{21}$
2. $c_{12} = a_{11}b_{12} + a_{12}b_{22}$
3. $c_{21} = a_{21}b_{11} + a_{22}b_{21}$
4. $c_{22} = a_{21}b_{12} + a_{22}b_{22}$

What's P?

If square matrices of size n, then repeat n^2-times

- n products
- $n - 1$ sums

Intuitively, as $n \to \infty$ then $P \to 1$.

Parallel tempering

- Run M independent MCMC chains at various temperatures, i.e. flattenings of the likelihood.
- At the end of each MCMC iteration, swap states in adjacent chains with the appropriate probability.
- Only take samples from the unflattened chain.

As $M \to \infty$, $P \to 1$.

Sequential Monte Carlo

- Have N weighted particles, approximating a posterior distribution.
- After collecting new data, each particle updates independently.
- Particles are reweighted.

As $N \to \infty$, $P \to 1$.

Latent variable modeling

- Each observation has an associated latent variable.
- One step in the MCMC is to independently update each observations latent variable.

As data size $\to \infty$, $P \to 1$.

Rejection sampling

- Attempting to sample from $p(\cdot)$.
- Sample $x \sim q(\cdot)$, accept with probability $p(x)/Mq(x)$.

As $p(x)/Mq(x) \to 0$, $P \to 1$.

Spatial analysis

- Split data (size n) into blocks of size c
- Calculate inverses and determinants of pairwise block covariances

Two opportunities for parallelization:

- $c \to \infty, n \to \infty, \implies P \to 1$, but cannot load into memory!
- $c \to 0, n \to \infty, \implies$ number of pairwise comparisons $\to \infty$, $P \to 1$.

Since blocks are an approximation, choose largest possible c and use GPU to calculate inverses and determinants.

- Lots of opportunity for parallelism
  - data size increases
  - approximating samples increases
  - number of chains increases
    (very large?, suitable for CPU parallelization)

- Reduced computing time is on the order of 200
  - 6 months becomes 1 day
  - 1 day becomes 7 minutes
  - 1 minute becomes a fraction of a second

General purpose graphical processing units (GPGPUs)

NVIDIA Tesla C2075
Ebay: $1750
Cores: 448
Memory: 6GB
* Higher fault tolerance *

NVIDIA GeForce GTX 590
Ebay: $840
Cores: 1024
Memory: 3GB
* No faults detected *

** Require PCI-E 2.0 x16 bus **

Coding in C

```
#include <iostream>

int main ( void ) {
  printf( "Hello, World!\n" );
  return 0;
}
```

Coding in CUDA C

```
#include <iostream>

__global__ void kernel ( void ) {
}

int main ( void ) {
  kernel<<<1,1>>>();
  printf( "Hello, World!\n" );
  return 0;
}
```

```
#include "../common/book.h"

# define N 10

int main( void ) {
  int a[N], b[N], c[N];
  int *dev_a, *dev_b, *dev_c;

  // allocate memory on the GPU
  HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

  // fill the arrays 'a' and 'b' on the CPU
  for (int i=0; i<N; i++) {
    a[i] = -i;
    b[i] = i * i;
  }

  // copy the arrays 'a' and 'b' to the GPU
  HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice ) );
  HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice ) );

  add<<<N,1>>>( dev_a, dev_b, dev_c);

  // copy array 'c' back from the GPU to the CPU
  HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );

  // display the results
  for (int i=0; i<N; i++) { printf( "%d + %d = %d\n", a[i], b[i], c[i] ); }

  // free memory allocated on the GPU
  cudaFree( dev_a );
  cudaFree( dev_b );
  cudaFree( dev_c );
```

Coding in CUDA C

The GPU kernel:

```
__global__ void add( int *a, int *b, int *c ) {
  int tid = blockIdx.x;     // handle the data at this index
  if (tid < N)
    c[tid] = a[tid] + b[tid];
}
```

