A short course in GPU computing for statisticians

Will Landau

Iowa State University

September 9, 2013

Outline

GPUs, parallelism, and why we care

CUDA and our CUDA systems

GPU computing with R

How GPU parallelism works

Examples

- Vector addition
- Pairwise sum
- K-means clustering
- Markov chain Monte Carlo
- Ordinary least squares

Outline

A short course in GPU computing for statisticians

Will Landau

GPUs, parallelism, and why we care

CUDA and our CUDA systems

GPU computing with R

How GPU parallelism works

Examples

Vector addition
Pairwise sum
K-means clustering
Markov chain Monte Carlo
Ordinary least squares

The single instruction, multiple data (SIMD) paradigm

- SIMD: apply the same command to multiple places in a dataset.

```
1 for(i = 0; i < 1e6; ++i)
2   a[i] = b[i] + c[i];
```

- On CPUs, the iterations of the loop run sequentially.
- With GPUs, we can easily run all 1,000,000 iterations simultaneously.

```
1 i = threadIdx.x;
2 a[i] = b[i] + c[i];
```

- We can similarly *parallelize* a lot more than just loops.

Parallel MCMC by Lee, Yau, Giles, and others

# chains	CPU time (min)	GTX 280 (min)	CPU time / GPU time
8	0.0166	0.0148	1.1
32	0.0656	0.0151	4
128	0.262	.0154	17
512	1.04	0.0174	60
2,048	4.16	0.0248	168
8,192	16.64	0.0720	230
32,768	66.7	0.249	268
131,072	270.3	0.970	279

Parallel sequential MC by Lee, Yau, Giles, and others

Sample size	CPU time (min)	GTX 280 (min)	CPU time / GPU time
8,192	4.44	0.0199	223.1
16,384	8.82	0.0355	263
32,768	17.7	0.0666	265
65,536	35.3	0.131	269
131,076	70.6	0.261	270.5
262,144	141	0.52	271.2

Parallel Bayesian EM by Suchard, Wang, Chan, and others

Sample size	cpu 1 (sec)	gpu 1 (sec)	CPU time / GPU Time
10^2	4.0	71.1	0.1
10^3	40.0	81.3	0.5
10^4	607.0	91.2	6.7
10^5	7793.0	129.6	60.1
10^6	78765.0	680.6	115.7

Other applications

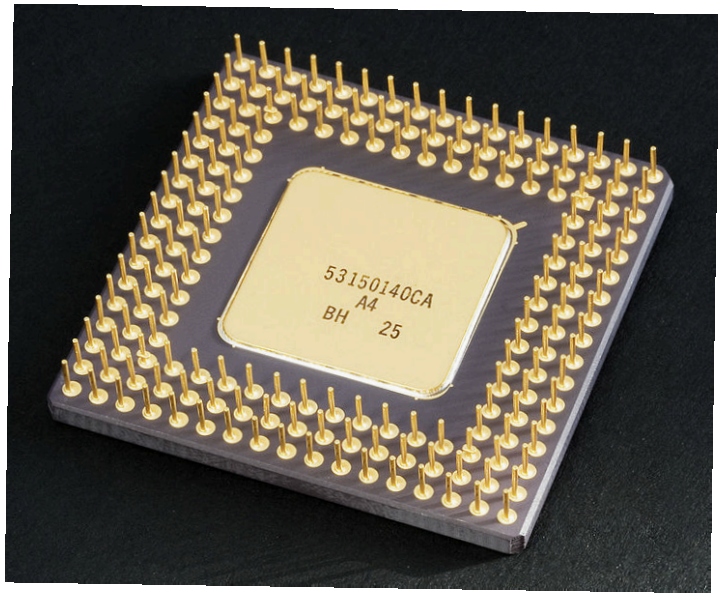
- Clustering
- Bootstrap
- Regression
- Matrix algebra
- EM Algorithm
- Rejection sampling
- Multiple testing
- Cross validation
- ...

Computer processors

- **Processing unit**: a computer chip that performs executive functions.
- **Core**: One of possibly many "sub-processors" placed on the same processing unit, each of which has the full functionality of the processing unit.

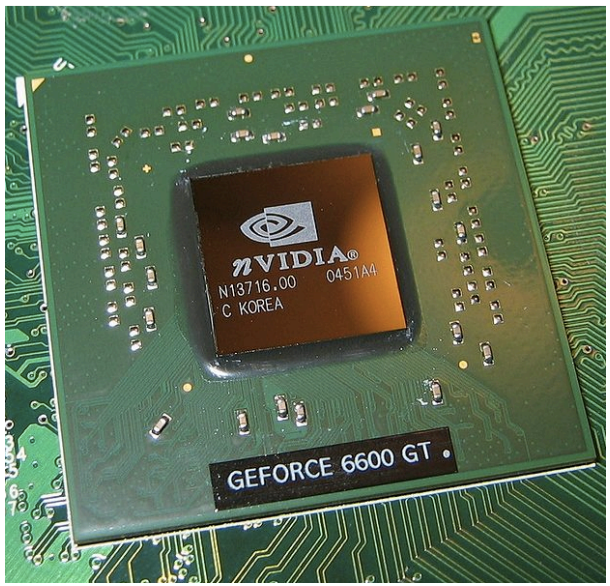
The Central Processing Unit (CPU)

- Regular computer processor.
- Allows parallelism, but not *massive* parallelism on its own.
- Usually contains 1 to 8 cores.
- Examples:
  - Intel 8086 (1979, x86)
  - Intel Core 2 Duo
  - Intel 80486DX2 (below)



