CUDA C: K-means and MCMC

Will Landau

Iowa State University

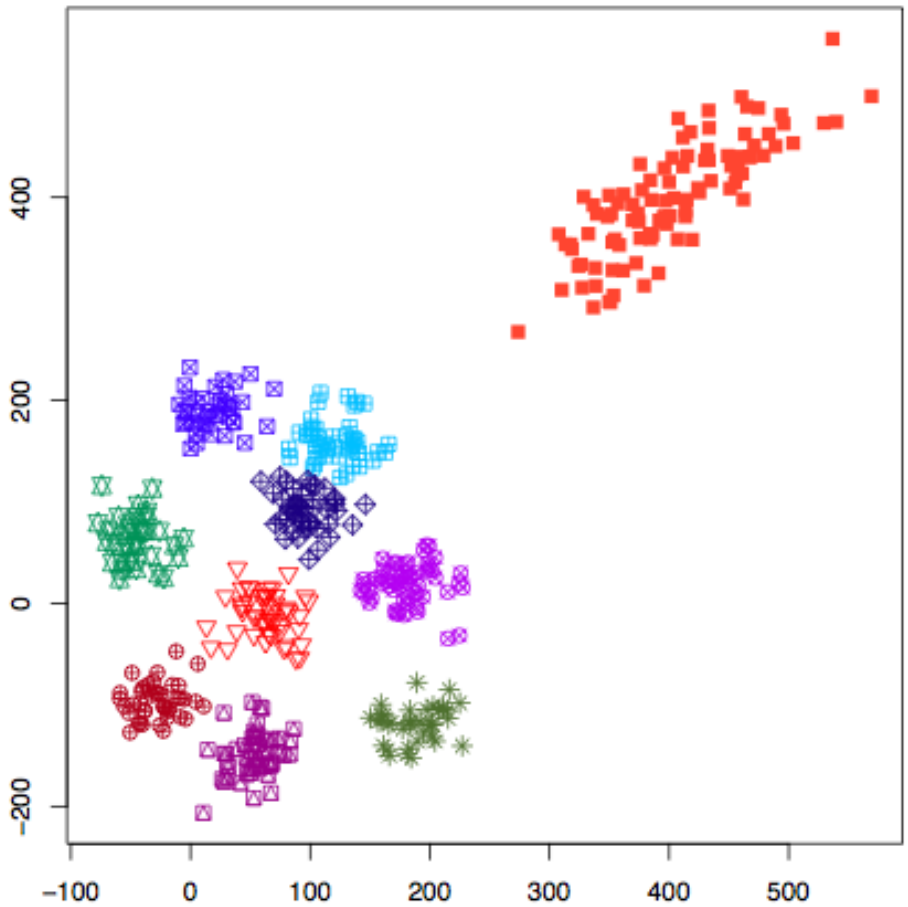
October 7, 2013

Outline

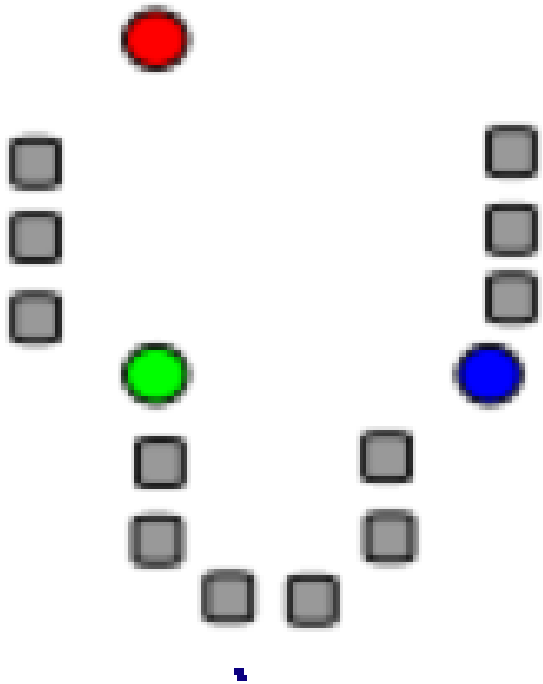
Outline

Lloyd's K-means algorithm

- Cluster N vectors in Euclidian space into K groups.

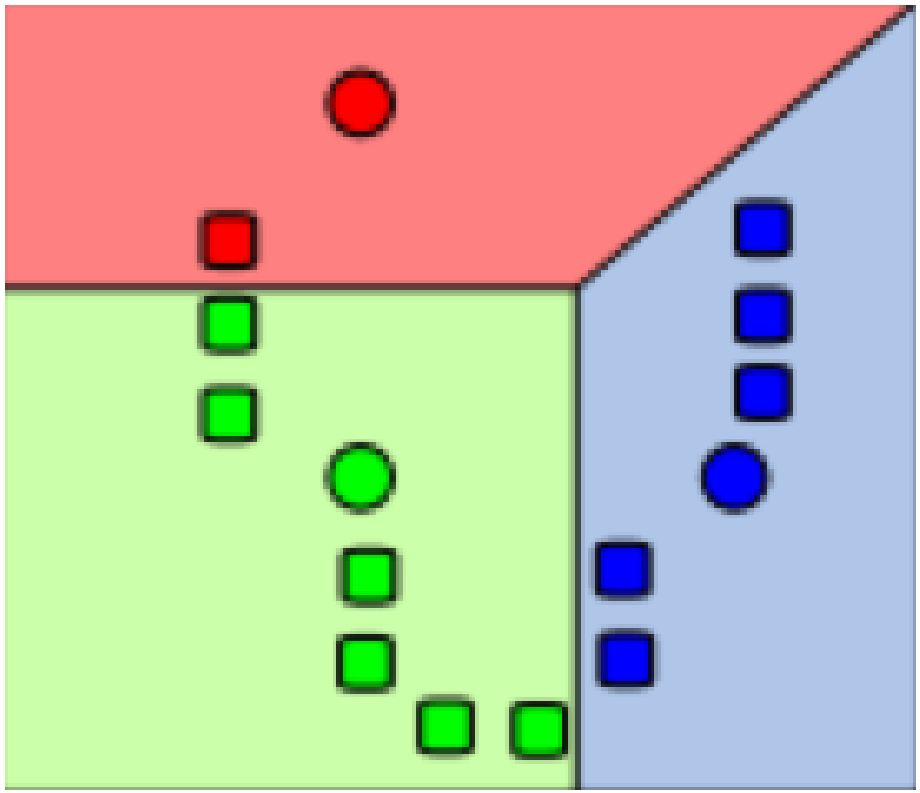


Step 1: choose initial cluster centers.