Coding in CUDA C

SIMD: each block acts on its own data

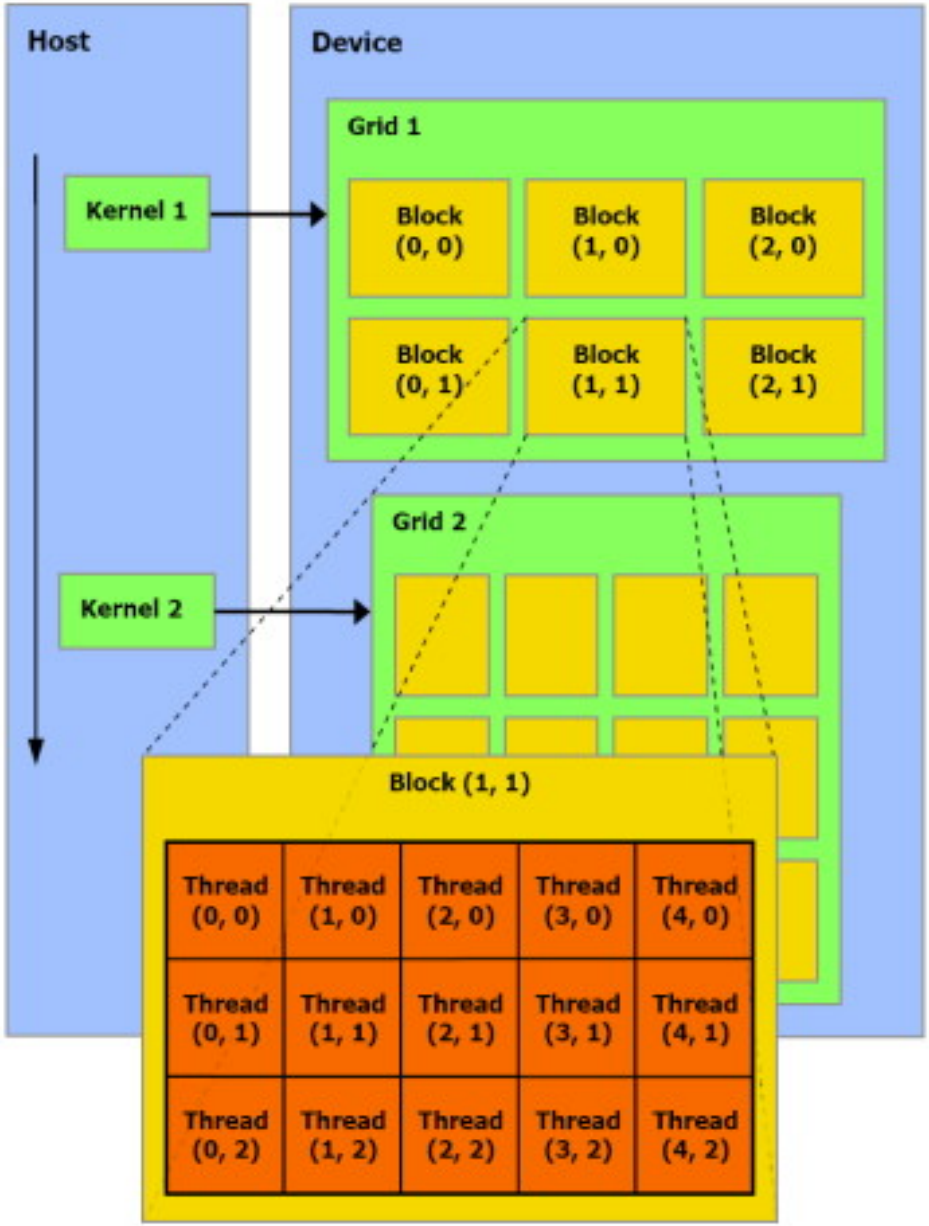
The code each block "sees":