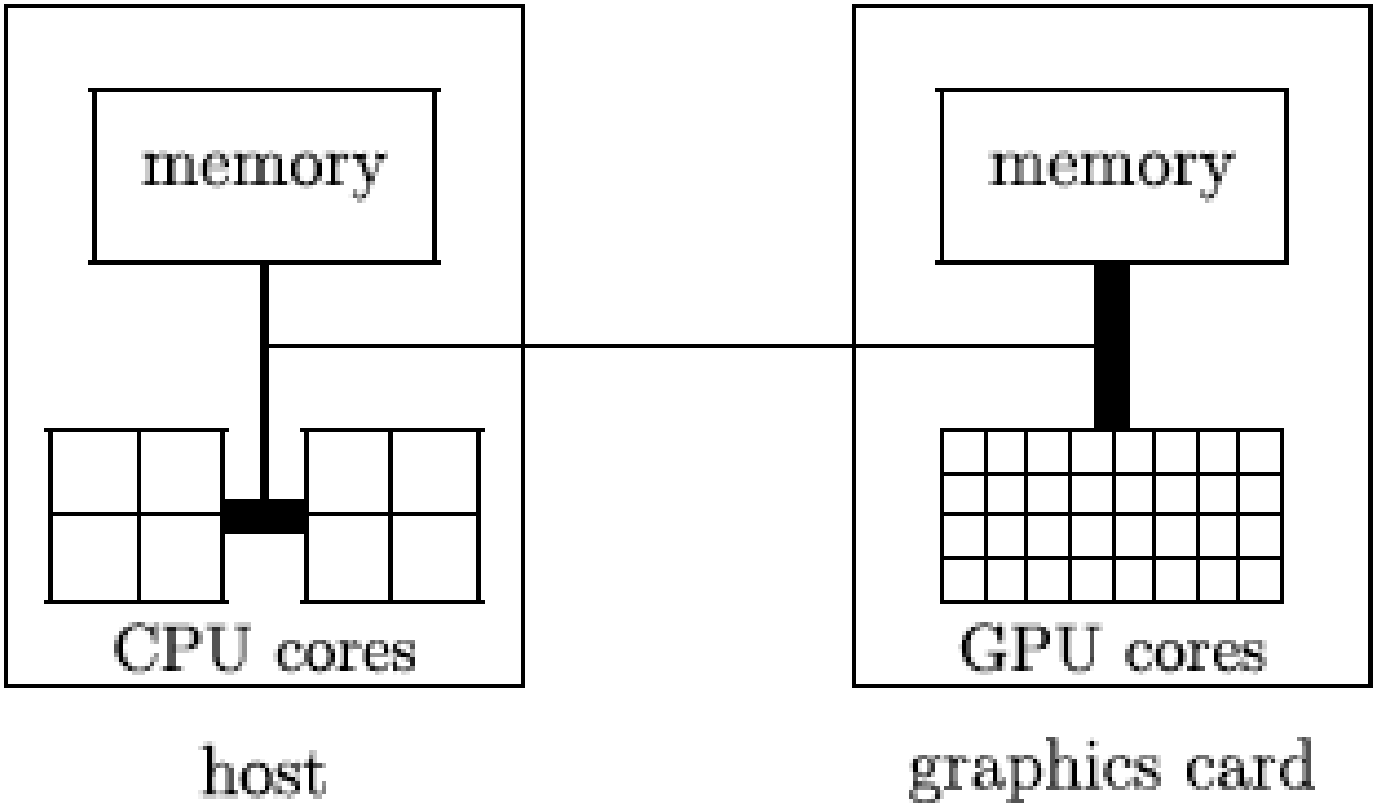
The Graphics Processing Unit (GPU)

- Processor in a video card or graphics card.
- Massively parallel: originally designed to speed up graphics throughput in video games.
- Cannot run by itself. Needs to be hooked up to a computer with a CPU.
- Contains several hundred cores.
- Higher memory bandwidth than a CPU.
- Examples:
  - NVIDIA GeForce 6600 (bottom left)
  - NVIDIA GeForce GTX 580
  - NVIDIA Tesla M2070 (on our GPU-enabled machines)



CPU / GPU cooperation

- The CPU ("host") is in charge.
- The CPU sends computationally intensive instruction sets to the GPU ("device") just like a human uses a pocket calculator.



More on parallelism

- **Parallelism**: the practice of running multiple calculations simultaneously.
- The architecture of GPUs is extremely well-suited to massively parallel workflows.
- Note: GPU parallelism is one of many kinds of parallelism. Others include:
  - Posix threads (CPU parallelism)
  - parallel cloud computing
  - openMP parallelism
  - openCL parallelism

GPU parallelism speeds up calculations

- Amdahl's Law says that the maximum theoretical speedup (CPU time / GPU time) is

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

  where:
  - P = fraction of the program's (in terms of execution time) that can be parallelized
  - N = number of parallel processors
- As $N \to \infty$, Amdahl's quantity goes to

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

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

Outline

CUDA: making a gaming toy do science

- GPUs were originally meant to speed up graphical displays for Windows OS and video games.

- **CUDA**: Compute Unified Device Architecture.
- Before CUDA, programmers could only program on GPUs using graphics languages, which are appropriate for video games but clumsy for science.
- CUDA devices support CUDA C, an extension of C for programs that use GPUs.

CUDA-enabled servers at Iowa State

- impact1.stat.iastate.edu (Red Hat Enterprise Linux Server release 6.2)
- impact2.stat.iastate.edu (CentOS release 6.3)
- impact3.stat.iastate.edu (Red Hat Enterprise Linux Server release 6.4)
- impact4.stat.iastate.edu (CentOS release 6.4)

Specs of our CUDA systems

- No graphical user interface or remote desktop capabilities. (Use the Linux command line.)
- 24 CPUs and 4 Tesla M2070 GPUs, where each GPU has 448 cores.:
- For more specs, log into impact1, 2, or 3 and enter into the command line:

```
cd /usr/local/NVIDIA_GPU_Computing_SDK/C/bin/linux/release
./deviceQuery
```

Logging in

- Open a command line program (Terminal in Linux and Mac OS, Cygwin or MinGW in Windows).
- Enter:

```
1 ssh -p 323 ISU_ID@impact1.stat.iastate.edu
```

- Note: in general, the port number for `ssh` is not always 323.
- Refer to http://www.tuxfiles.org/linuxhelp/cli.html or contact me at landau@iastate.edu for help with the Linux command line.
- Contact Stat IT at statit@iastate.edu or me if:
  - You can't log on, or
  - You want to be able to log on without entering your password every time, or
  - You want to shorten `ssh -p 323 ISU_ID@impact1.stat.iastate.edu` into a more manageable alias on your local machine.

Other resources for Linux command line tools and the Linux filesystem

- http://www.makeuseof.com/tag/an-introduction-to-the-linux-command-line/
- http://www.freesoftwaremagazine.com/articles/command_line_intro
- http://tldp.org/LDP/intro-linux/html/
- http://tldp.org/LDP/intro-linux/html/sect_03_01.html
- http://linux.die.net/Intro-Linux/chap_03.html
- http://linux.about.com/od/itl_guide/a/gdeitl28t02.htm

Important directories

- **/home/ISU_ID** Your private home folder on SMB (the collective storage system for all the stat dept linux servers). Files in here are stored remotely on SMB, not locally on impact1-3.
- **/Cyfiles/ISU_ID** Your private Cyfiles folder. Files in here are stored remotely on the Cyfiles server, not locally on impact1-3.
- **/tmp**
  - Everything in here is stored locally on impact1, etc., wherever you're logged in.
  - To avoid communicating over a network when you want fast computation, put large datasets here.
  - Note: **/tmp** automatically empties periodically.
- **/usr/local/NVIDIA_GPU_Computing_SDK**
  - Example CUDA C code. Stored locally on impact1, etc.
  - You do not have write privileges here.

Outline

GPU-enabled R packages

- `WideLM` - used to quickly fit a large number of linear models to a fixed design matrix and response vector.
- `magma` - a small linear algebra with implementations of backsolving and the LU factorization.
- `cudaBayesreg` - implements a Bayesian model for fitting fMRI data.
- `gcbd` - a Debian package for benchmarking linear algebra algorithms such as the QR, SVD and LU.factorizations.
- `gputools` - probably the most useful of these 5.