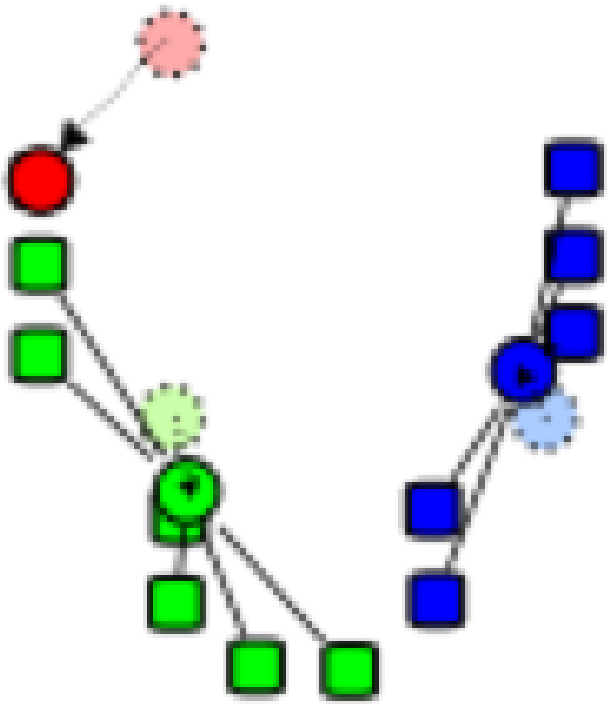


- The circles are the cluster means, the squares are the data points, and the color indicates the cluster.

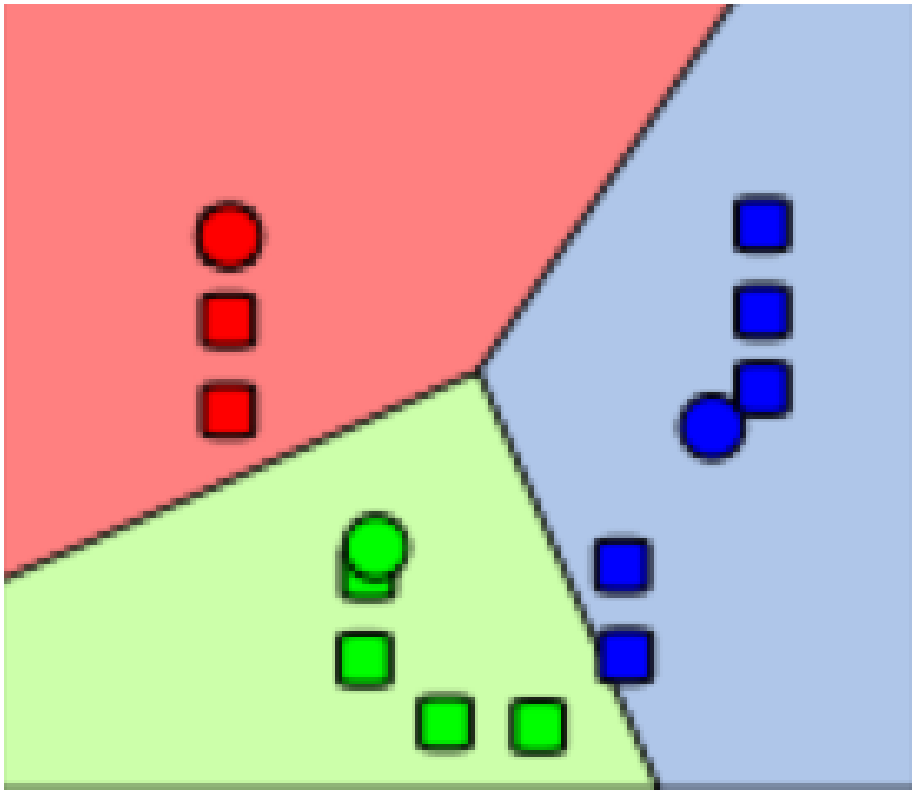
Step 2: assign each data point (square) to its closest center (circle).



Step 3: update the cluster centers to be the within-cluster data means.



Repeat step 2: reassign points to their closest cluster centers.



- ... and repeat until convergence.

Parallel K-means

- Step 2: assign points to closest cluster centers
  - Spawn N blocks with K threads each.
  - Let thread (n, k) compute the distance between data point n and cluster center k.
  - Synchronize threads within each block.
  - Let thread $(n, 1)$ assign data point n to its nearest cluster center.
- Step 3: recompute cluster centers
  - Spawn one block for each cluster.
  - Within each block, compute the mean of the data in the corresponding cluster.

gpu_kmeans.cu

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define I(row, col, ncols) (row * ncols + col)
5
6 #define CUDA_CALL(x) {if((x) != cudaSuccess){ \
7   printf("CUDA error at %s:%d\n",__FILE__,__LINE__); \
8   printf("  %s\n", cudaGetErrorString(cudaGetLastError())); \
9   exit(EXIT_FAILURE);}}
```

gpu_kmeans.cu: step 2

```
10 __global__ void get_dst(float *dst, float *x, float *y,
11 float *mu_x, float *mu_y){
12   int i = blockIdx.x;
13   int j = threadIdx.x;
14
15   dst[I(i, j, blockDim.x)] = (x[i] - mu_x[j]) * (x[i] - mu_x[j]);
16   dst[I(i, j, blockDim.x)] += (y[i] - mu_y[j]) * (y[i] - mu_y[j]);
17 }
18
19 __global__ void regroup(int *group, float *dst, int k){
20   int i = blockIdx.x;
21   int j;
22   float min_dst;
23
24   min_dst = dst[I(i, 0, k)];
25   group[i] = 1;
26
27   for(j = 1; j < k; ++j){
28     if(dst[I(i, j, k)] < min_dst){
29       min_dst = dst[I(i, j, k)];
30       group[i] = j + 1;
31     }
32   }
33 }
```

gpu_kmeans.cu: step 3

```
34 __global__ void clear(float *sum_x, float *sum_y, int *nx, int *ny){
35   int j = threadIdx.x;
36
37   sum_x[j] = 0;
38   sum_y[j] = 0;
39   nx[j] = 0;
40   ny[j] = 0;
41 }
42
43 __global__ void recenter_step1(float *sum_x, float *sum_y, int *nx,
       int *ny,
44         float *x, float *y, int *group, int n){
45   int i;
46   int j = threadIdx.x;
47
48   for(i = 0; i < n; ++i){
49     if(group[i] == (j + 1)){
50       sum_x[j] += x[i];
51       sum_y[j] += y[i];
52       nx[j]++;
53       ny[j]++;
54     }
55   }
56 }
```

gpu_kmeans.cu: step 3