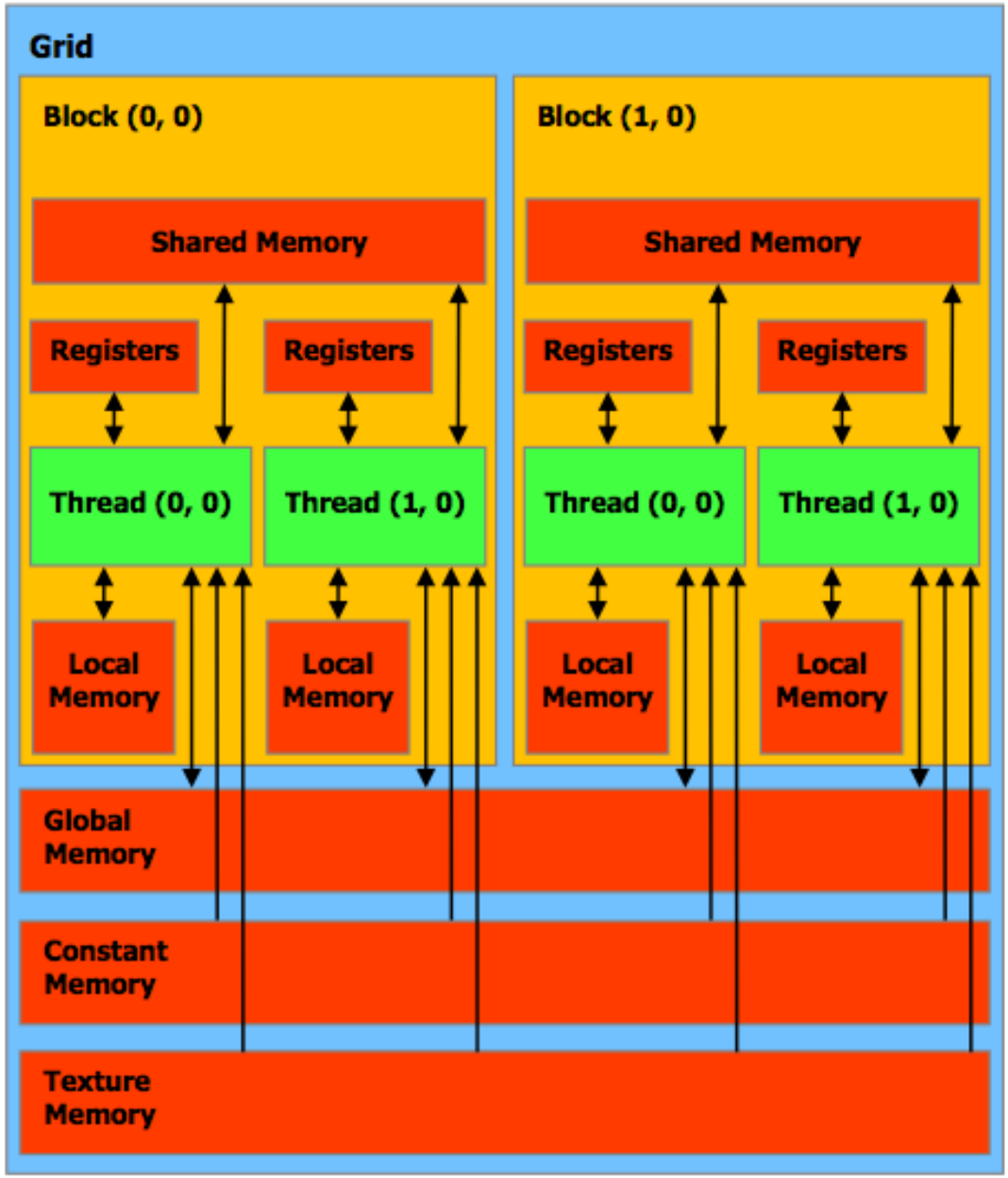
Block 1

```
__global__ void add( int *a, int *b, int *c ) {
  int tid = 0;     // handle the data at this index
  if (tid < N)
    c[tid] = a[tid] + b[tid];
}
```

Block 2

```
__global__ void add( int *a, int *b, int *c ) {
  int tid = 1;     // handle the data at this index
  if (tid < N)
    c[tid] = a[tid] + b[tid];
}
```


Host
Device
Grid 1
Kernel 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Kernel 2
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