Contents of gputools

- Choose your device:

gputools function	CPU analog	Same usage?
chooseGpu()	none	NA
getGpuId()	none	NA

- Linear algebra:

gputools function	CPU analog	Same usage?
gpuDist()	dist()	no
gpuMatMult()	%*% operator	no
gpuCrossprod()	crossprod()	yes
gpuTcrossprod()	tcrossprod()	yes
gpuQr()	qr()	almost
gpuSolve()	solve()	no
gpuSvd()	svd()	almost

- Simple model fitting:

gputools function	CPU analog	Same usage?
gpuLm()	lm()	yes
gpuLsfit()	lsfit()	yes
gpuGlm()	glm()	yes
gpuGlm.fit()	glm.fit()	yes

- Hypothesis testing:

gputools function	CPU analog	Same usage?
gpuTtest()	t.test()	no
getAucEstimate()	???	???

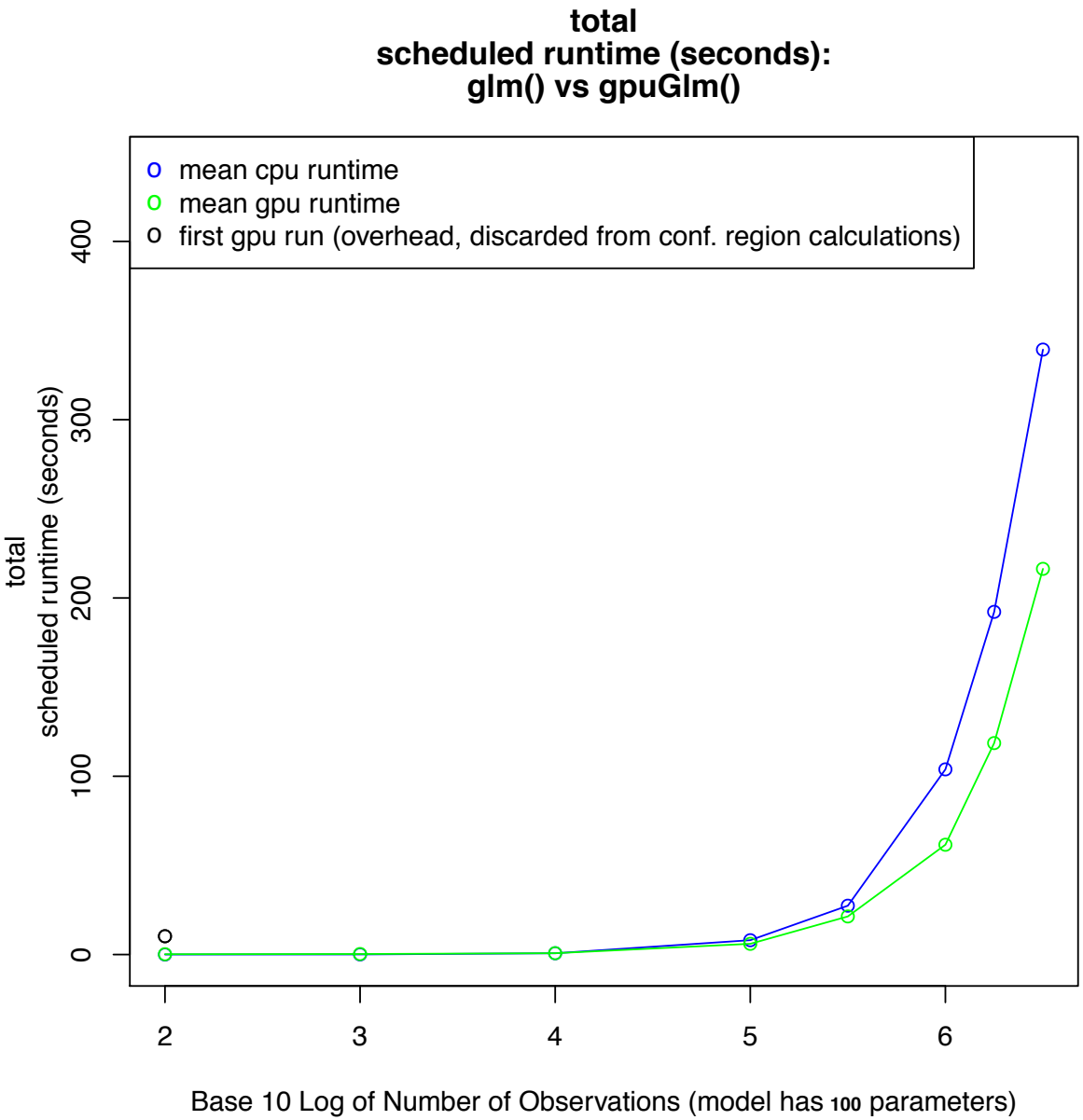
- Other routines:

gputools function	CPU analog	Same usage?
gpuHclust()	hclust()	no
gpuDistClust()	hclust(dist())	no
gpuFastICA()	fastICA() (fastICA package)	yes
gpuGranger()	grangertest() (lmtest package)	no
gpuMi()	???	???
gpuSvmPredict()	www.jstatsoft.org/v15/i09/paper	no
gpuSvmTrain()	www.jstatsoft.org/v15/i09/paper	no

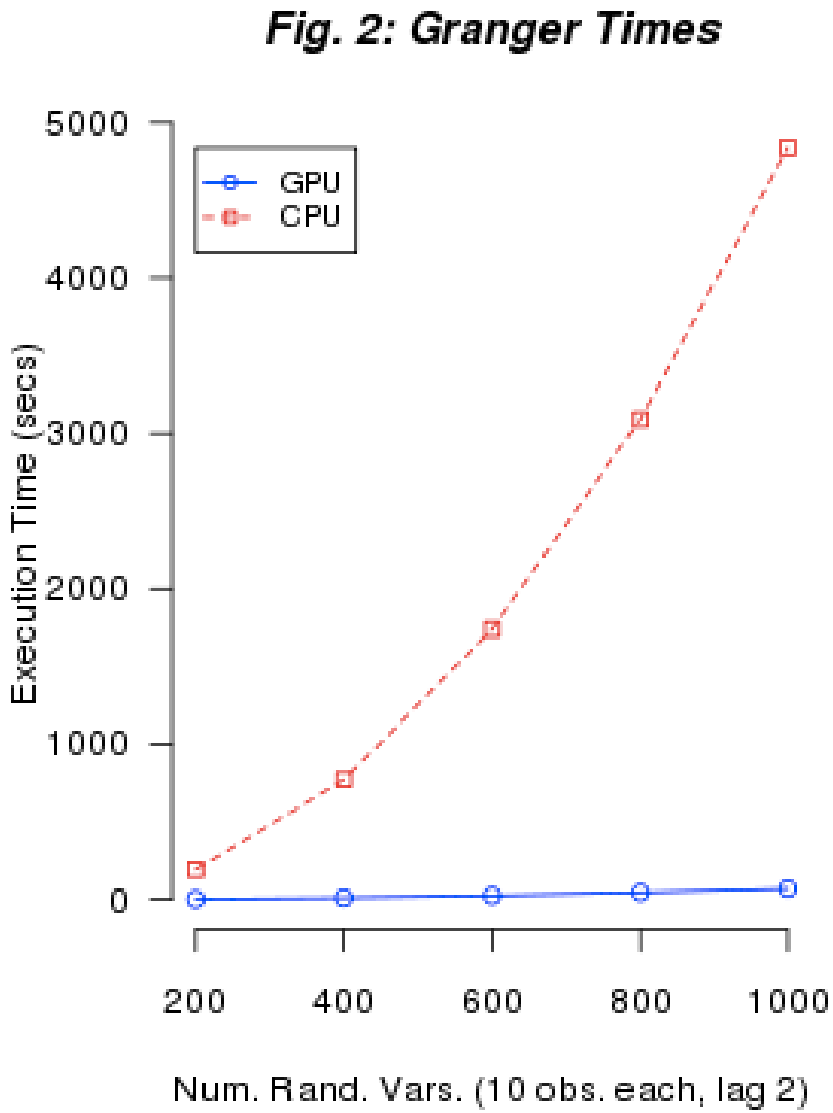
Example

```
> getGpuID()
 [1] 0
> chooseGpu(3)
 [[1]]
 [1] 3
> getGpuID()
 [1] 3
> A <- matrix(runif(1e7), nrow = 1e4)
> B <- matrix(runif(1e7), nrow = 1e4)
> ptm <- proc.time(); C <- gpuMatMult(A, B);
> proc.time() - ptm
    user   system  elapsed
   2.959   2.190     5.159
> ptm <-  proc.time(); C <- A %*% B;
> proc.time() - ptm
    user   system  elapsed
 116.389   0.166   116.503
```