```
57 __global__ void recenter_step2(float *mu_x, float *mu_y, float *sum_x
       ,
58         float *sum_y, int *nx, int *ny){
59   int j = threadIdx.x;
60
61   mu_x[j] = sum_x[j]/nx[j];
62   mu_y[j] = sum_y[j]/ny[j];
63 }
64
65 void kmeans(int nreps, int n, int k,
66             float *x_d, float *y_d, float *mu_x_d, float *mu_y_d,
67             int *group_d, int *nx_d, int *ny_d,
68             float *sum_x_d, float *sum_y_d, float *dst_d){
69   int i;
70   for(i = 0; i < nreps; ++i){
71     get_dst<<<n,k>>>(dst_d, x_d, y_d, mu_x_d, mu_y_d);
72     regroup<<<n,1>>>(group_d, dst_d, k);
73     clear<<<1,k>>>(sum_x_d, sum_y_d, nx_d, ny_d);
74     recenter_step1<<<1,k>>>(sum_x_d, sum_y_d, nx_d, ny_d, x_d, y_d,
           group_d, n);
75     recenter_step2<<<1,k>>>(mu_x_d, mu_y_d, sum_x_d, sum_y_d, nx_d,
           ny_d);
76   }
77 }
78
79 void read_data(float **x, float **y, float **mu_x, float **mu_y, int
       *n, int *k);
80 void print_results(int *group, float *mu_x, float *mu_y, int n, int k
       );
```

gpu_kmeans.cu

```
81  int main(){
82    /* cpu variables */
83    int n; /* number of points */
84    int k; /* number of clusters */
85    int *group;
86    float *x = NULL, *y = NULL, *mu_x = NULL, *mu_y = NULL;
87
88    /* gpu variables */
89    int *group_d, *nx_d, *ny_d;
90    float *x_d, *y_d, *mu_x_d, *mu_y_d, *sum_x_d, *sum_y_d, *dst_d;
91
92    /* read data from files on cpu */
93    read_data(&x, &y, &mu_x, &mu_y, &n, &k);
94
95    /* allocate cpu memory */
96    group = (int*) malloc(n*sizeof(int));
97
98    /* allocate gpu memory */
99    CUDA_CALL(cudaMalloc((void**) &group_d,n*sizeof(int)));
100   CUDA_CALL(cudaMalloc((void**) &nx_d, k*sizeof(int)));
101   CUDA_CALL(cudaMalloc((void**) &ny_d, k*sizeof(int)));
102   CUDA_CALL(cudaMalloc((void**) &x_d, n*sizeof(float)));
103   CUDA_CALL(cudaMalloc((void**) &y_d, n*sizeof(float)));
104   CUDA_CALL(cudaMalloc((void**) &mu_x_d, k*sizeof(float)));
105   CUDA_CALL(cudaMalloc((void**) &mu_y_d, k*sizeof(float)));
106   CUDA_CALL(cudaMalloc((void**) &sum_x_d, k*sizeof(float)));
107   CUDA_CALL(cudaMalloc((void**) &sum_y_d, k*sizeof(float)));
108   CUDA_CALL(cudaMalloc((void**) &dst_d, n*k*sizeof(float)));
```

gpu_kmeans.cu

```
109   /* write data to gpu */
110   CUDA_CALL(cudaMemcpy(x_d, x, n*sizeof(float),
            cudaMemcpyHostToDevice));
111   CUDA_CALL(cudaMemcpy(y_d, y, n*sizeof(float),
            cudaMemcpyHostToDevice));
112   CUDA_CALL(cudaMemcpy(mu_x_d, mu_x, k*sizeof(float),
            cudaMemcpyHostToDevice));
113   CUDA_CALL(cudaMemcpy(mu_y_d, mu_y, k*sizeof(float),
            cudaMemcpyHostToDevice));
114
115   /* perform kmeans */
116   kmeans(10, n, k, x_d, y_d, mu_x_d, mu_y_d, group_d, nx_d, ny_d,
            sum_x_d, sum_y_d, dst_d);
117
118   /* read back data from gpu */
119   CUDA_CALL(cudaMemcpy(group, group_d, n*sizeof(int),
            cudaMemcpyDeviceToHost));
120   CUDA_CALL(cudaMemcpy(mu_x, mu_x_d, k*sizeof(float),
            cudaMemcpyDeviceToHost));
121   CUDA_CALL(cudaMemcpy(mu_y, mu_y_d, k*sizeof(float),
            cudaMemcpyDeviceToHost));
122
123   /* print results and clean up */
124   print_results(group, mu_x, mu_y, n, k);
```

gpu_kmeans.cu

```
125     free(x);
126     free(y);
127     free(mu_x);
128     free(mu_y);
129     free(group);
130 
131     CUDA_CALL(cudaFree(x_d));
132     CUDA_CALL(cudaFree(y_d));
133     CUDA_CALL(cudaFree(mu_x_d));
134     CUDA_CALL(cudaFree(mu_y_d));
135     CUDA_CALL(cudaFree(group_d));
136     CUDA_CALL(cudaFree(nx_d));
137     CUDA_CALL(cudaFree(ny_d));
138     CUDA_CALL(cudaFree(sum_x_d));
139     CUDA_CALL(cudaFree(sum_y_d));
140     CUDA_CALL(cudaFree(dst_d));
141 
142     return 0;
143 }
```

Compile, test, and run.

- Compile CPU and GPU versions.

```
> make
gcc kmeans.c -o kmeans -Wall -ansi -pedantic
nvcc gpu_kmeans.cu -o gpu_kmeans --compiler-options -ansi --
    compiler-options -Wall
```

- Always run through CUDA MEMCHECK.

```
> cuda-memcheck ./gpu_kmeans
========= CUDA-MEMCHECK
========= ERROR SUMMARY: 0 errors
```

- Check CPU side with Valgrind. Ignore "errors" and apparent memory leaks on the GPU side.