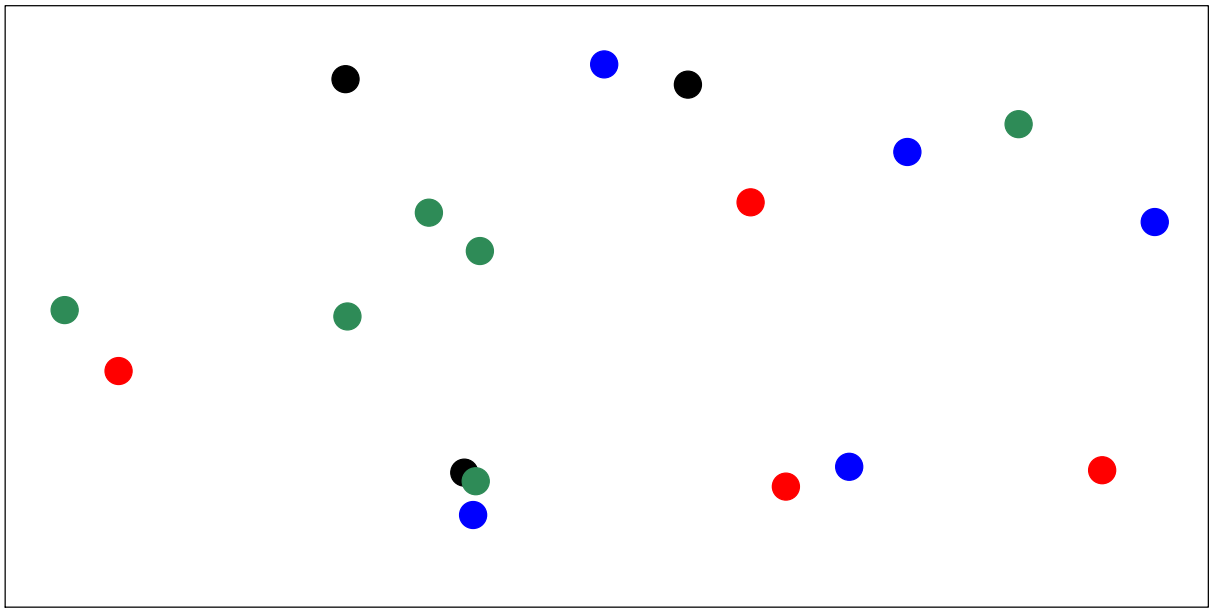

Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Local Memory
Local Memory
Local Memory
Local Memory
Global Memory
Constant Memory
Texture Memory

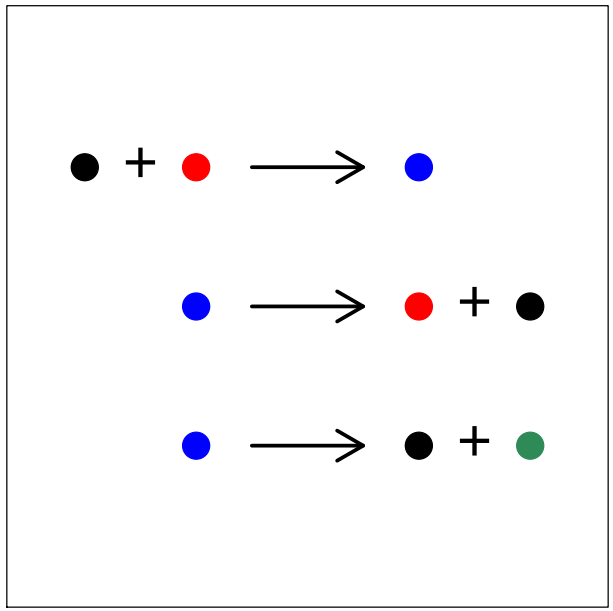
Device 0: "Tesla T10 Processor"		
CUDA Driver Version:	3.10	
CUDA Runtime Version:	3.10	
CUDA Capability Major revision number:	1	
CUDA Capability Minor revision number:	3	
Total amount of global memory:	4294770688	bytes
Number of multiprocessors:	30	
Number of cores:	240	
Total amount of constant memory:	65536	bytes
Total amount of shared memory per block:	16384	bytes
Total number of registers available per block:	16384	
Warp size:	32	
Maximum number of threads per block:	512	
Maximum sizes of each dimension of a block:	512 x 512 x 64	
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1	
Maximum memory pitch:	2147483647	bytes
Texture alignment:	256	bytes
Clock rate:	1.44	GHz
Concurrent copy and execution:	Yes	
Run time limit on kernels:	No	
Integrated:	No	
Support host page-locked memory mapping:	Yes	
Compute mode:	Default (multiple host threads can use this device simultaneously)	

Wrap-up

Imagine a *well-mixed* system in *thermal equilibrium* with

- N species: $S_1, \ldots, S_N$ with
- number of molecules $X_1, \ldots, X_N$ with elements $X_j \in \mathbb{Z}^+$
- which change according to M reactions: $R_1, \ldots, R_M$ with
- propensities $a_1(x), \ldots, a_M(x)$.
- The propensities are given by $a_j(x) = \theta_j h_j(x)$
- where $h_j(x)$ is a known function of the system state.
- If reaction j occurs, the state is updated by the stoichiometry ν_j with
- elements $\nu_{ij} \in \{-1, 0, 1\}$.