Speedup



Speedup



Tentative syllabus for the series

1. Intro and `gputools`
2. A codeless intro to GPU parallelism
3. Intro to CUDA C
4. CUDA C: K-means and MCMC
5. CUDA C: Shared memory and performance measurement
6. CUDA C: Race conditions, atomics, and warps
7. CUBLAS and CULA: linear algebra libraries for CUDA C
8. CURAND: a GPU-enabled library for fast random number generation
9. Thrust: the GPU analog of the C++ Standard Template Library
10. Intro to Python: preparation for PyCUDA
11. PyCUDA: a Python module for GPU computing

Outline

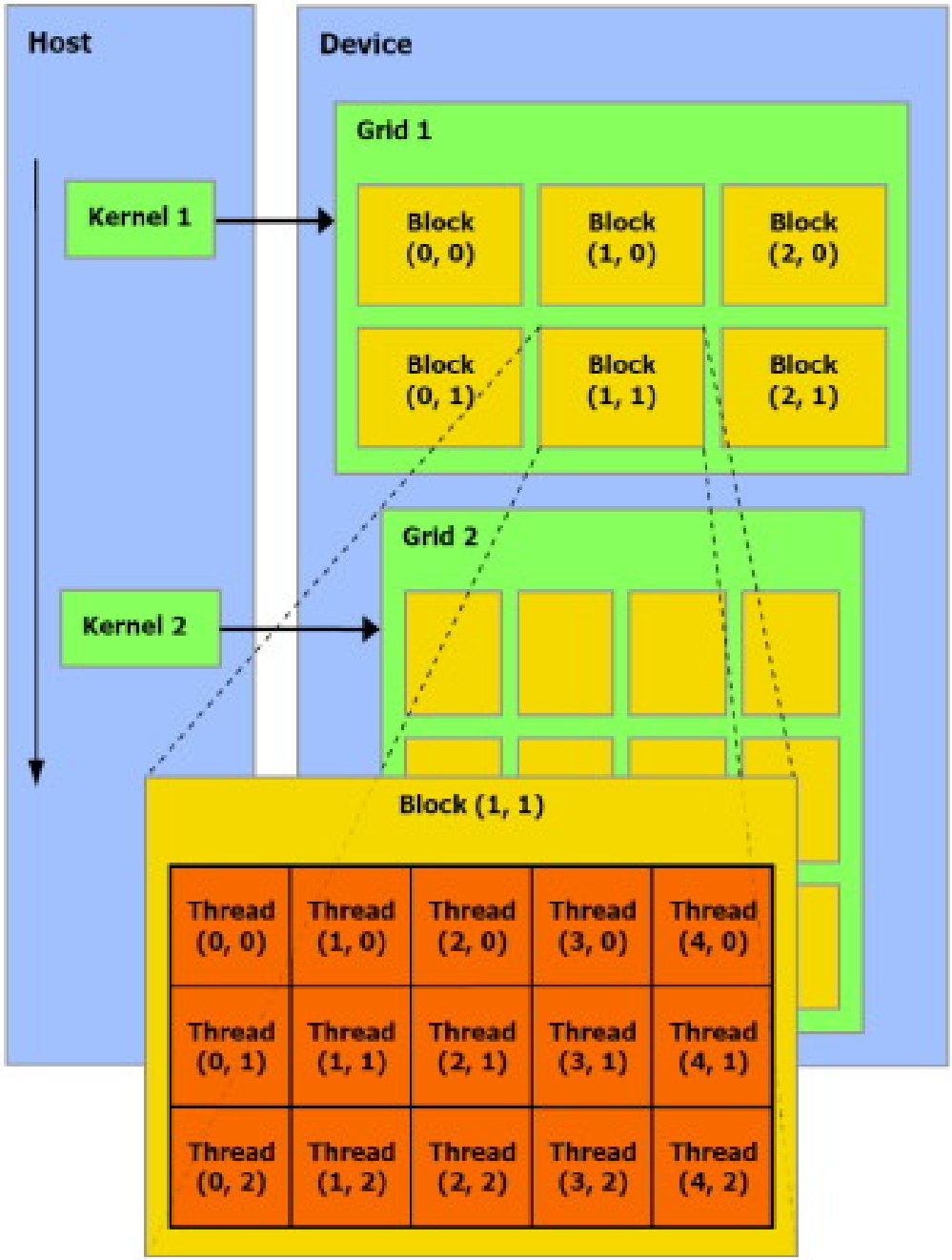
How GPU parallelism works

1. The CPU sends a command called a **kernel** to a GPU.
2. The GPU executes several duplicate realizations of this command, called **threads**.
   - These threads are grouped into bunches called **blocks**.
   - The sum total of all threads in a kernel is called a **grid**.

- Toy example:
  - CPU says something like, "Hey, GPU. Sum pairs of adjacent numbers. Use the array, (1, 2, 3, 4, 5, 6, 7, 8). Use 2 blocks of 2 threads each."
  - GPU thinks: "Sum pairs of adjacent numbers" is a kernel that I need to apply to the array, (1, 2, 3, 4, 5, 6, 8).
  - The GPU spawns 2 blocks, each with 2 threads:

Block	0		1	
Thread	0	1	0	1
Action	$1 + 2$	$3 + 4$	$5 + 6$	$7 + 8$

- All four actions above happen simultaneously.
- I could have also used 1 block with 4 threads and given the threads different pairs of numbers.