```
> valgrind ./gpu_kmeans
==13523== Memcheck, a memory error detector
==13523== Copyright (C) 2002-2010, and GNU GPL'd, by Julian
    Seward et al.
==13523== Using Valgrind-3.6.0 and LibVEX; rerun with -h for
    copyright info
==13523== Command: ./gpu_kmeans
==13523==
==13523== Warning: set address range perms: large range [0
    x800000000, 0x2100000000) (noaccess)
==13523== Warning: set address range perms: large range [0
    x2100000000, 0x2800000000) (noaccess)
```

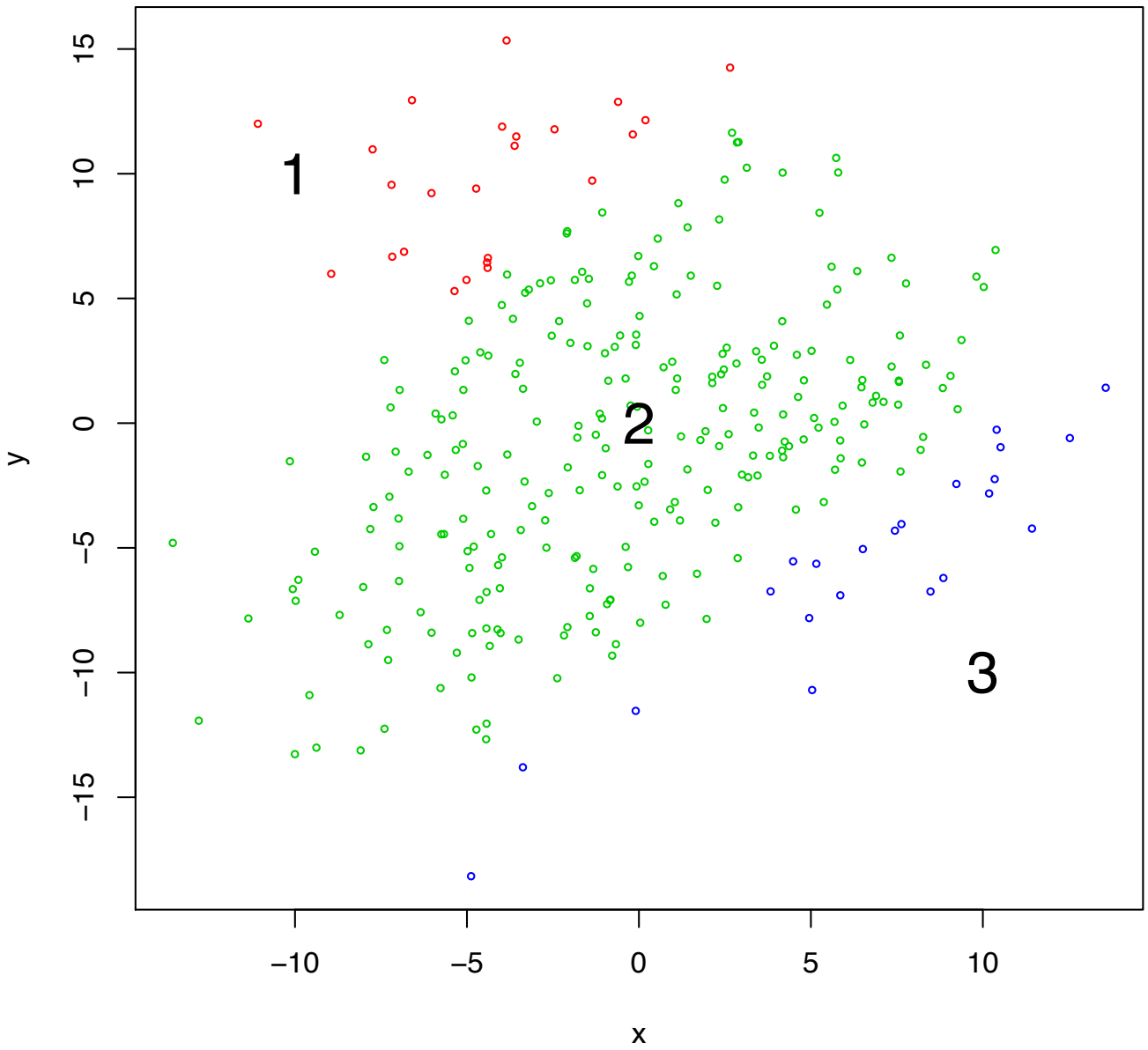
Compile, test, and run.

```
1  ==13523==
2  ==13523== HEAP SUMMARY:
3  ==13523==     in use at exit: 1,308,694 bytes in 2,469 blocks
4  ==13523==   total heap usage: 4,479 allocs, 2,010 frees,
       2,952,191 bytes allocated
5  ==13523==
6  ==13523== LEAK SUMMARY:
7  ==13523==    definitely lost: 16 bytes in 1 blocks
8  ==13523==    indirectly lost: 0 bytes in 0 blocks
9  ==13523==      possibly lost: 33,064 bytes in 242 blocks
10 ==13523==    still reachable: 1,275,614 bytes in 2,226 blocks
11 ==13523==         suppressed: 0 bytes in 0 blocks
12 ==13523== Rerun with --leak-check=full to see details of leaked
       memory
13 ==13523==
14 ==13523== For counts of detected and suppressed errors, rerun
       with: -v
15 ==13523== ERROR SUMMARY: 0 errors from 0 contexts (suppressed:
       11 from 9)
```

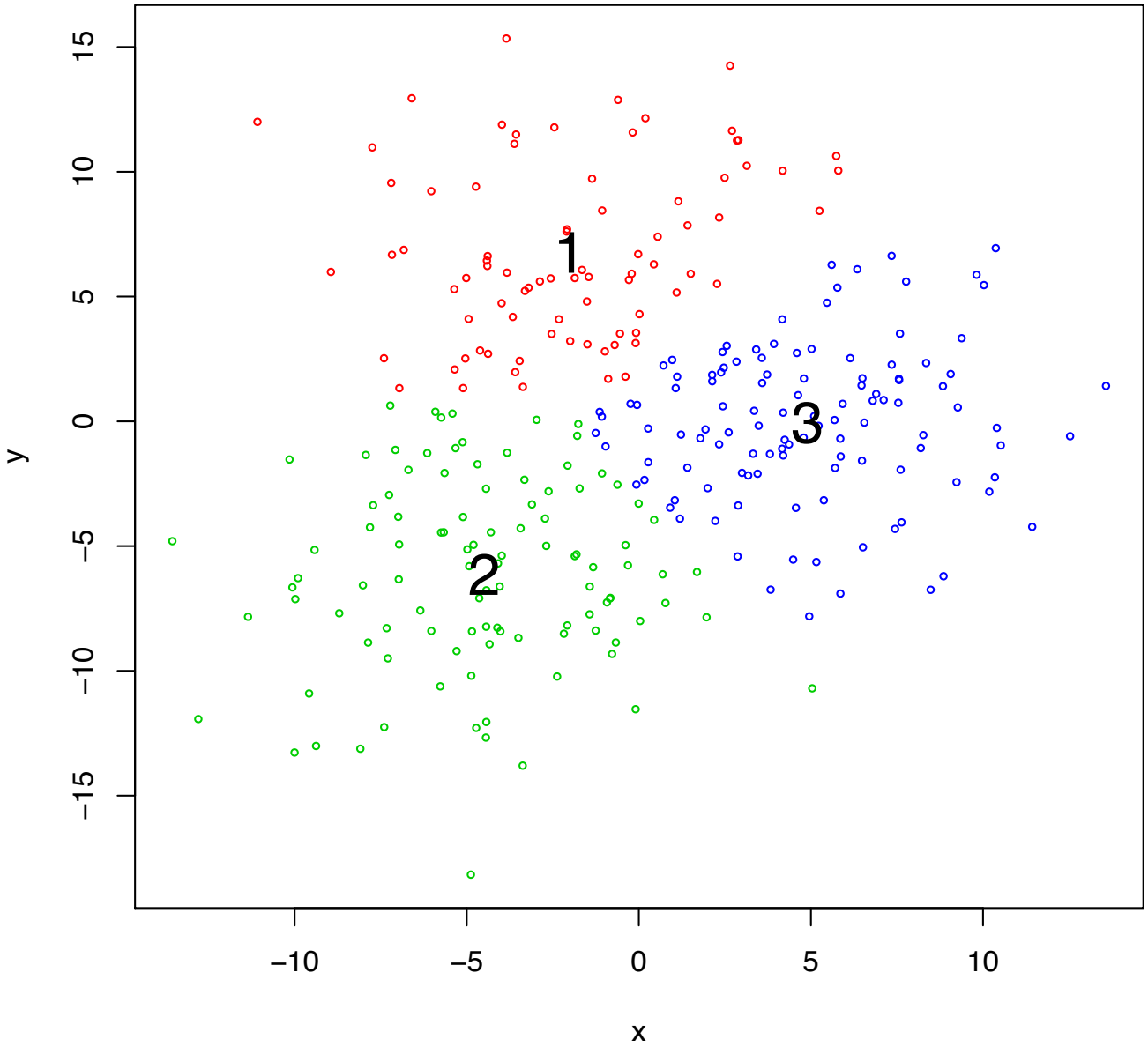
- Run for real.