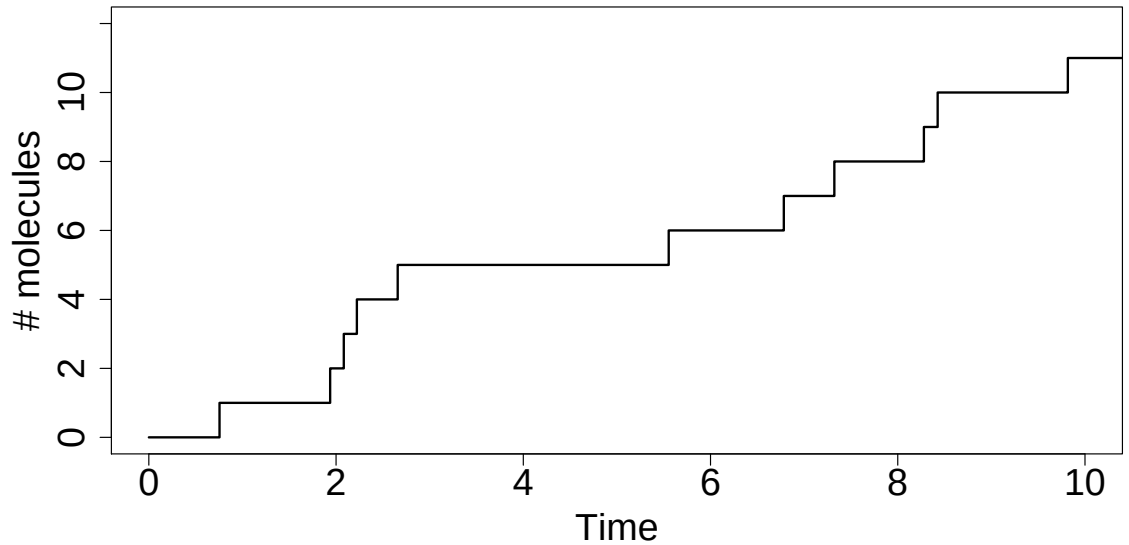


- If reaction $j \in \{1, \ldots, M\}$ has the following probability

$$\lim_{dt \to 0} P(\text{reaction } j \text{ within the interval } (t, t + dt) | X_t) = a_j(X_t)dt,$$

  then this defines a continuous-time Markov jump process.

- Then a realization from this model can be obtained using the Gillespie algorithm:
  - For $j \in \{1, \ldots, M\}$, calculate $a_j(X_t)$.
  - Calculate $a_0(X_t) = \sum_{j=1}^{M} a_j(X_t)$.
  - Simulate a reaction time $\tau \sim Exp(a_0(X_t))$
  - Simulate a reaction id $k \in \{1, \ldots, M\}$ with probability $a_k(X_t)/a_0(X_t)$



Suppose you observe all system transitions:

- n reactions occur in the interval $[0, T]$
- $t_1, \ldots, t_n$ are the reaction times
- $r_1, \ldots, r_n$ are the reaction indicators, $r_i \in \{1, \ldots, M\}$

Then inference can be performed based on the likelihood

$$L(\theta) \propto \prod_{j=1}^{M} \theta_j^{n_j} \exp\left(-\theta_j I_j\right)$$