Outline

Vector addition

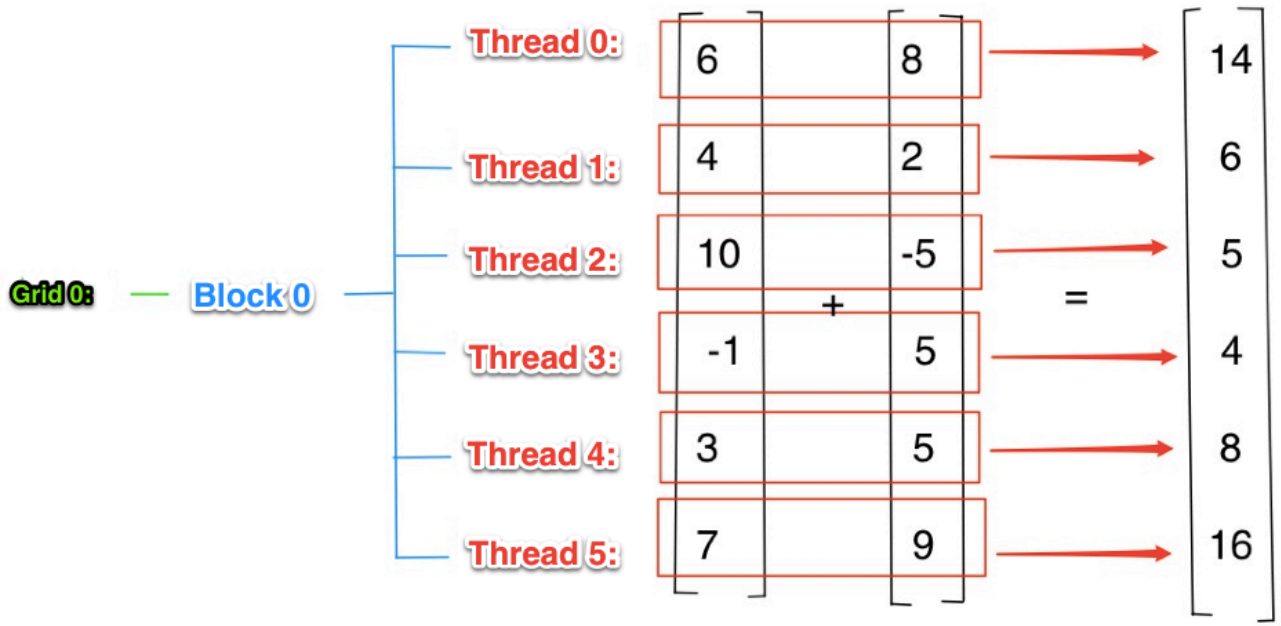
- Say I have 2 vectors,

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

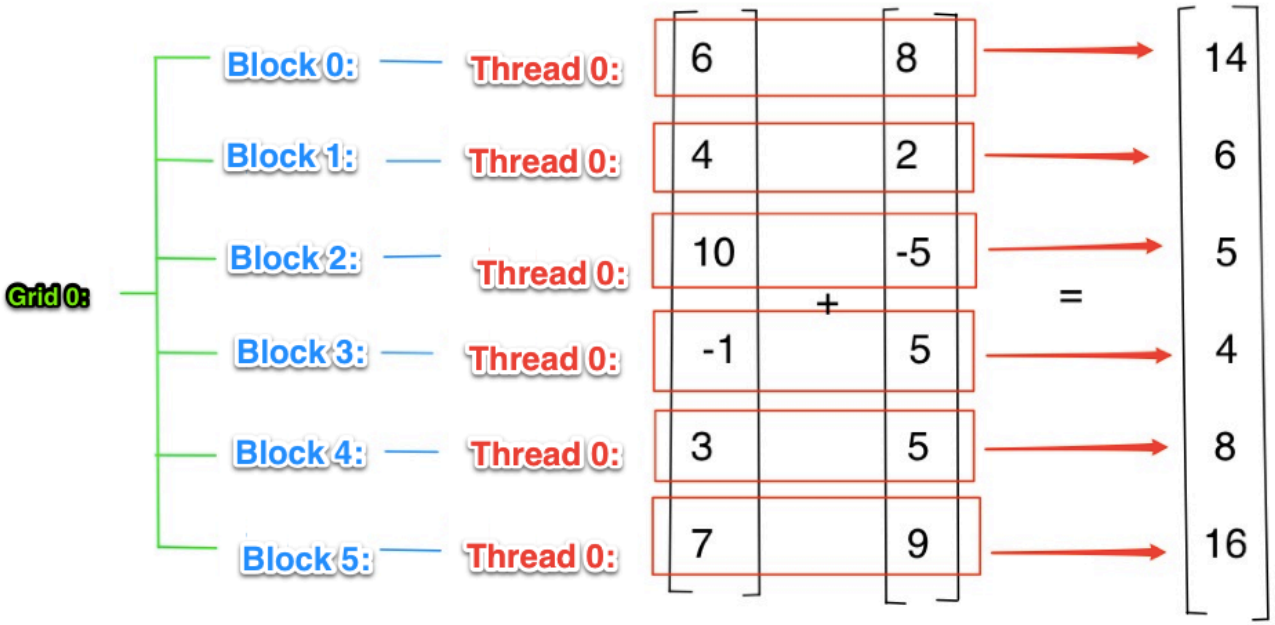
- I want to compute their component-wise sum,