```
1  > ./gpu_kmeans
```

Initial clustering: 300 points, 3 clusters



Final clustering after 10 iterations



Outline

Markov chain Monte Carlo

- Consider a bladder cancer data set:
  - Available from http://ratecalc.cancer.gov/.
  - Rates of death from bladder cancer of white males from 2000 to 2004 in each county in the USA.
- Let:
  - y_k = number of observed deaths in county k.
  - n_k = the number of person-years in county k divided by 100,000.
  - θ_k = expected number of deaths per 100,000 person-years.
- The model:

$$y_k \overset{\text{ind}}{\sim} \text{Poisson}(n_k \cdot \theta_k)$$

$$\theta_k \overset{\text{iid}}{\sim} \text{Gamma}(\alpha,\ \beta)$$

$$\alpha \sim \text{Uniform}(0, a_0)$$

$$\beta \sim \text{Uniform}(0, b_0)$$

  - Also assume that shape α and rate β are independent and fix a_0 and b_0.

Full conditional distributions

- We want to sample from the joint posterior,

$$
\begin{aligned}
p(\boldsymbol{\theta}, \alpha, \beta \mid y) &\propto p(y \mid \boldsymbol{\theta}, \alpha, \beta) p(\boldsymbol{\theta}, \alpha, \beta) \\
&\propto p(y \mid \boldsymbol{\theta}, \alpha, \beta) p(\boldsymbol{\theta} \mid \alpha, \beta) p(\alpha, \beta) \\
&\propto p(y \mid \boldsymbol{\theta}, \alpha, \beta) p(\boldsymbol{\theta} \mid \alpha, \beta) p(\alpha) p(\beta) \\
&\propto \prod_{k=1}^{K} [p(y_k \mid \theta_k, n_k) p(\theta_k \mid \alpha, \beta)] p(\alpha) p(\beta) \\
&\propto \prod_{k=1}^{K} \left[e^{-n_k \theta_k} \theta_k^{y_k} \frac{\beta^\alpha}{\Gamma(\alpha)} \theta_k^{\alpha - 1} e^{-\theta_k \beta} \right] I(0 < \alpha < a_0) I(0 < \beta < b_0)
\end{aligned}
$$

- We iteratively sample from the full conditional distributions.

$$
\begin{aligned}
\alpha &\leftarrow p(\alpha \mid y, \boldsymbol{\theta}, \beta) \\
\beta &\leftarrow p(\beta \mid y, \boldsymbol{\theta}, \alpha) \\
\theta_k &\leftarrow p(\theta_k \mid y, \boldsymbol{\theta}_{-k}, \alpha, \beta) \qquad \Leftarrow \text{IN PARALLEL!}
\end{aligned}
$$

Full conditional distributions

$$
\begin{aligned}
p(\theta_k \mid y, \boldsymbol{\theta}_{-k}, \alpha, \beta) &\propto p(\boldsymbol{\theta}, \alpha, \beta \mid y) \\
&\propto e^{-n_k \theta_k} \theta_k^{y_k} \theta_k^{\alpha - 1} e^{-\theta_k \beta} \\
&= \theta_k^{y_k + \alpha - 1} e^{-\theta_k (n_k + \beta)} \\
&\propto \text{Gamma}(y_k + \alpha,\ n_k + \beta)
\end{aligned}
$$

Conditional distributions of α and β

$$
\begin{aligned}
p(\alpha \mid y, \boldsymbol{\theta}, \beta) &\propto p(\boldsymbol{\theta}, \alpha, \beta \mid y) \\
&\propto \prod_{k=1}^{K} \left[\theta_k^{\alpha-1} \frac{\beta^\alpha}{\Gamma(\alpha)} \right] I(0 < \alpha < a_0) \\
&= \left(\prod_{k=1}^{K} \theta_k \right)^{\alpha} \beta^{K\alpha} \Gamma(\alpha)^{-K} I(0 < \alpha < a_0)
\end{aligned}
$$

$$
\begin{aligned}
p(\beta \mid y, \boldsymbol{\theta}, \alpha) &\propto p(\boldsymbol{\theta}, \alpha, \beta \mid y) \\
&\propto \prod_{k=1}^{K} \left[e^{-\theta_k \beta} \beta^\alpha \right] I(0 < \beta < b_0) \\
&= \beta^{K\alpha} e^{-\beta \sum_{k=1}^{K} \theta_k} I(0 < \beta < b_0) \\
&\propto \text{Gamma}\left(K\alpha + 1, \sum_{k=1}^{K} \theta_k \right) I(0 < \beta < b_0)
\end{aligned}
$$

Summarizing the Gibbs sampler

1. Sample $\boldsymbol{\theta}$ from from its full conditional.
   - Draw the θ_k's *in parallel* from independent Gamma($y_k + \alpha$, $n_k + \beta$) distributions.
   - In other words, assign each thread to draw an individual θ_k from its Gamma($y_k + \alpha$, $n_k + \beta$) distribution.
2. Sample α from its full conditional using a random walk Metropolis step.
3. Sample β from its full conditional (truncated Gamma) using the inverse cdf method if b_0 is low or a non-truncated Gamma if b_0 is high.

The gamma sampler for β and $\theta_1, \ldots, \theta_K$

- Taken from Marsaglia and Tsang (2001).
- Rejection sampler with acceptance rate $\geq 95\%$ when the shape parameter is ≥ 1.
- Steps for a Gamma(a, 1) sampler:
  1. Let $d = a - 1/3$ and $c = 1/\sqrt{9d}$.
  2. Draw $x \sim$ Normal(0, 1) and let $v = (1 + c \cdot x)^3$. Repeat if $v \leq 0$.
  3. Let $u \sim$ Uniform(0, 1).
  4. If $u < 1 - 0.0331 \cdot x^4$, return $d \cdot v$.
  5. If $\log(u) < 0.5 \cdot x^2 + d \cdot (1 - v + \log(v))$, return $d \cdot v$.
  6. Go to step 2.

The metropolis step for α

- The goal is to sample from the full conditional distribution,

$$p(\alpha \mid y, \boldsymbol{\theta}, \beta) \propto \left(\prod_{k=1}^{K} \theta_k \right)^{\alpha} \beta^{K\alpha} \Gamma(\alpha)^{-K} I(0 < \alpha < a_0)$$

- Let $\alpha^{(j)}$ be the last sampled value of α. Sample $\alpha^{(j+1)}$ as follows:
  1. Draw $\alpha^* \sim N(\alpha^{(j)}, \sigma^2)$, where σ^2 is a tuning parameter.
  2. If $\alpha^* < 0$ or $\alpha^* > a_0$, let $p = 0$. Otherwise,

$$p = \frac{p(\alpha^* \mid y, \boldsymbol{\theta}, \beta)}{p(\alpha^{(j)} \mid y, \boldsymbol{\theta}, \beta)} = \left(\prod_{k=1}^{K} \theta_k \right)^{\alpha^* - \alpha^{(j)}} \beta^{K(\alpha^* - \alpha^{(j)})} \left(\frac{\Gamma(\alpha^*)}{\Gamma(\alpha^{(j)})} \right)^{-K}$$

  3. Draw $u \sim U(0, 1)$.
  4. If $u < p$, $\alpha^{(j+1)} = \alpha^*$. Otherwise, $\alpha^{(j+1)} = \alpha^{(j)}$.
  5. Raise σ^2 if α^* was accepted and lower σ^2 if α^* was rejected. The optimal acceptance rate is roughly 44%.

Zebulun Arendsee's implementation