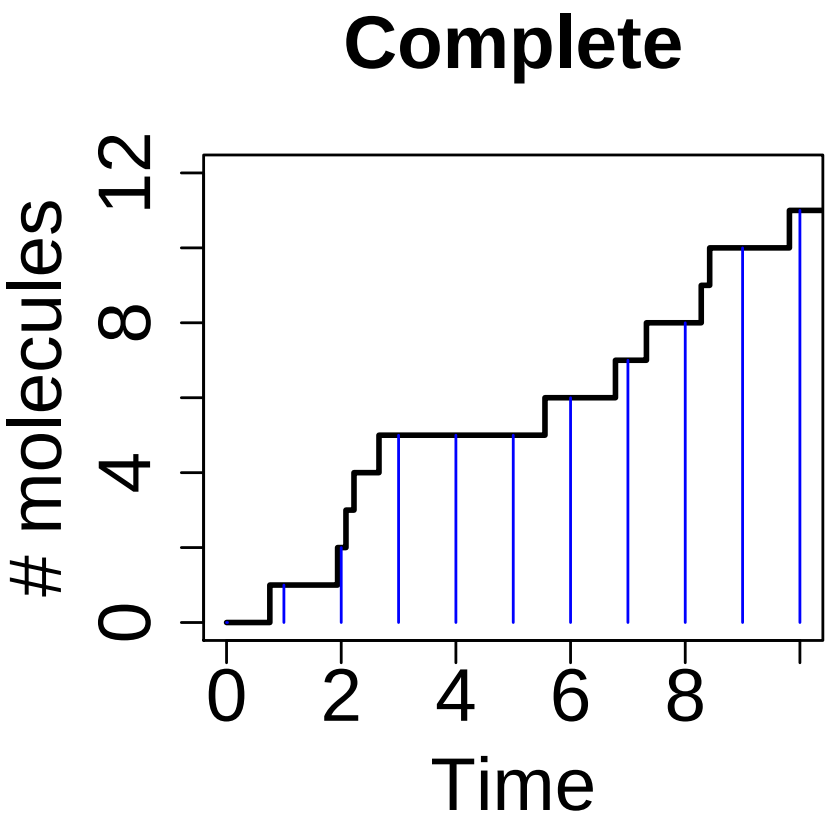
where

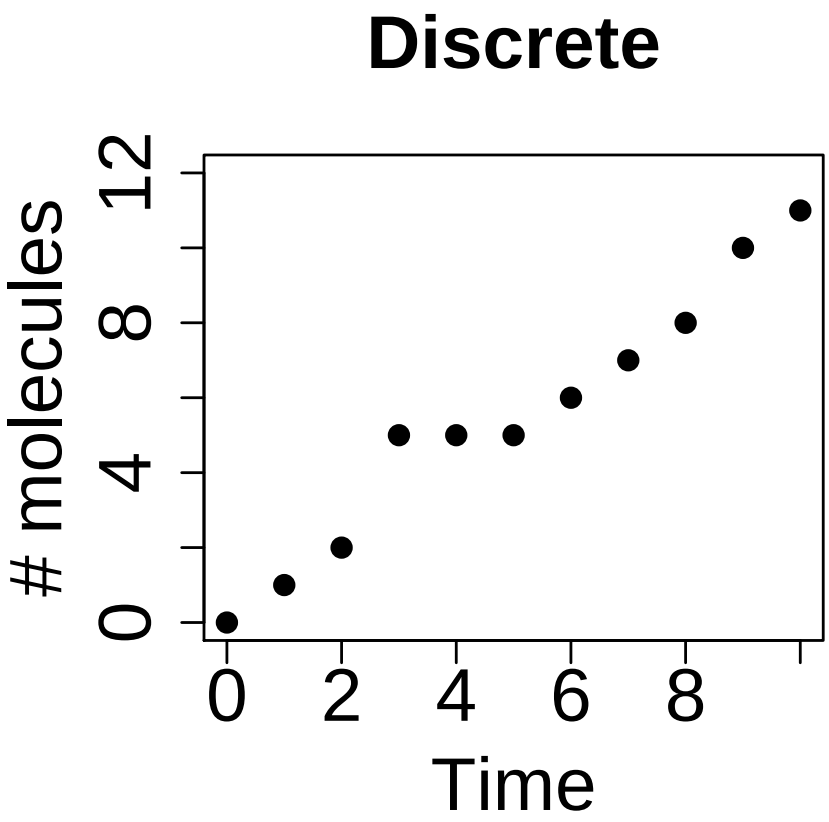
$$n_j = \sum_{i=1}^{n} \mathrm{I}(r_i = j) \quad \# \text{ of } j \text{ reactions}$$

$$I_j = \int_0^T h_j(X_t)dt = \sum_{i=1}^{n} h_j(X_{t_{i-1}})(t_i - t_{i-1}) + h_j(X_{t_n})[T - t_n]$$

Suppose you only observe the system at discrete-times:

- For simplicity, observe the system at times $t = 1, 2, \ldots, T$.
- At these times, we observe $y_t = X_t$ the system state.
- But do not observe the system between these times.





Inference is still performed based on the likelihood

$$L(\theta) = p(y|\theta) = p(t, y)$$

but this is the solution to the chemical master equation

$$\frac{\partial}{\partial t} p(t, y) = \sum_{j=1}^{M} \left(a_j(y - \nu_m) p(t, y - \nu_m) - a_j(y) p(t, y) \right)$$

which is generally intractable.