$$c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_n + b_n \end{bmatrix}$$

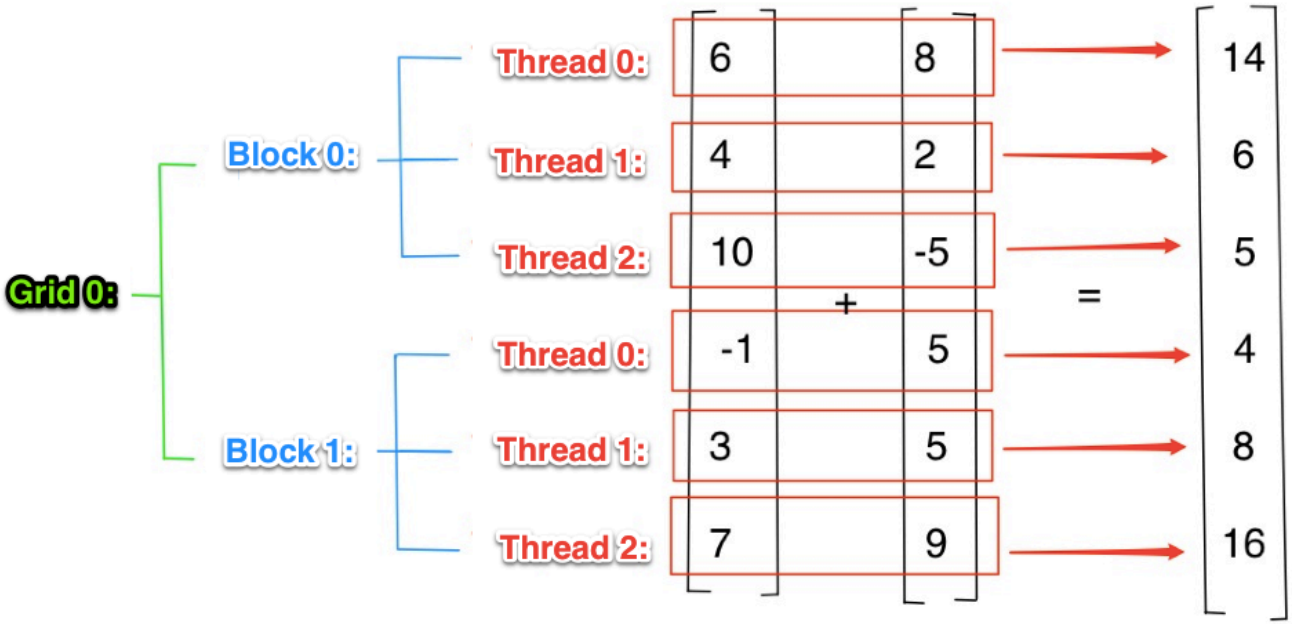
Vector addition



Vector addition



Vector addition



Vector addition: vectorsums.cu

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <cuda.h>
4  #include <cuda_runtime.h>
5
6  #define N 10
7
8  __global__ void add(int *a, int *b, int *c){
9    int bid = blockIdx.x;
10   if(bid < N)
11     c[bid] = a[bid] + b[bid];
12 }
13
14 int main(void) {
15   int i, a[N], b[N], c[N];
16   int *dev_a, *dev_b, *dev_c;
17
18   cudaMalloc((void**) &dev_a, N*sizeof(int));
19   cudaMalloc((void**) &dev_b, N*sizeof(int));
20   cudaMalloc((void**) &dev_c, N*sizeof(int));
21
22   for(i=0; i<N; i++){
23     a[i] = -i;
24     b[i] = i*i;
25   }
26
27   cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
28   cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);
```

Vector addition: vectorsums.cu

```
29    add<<<N,1>>>(dev_a, dev_b, dev_c);
30
31    cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);
32
33    printf("\na + b = c\n");
34    for(i = 0; i<N; i++){
35       printf("%5d + %5d = %5d\n", a[i], b[i], c[i]);
36    }
37
38    cudaFree(dev_a);
39    cudaFree(dev_b);
40    cudaFree(dev_c);
41 }
```

Compiling and running `vectorsums.cu`