```
1  /*
2  Created by Zebulun Arendsee.
3  March 26, 2013
4
5  Modified by Will Landau.
6  June 30, 2013
7  will-landau.com
8  landau@iastate.edu
9
10 This program implements a MCMC algorithm for the following
       hierarchical
11 model:
12
13 y_k     ~ Poisson(n_k * theta_k)     k = 1, ..., K
14 theta_k ~ Gamma(a, b)
15 a       ~ Unif(0, a0)
16 b       ~ Unif(0, b0)
17
18 We let a0 and b0 be arbitrarily large.
19
20 Arguments:
21     1) input filename
22         With two space delimited columns holding integer values for
23         y and float values for n.
24     2) number of trials (1000 by default)
25
26 Output: A comma delimited file containing a column for a, b, and each
27 theta. All output is written to stdout. */
```

Zebulun Arendsee's implementation

```
28 /*
29 Example dataset:
30
31 $ head -3 data.txt
32 4 0.91643
33 23 3.23709
34 7 0.40103
35
36  Example of compilation and execution:
37
38 $ nvcc gibbs_metropolis.cu -o gibbs
39 $ ./gibbs mydata.txt 2500 > output.csv
40 $
41
42 This code borrows from the nVidia developer zone documentation,
43 specifically http://docs.nvidia.com/cuda/curand/index.html#
       topic_1_2_1
44 */
45
46 #include <stdio.h>
47 #include <stdlib.h>
48 #include <cuda.h>
49 #include <math.h>
50 #include <curand_kernel.h>
51 #include <thrust/reduce.h>
52
53 #define PI 3.14159265359f
54 #define THREADS_PER_BLOCK 64
```

Zebulun Arendsee's implementation

```
55 #define CUDA_CALL(x) {if((x) != cudaSuccess){ \
56    printf("CUDA error at %s:%d\n",__FILE__,__LINE__); \
57    printf("  %s\n", cudaGetErrorString(cudaGetLastError())); \
58    exit(EXIT_FAILURE);}}
59
60 #define CURAND_CALL(x) {if((x) != CURAND_STATUS_SUCCESS) { \
61    printf("Error at %s:%d\n",__FILE__,__LINE__); \
62    printf("  %s\n", cudaGetErrorString(cudaGetLastError())); \
63    exit(EXIT_FAILURE);}}
64
65 __host__ void load_data(int argc, char **argv, int *K, int **y, float
       **n);
66
67 __host__ float sample_a(float a, float b, int K, float sum_logs);
68 __host__ float sample_b(float a, int K, float flat_sum);
69
70 __host__ float rnorm();
71 __host__ float rgamma(float a, float b);
72
73 __device__ float rgamma(curandState *state, int id, float a, float b)
       ;
74
75 __global__ void sample_theta(curandState *state, float *theta, float
       *log_theta, int *y, float *n, float a, float b, int K);
76 __global__ void setup_kernel(curandState *state, unsigned int seed,
       int);
```

Zebulun Arendsee's implementation

```
77  int main(int argc, char **argv){
78
79    curandState *devStates;
80    float a, b, flat_sum, sum_logs, *n, *dev_n, *dev_theta, *
          dev_log_theta;
81    int i, K, *y, *dev_y, nBlocks, trials = 1000;
82
83    if(argc > 2)
84      trials = atoi(argv[2]);
85
86    load_data(argc, argv, &K, &y, &n);
87
88
89    /*------ Allocate memory ------------------------------------------
          */
90
91    CUDA_CALL(cudaMalloc((void **)&dev_y, K * sizeof(int)));
92    CUDA_CALL(cudaMemcpy(dev_y, y, K * sizeof(int),
93              cudaMemcpyHostToDevice));
94
95    CUDA_CALL(cudaMalloc((void **)&dev_n, K * sizeof(float)));
96    CUDA_CALL(cudaMemcpy(dev_n, n, K * sizeof(float),
97              cudaMemcpyHostToDevice));
98
99    /* Allocate space for theta and log_theta on device and host */
100   CUDA_CALL(cudaMalloc((void **)&dev_theta, K * sizeof(float)));
101   CUDA_CALL(cudaMalloc((void **)&dev_log_theta, K * sizeof(float)));
102
103   /* Allocate space for random states on device */
104   CUDA_CALL(cudaMalloc((void **)&devStates, K * sizeof(curandState)))
          ;
```

Zebulun Arendsee's implementation

```
105  /*------ Setup random number generators (one per thread) ------
         */
106
107  nBlocks = (K + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
108  setup_kernel<<<nBlocks, THREADS_PER_BLOCK>>>(devStates, 0, K);
```

Zebulun Arendsee's implementation