Gibbs sampling

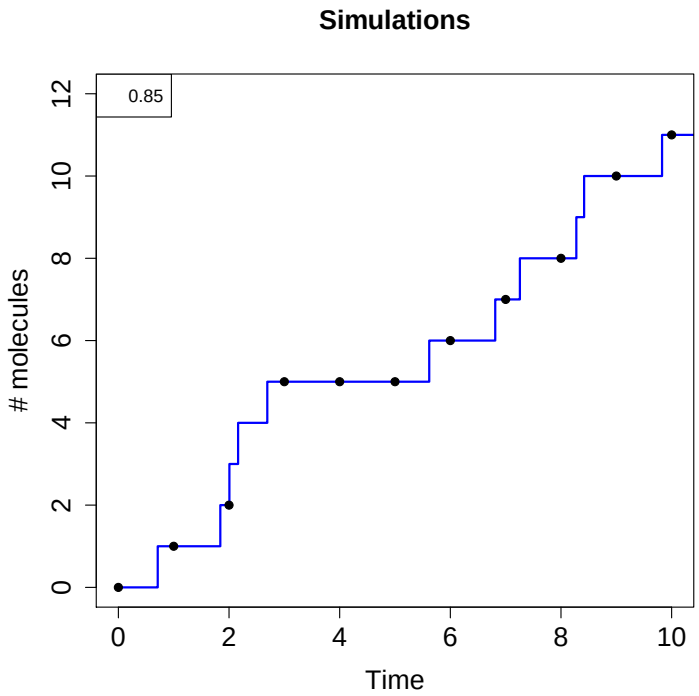
Our objective is samples from the posterior

$$p(\theta|y) = \int p(\theta, X|y)dX \propto \int p(y|X)p(X|\theta)p(\theta)dX$$