```
> nvcc vectorsums.cu -o vectorsums
> ./vectorsums
a + b = c
    0 +    0 =    0
   -1 +    1 =    0
   -2 +    4 =    2
   -3 +    9 =    6
   -4 +   16 =   12
   -5 +   25 =   20
   -6 +   36 =   30
   -7 +   49 =   42
   -8 +   64 =   56
   -9 +   81 =   72
```

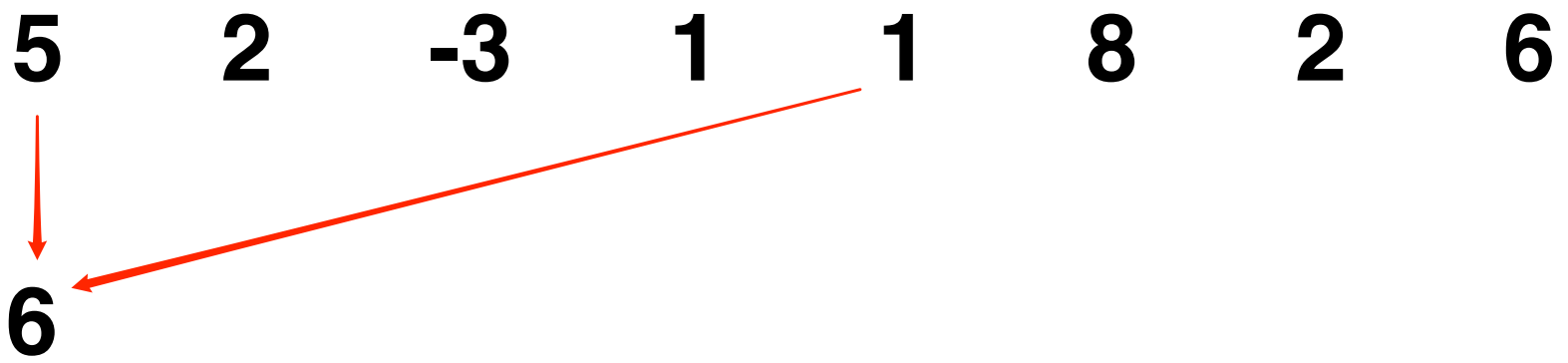
Pairwise sum

- Let's take the pairwise sum of the vector,

$$(5, 2, -3, 1, 1, 8, 2, 6)$$