```
110   /*---------- MCMC ----------*/
111
112   printf("alpha, beta\n");
113
114   /* starting values of hyperparameters */
115   a = 20;
116   b = 1;
117
118   /* Steps of MCMC */
119   for(i = 0; i < trials; i++){
120     sample_theta<<<nBlocks, THREADS_PER_BLOCK>>>(devStates, dev_theta
            , dev_log_theta, dev_y, dev_n, a, b, K);
121
122     /* print hyperparameters. */
123     printf("%f, %f\n", a, b);
124
125     /* Make iterators for thetas and log thetas. */
126     thrust::device_ptr<float> theta(dev_theta);
127     thrust::device_ptr<float> log_theta(dev_log_theta);
128
129     /* Compute pairwise sums of thetas and log_thetas. */
130     flat_sum = thrust::reduce(theta, theta + K);
131     sum_logs = thrust::reduce(log_theta, log_theta + K);
132
133     /* Sample hyperparameters. */
134     a = sample_a(a, b, K, sum_logs);
135     b = sample_b(a, K, flat_sum);
136   }
```

Zebulun Arendsee's implementation

```
137     /*--------- Free Memory ---------------------------------------------------*/
138
139     free(y);
140     free(n);
141
142     CUDA_CALL(cudaFree(devStates));
143     CUDA_CALL(cudaFree(dev_theta));
144     CUDA_CALL(cudaFree(dev_log_theta));
145     CUDA_CALL(cudaFree(dev_y));
146     CUDA_CALL(cudaFree(dev_n));
147
148     return EXIT_SUCCESS;
149 }
```

Zebulun Arendsee's implementation

```
150 /*
151  *  Metropolis algorithm for producing random a values.
152  *  The proposal distribution in normal with a variance that
153  *  is adjusted at each step.
154  */
155
156 __host__ float sample_a(float a, float b, int K, float sum_logs){
157
158   static float sigma = 2;
159   float U, log_acceptance_ratio, proposal = rnorm() * sigma + a;
160
161   if(proposal <= 0)
162     return a;
163
164   log_acceptance_ratio = (proposal - a) * sum_logs +
165                          K * (proposal - a) * log(b) -
166                          K * (lgamma(proposal) - lgamma(a));
167
168   U = rand() / float(RAND_MAX);
169
170   if(log(U) < log_acceptance_ratio){
171     sigma *= 1.1;
172     return proposal;
173   } else {
174     sigma /= 1.1;
175     return a;
176   }
177 }
```

Zebulun Arendsee's implementation

```
178 /*
179  *  Sample b from a gamma distribution.
180  */
181
182 __host__ float sample_b(float a, int K, float flat_sum){
183
184   float hyperA = K * a + 1;
185   float hyperB = flat_sum;
186   return rgamma(hyperA, hyperB);
187 }
188
189
190 __host__ float rnorm(){
191
192   float U1 = rand() / float(RAND_MAX);
193   float U2 = rand() / float(RAND_MAX);
194   float V1 = sqrt(-2 * log(U1)) * cos(2 * PI * U2);
195   /* float V2 = sqrt(-2 * log(U2)) * cos(2 * PI * U1); */
196   return V1;
197 }
```

Zebulun Arendsee's implementation

```
198 __host__ float rgamma(float a, float b){
199
200   float d = a - 1.0 / 3;
201   float Y, U, v;
202
203   while(1){
204     Y = rnorm();
205     v = pow((1 + Y / sqrt(9 * d)), 3);
206
207     /* Necessary to avoid taking the log of a negative number later.
          */
208     if(v <= 0)
209       continue;
210
211     U = rand() / float(RAND_MAX);
212
213     /* Accept the sample under the following condition.
214        Otherwise repeat loop. */
215     if(log(U) < 0.5 * pow(Y,2) + d * (1 - v + log(v)))
216           return d * v / b;
217   }
218 }
```

Zebulun Arendsee's implementation

```
219 __device__ float rgamma(curandState *state, int id, float a, float b)
        {
220
221   float d = a - 1.0 / 3;
222   float Y, U, v;
223
224   while(1){
225     Y = curand_normal(&state[id]);
226     v = pow((1 + Y / sqrt(9 * d)), 3);
227
228     /* Necessary to avoid taking the log of a negative number later.
            */
229     if(v <= 0)
230       continue;
231
232     U = curand_uniform(&state[id]);
233
234     /* Accept the sample under the following condition.
235        Otherwise repeat loop. */
236     if(log(U) < 0.5 * pow(Y,2) + d * (1 - v + log(v)))
237       return d * v / b;
238   }
239 }
```

Zebulun Arendsee's implementation

```
240 /*
241  *  Sample each theta from the appropriate gamma distribution
242  */
243
244 __global__ void sample_theta(curandState *state,  float *theta,
245                              float *log_theta, int *y, float *n,
                                       float a, float b, int K){
246
247   int id = threadIdx.x + blockIdx.x * blockDim.x;
248   float hyperA, hyperB;
249
250   if(id < K){
251     hyperA = a + y[id];
252     hyperB = b + n[id];
253     theta[id] = rgamma(state, id, hyperA, hyperB);
254     log_theta[id] = log(theta[id]);
255   }
256 }
257
258 /*
259  *  Initialize GPU random number generators
260  */
261
262 __global__ void setup_kernel(curandState *state, unsigned int seed,
        int K){
263   int id = threadIdx.x + blockIdx.x * blockDim.x;
264   if(id < K)
265     curand_init(seed, id, 0, &state[id]);
266 }
```

Compile, test, and run.

- Compile, requiring compute capability 2.0 or above.

```
> nvcc -arch=sm_20 gibbs_metropolis.cu -o gibbs_metropolis
```

- Always check with CUDA MEMCHECK.

```
> cuda-memcheck ./gibbs_metropolis smallData.txt 10
========= CUDA-MEMCHECK
alpha, beta
19.070215, 1.226651
19.441961, 1.521381
16.017313, 1.413954
11.898635, 1.253917
11.898635, 1.767045
13.532320, 1.783169
13.532320, 1.648099
13.532320, 2.005379
13.532320, 2.136331
12.914721, 1.408586
========= ERROR SUMMARY: 0 errors
```

Compile, test, and run.

- Check CPU side with Valgrind. Ignore "errors" from GPU code.