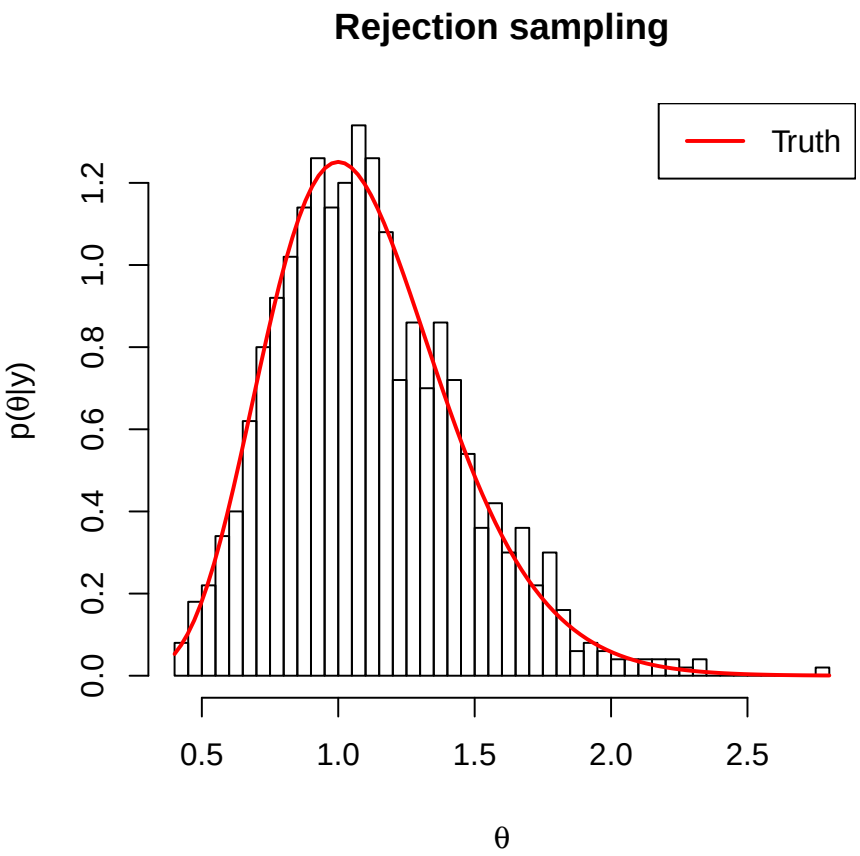
A Gibbs sampling procedure is

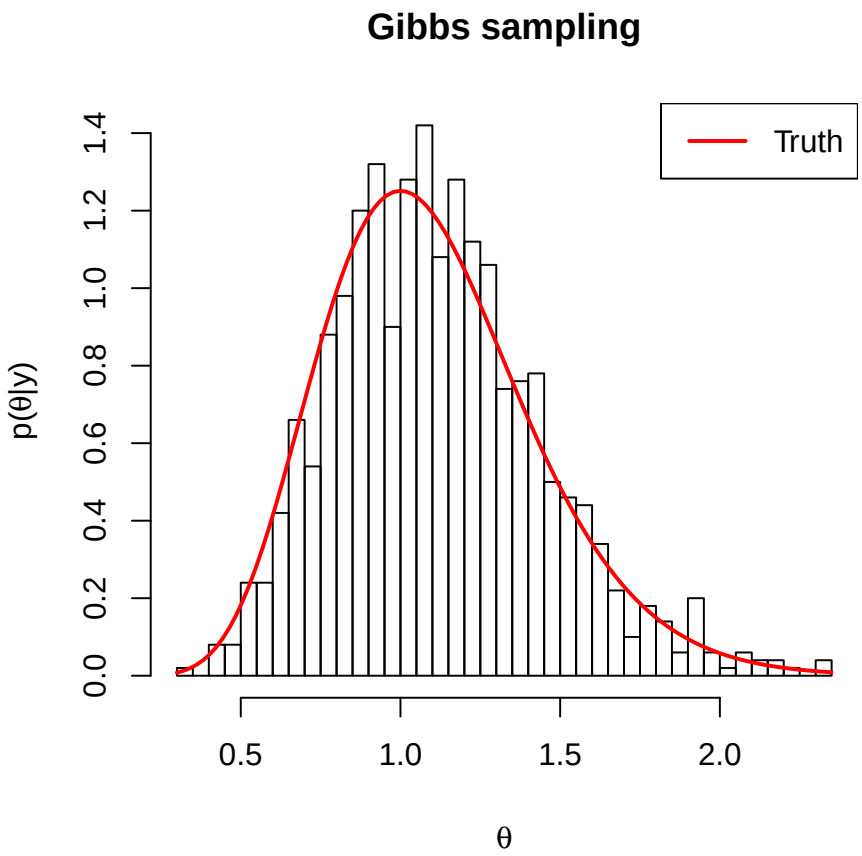
1. Start with $\theta^{(0)}, X^{(0)}$
2. For $k = 1, \ldots, K$,
   a. Sample $X^{(k)} \sim p(X|\theta^{(k-1)}, y)$
      a.k.a. rejection sampling
   b. Sample $\theta^{(k)} \sim p(\theta|X^{(k)})$

$\theta^{(k)}, X^{(k)}$ converge to samples from $p(\theta, X|y)$



Posteriors





So what's the problem? Extremely high rejection rates!!

Each thread forward simulates and checks to see if it matches the data.

Memory type	Amount (integers)	Algorithm allocation
Registers per block	32×512	Thread system time Thread loop variables
Shared per block	4 kb (8×512)	Twister state during SSA simulation Thread system state † Thread reaction propensities †
Local per thread	16 kb (4096)	Thread system state † Thread reaction propensities †
Global	4 Gb $(\approx 10^9)$	Success counter Successful twister state Twister states when not in use
Constant	64 kb (16,384)	Reaction rate parameters Stoichiometry matrix

† If shared memory is available, thread system state and reaction propensities are moved from local to shared memory.

Will the end user be able to utilize GPU parallelism without knowing the details (in R)?

- Pros
  - Hardware is accessible
  - Degree of parallelization is amazing
- Cons
  - Ability to write generic algorithms
  - Incorporating CUDA C into R

In the near future (5 years),

Matrix operations will parallelized, e.g. `A%*%B`: already here `gputools`

Standard statistical procedures, e.g. `lm`. already here `gputools`

Specialized statistical procedures in packages. already here `cudaBayesreg`

Generic all-purpose statistical algorithms. hard to imagine, but yes!

Not ideal, i.e. `solve(A)` $\rightarrow$ `gpuSolve(A)` rather than `solve(A, gpu=gpuID)`.

Resources

- Understanding GPU programming
  - Sanders and Kandrot. "CUDA by example"
  - Kirk and Hwu. "Programming Massively Parallel Processors"
  - Hwu. GPU "Computing Gems Emerald Edition"
- Getting started with GPU programming:
  CUDA toolkit and GPU computing SDK
- GPU computing in R
- Chris Holmes's GPU page