```
1  > valgrind ./gibbs_metropolis smallData.txt 10
2  ==12942== Memcheck, a memory error detector
3  ==12942== Copyright (C) 2002-2010, and GNU GPL'd, by Julian
       Seward et al.
4  ==12942== Using Valgrind-3.6.0 and LibVEX; rerun with -h for
       copyright info
5  ==12942== Command: ./gibbs_metropolis smallData.txt 10
6  ==12942==
7  ==12942== Warning: set address range perms: large range [0
       x800000000, 0x2100000000) (noaccess)
8  ==12942== Warning: set address range perms: large range [0
       x2100000000, 0x2800000000) (noaccess)
9  alpha, beta
10 19.070215, 1.226651
11 19.441961, 1.521381
12 16.017313, 1.413954
13 11.898635, 1.253917
14 11.898635, 1.767045
15 13.532320, 1.783169
16 13.532320, 1.648099
17 13.532320, 2.005379
18 13.532320, 2.136331
19 12.914721, 1.408586
```

Compile, test, and run.

```
==12942==
==12942== HEAP SUMMARY:
==12942==     in use at exit: 1,453,685 bytes in 2,647 blocks
==12942==   total heap usage: 4,175 allocs, 1,528 frees,
      2,706,460 bytes allocated
==12942==
==12942== LEAK SUMMARY:
==12942==    definitely lost: 16 bytes in 1 blocks
==12942==    indirectly lost: 0 bytes in 0 blocks
==12942==      possibly lost: 39,184 bytes in 287 blocks
==12942==    still reachable: 1,414,485 bytes in 2,359 blocks
==12942==         suppressed: 0 bytes in 0 blocks
==12942== Rerun with --leak-check=full to see details of leaked
      memory
==12942==
==12942== For counts of detected and suppressed errors, rerun
      with: -v
==12942== ERROR SUMMARY: 0 errors from 0 contexts (suppressed:
      11 from 9)
```

- Run 2 chains for real.

```
> ./gibbs_metropolis data.txt 10000 > chain1.txt
> ./gibbs_metropolis data.txt 10000 > chain2.txt
```

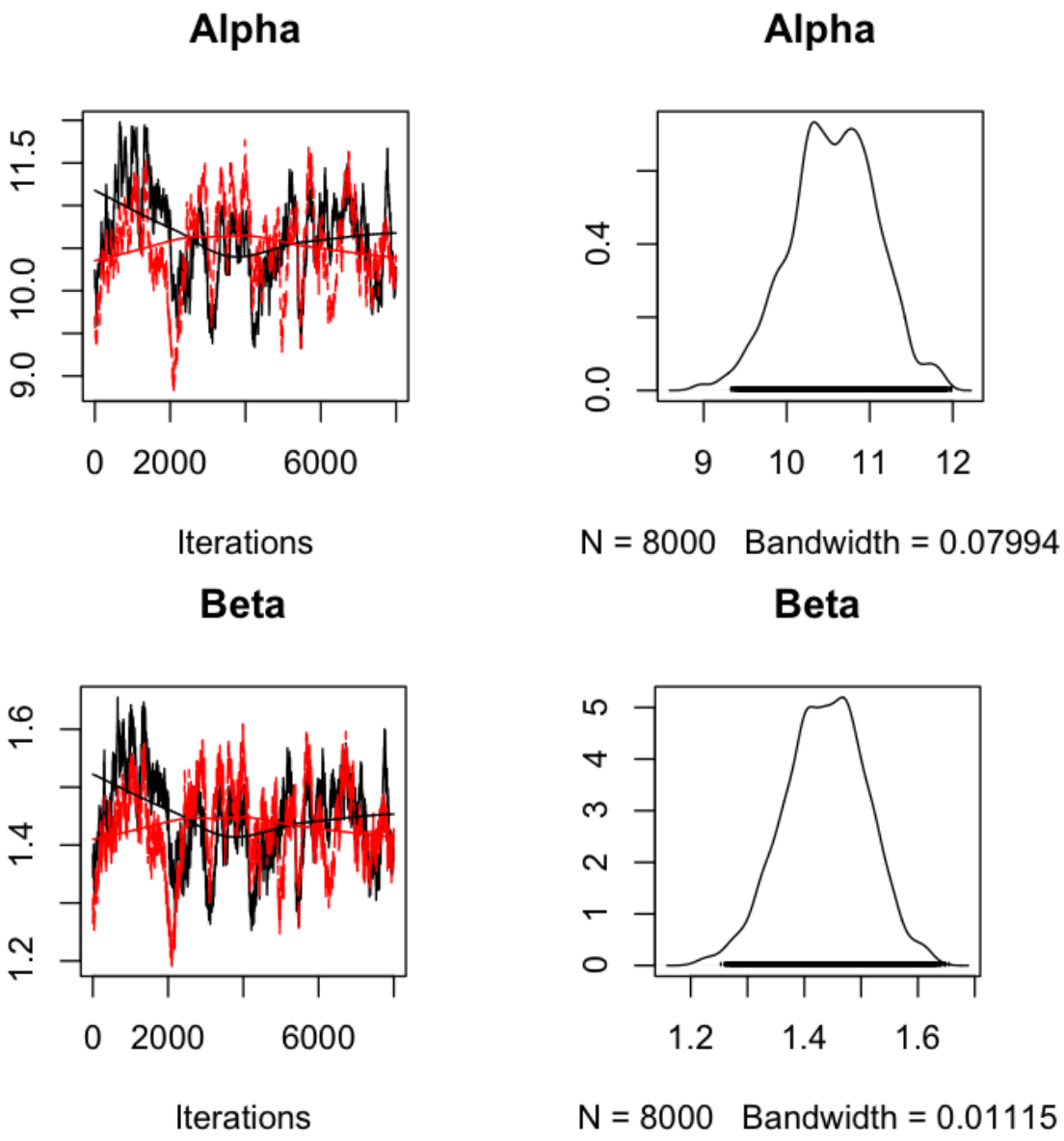
Diagnostics: Gelman-Rubin potential scale reduction factor

$$\widehat{R} = \sqrt{\frac{\frac{n-1}{n}W + \frac{1}{n}B}{W}} \approx \sqrt{1 + \frac{B}{nW}}$$

- n is the number of chains, B is the between-chain variability, and W is the within-chain variability.
- $\widehat{R} > 1.1$ is evidence of a lack of convergence.
- For 2 chains, each with 10000 iterations (including 2000 iterations of burn-in),

	Point est $\widehat{R}$	Upper 95% CI $\widehat{R}$
α	1.02	1.08
β	1.02	1.08

Diagnostics: trace plots after burn-in



Outline

Resources

- Sources:
  1. Dr. Ranjan Maitra's STAT 580 notes.
  2. Dr. Jarad Niemi's STAT 544 notes.
  3. Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis*. 2nd ed. Chapman & Hall/CRC, 2004.
  4. George Marsaglia and Wai Wan Tsang. "A Simple Method for Generating Gamma Variables." *ACM Transactions on Mathematical Software* 26(3), Sept 2000. 363-372.
- Code from today:
  - kmeans.zip
  - mcmc.zip

That's all for today.

- Series materials are available at http://will-landau.com/gpu.