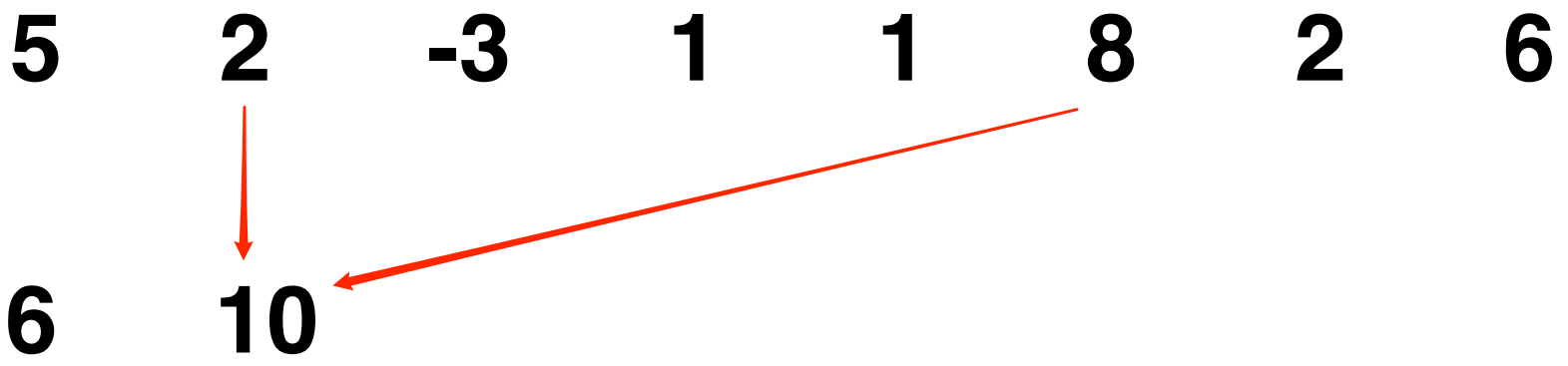
using 1 block of 4 threads.

Pairwise sum



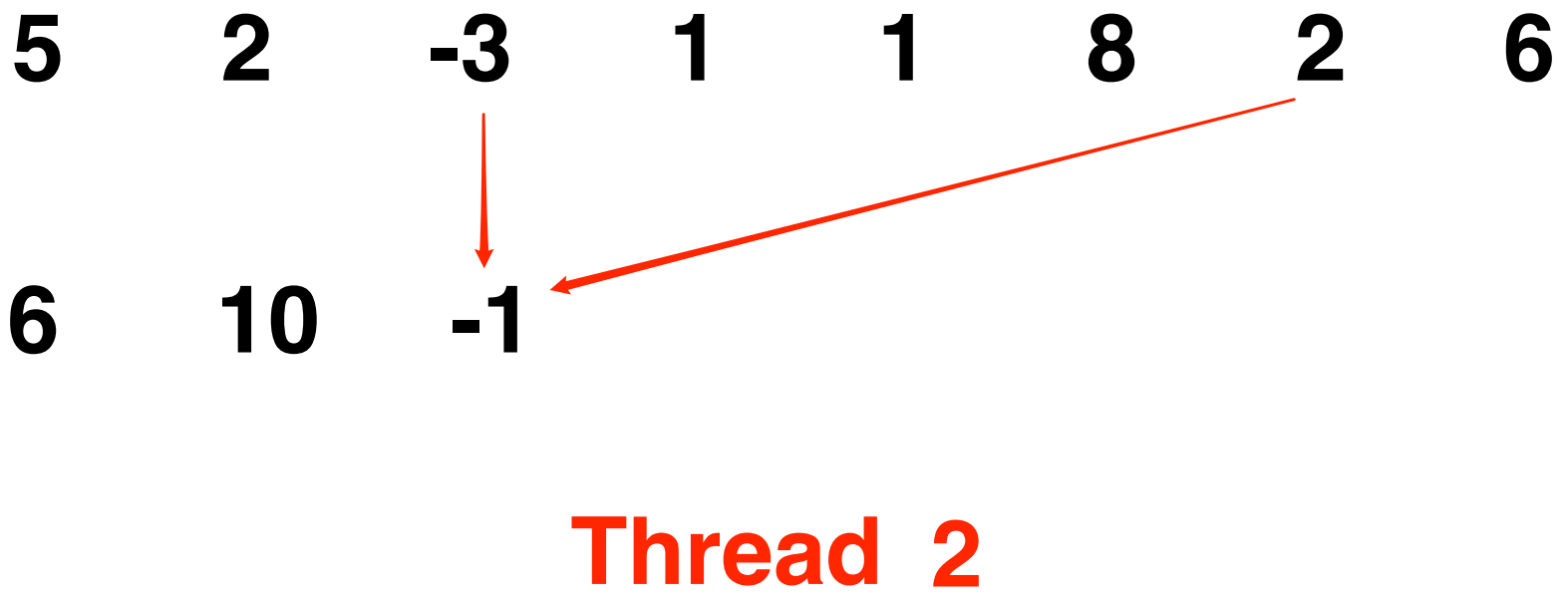
Thread 0

Pairwise sum

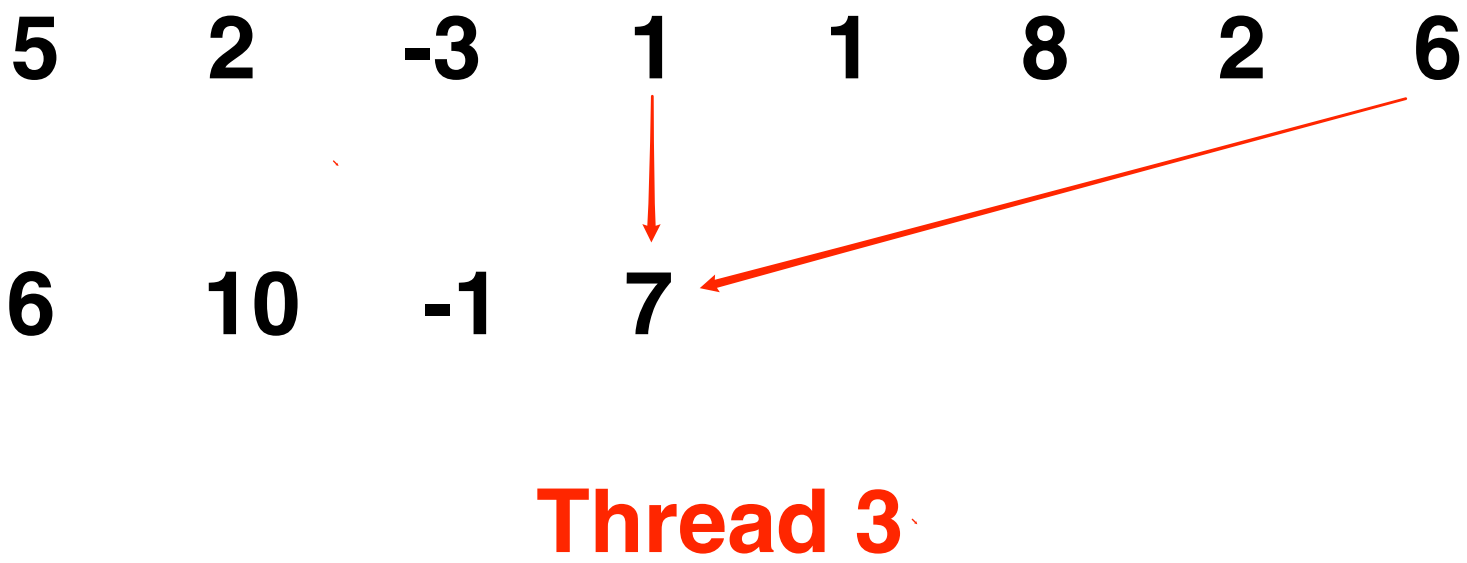


Thread 1

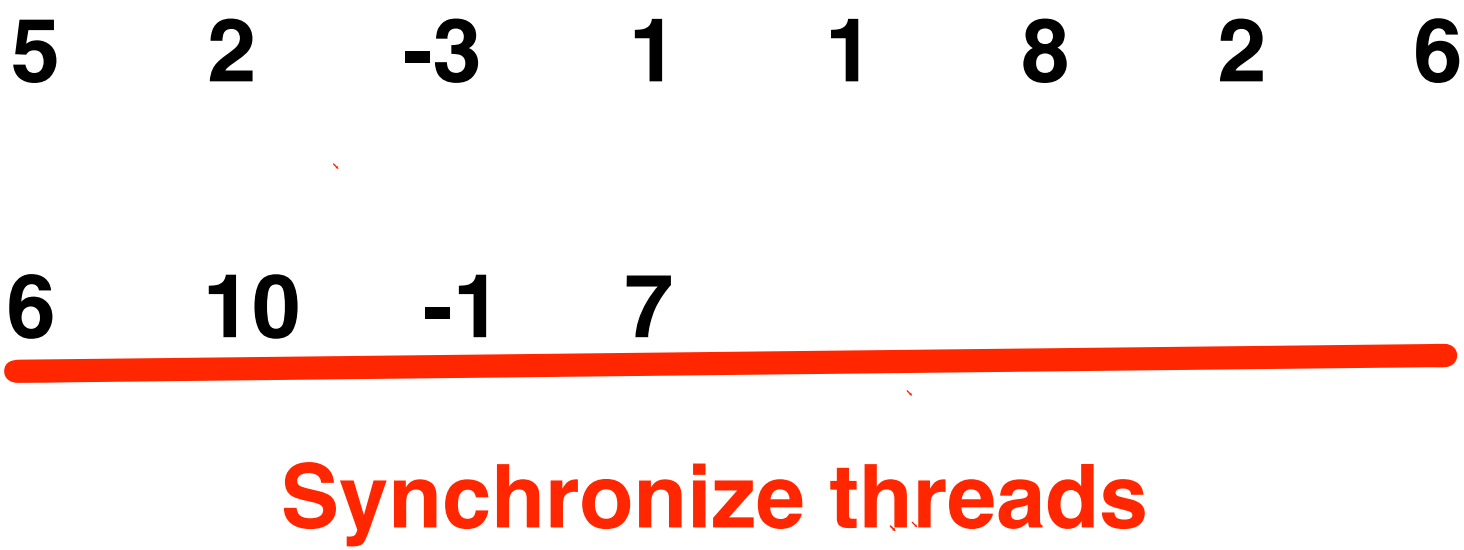
Pairwise sum



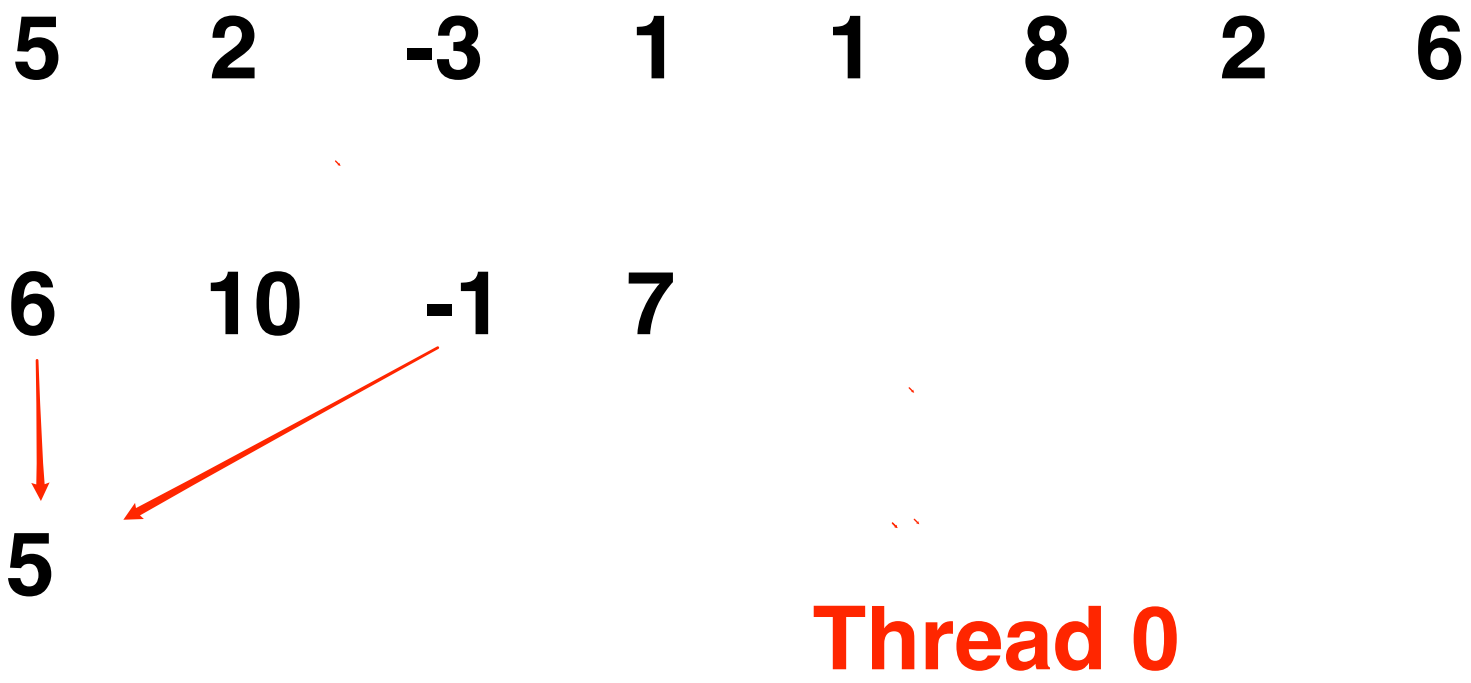
Pairwise sum



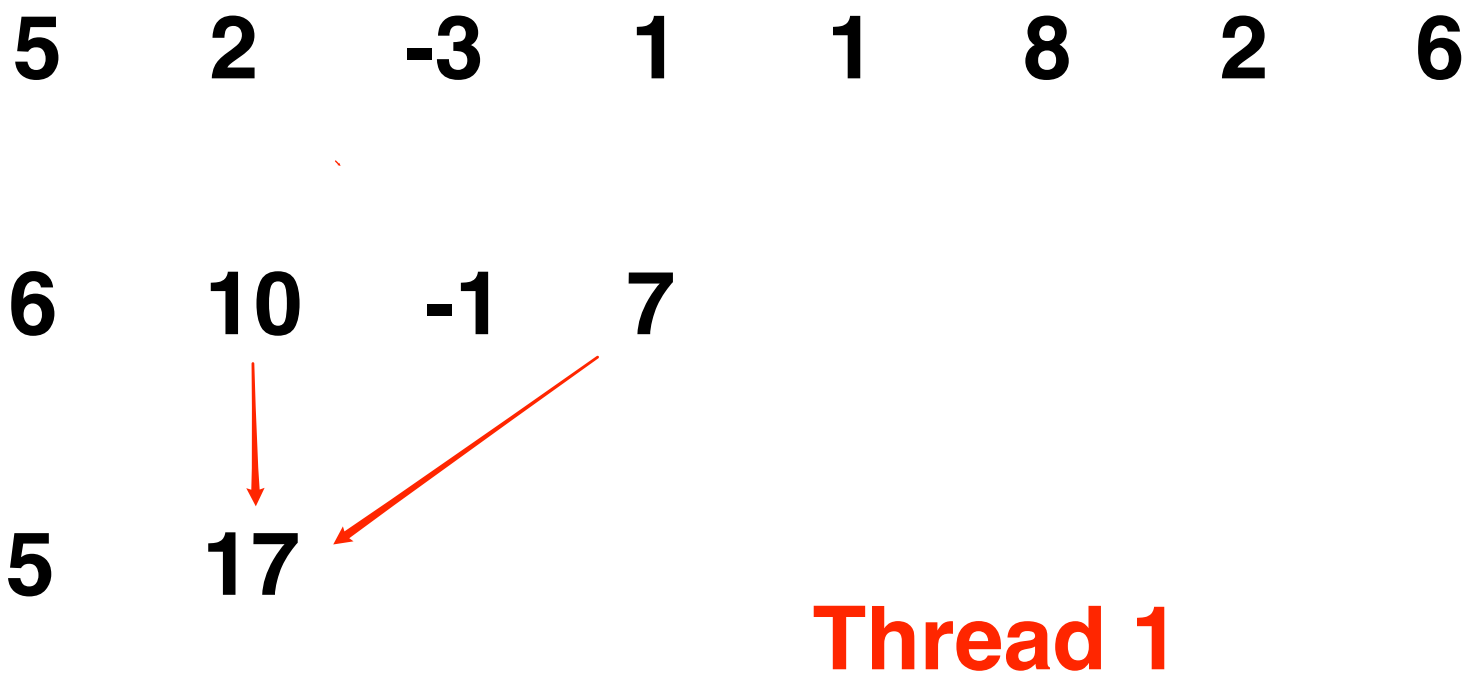
Pairwise sum



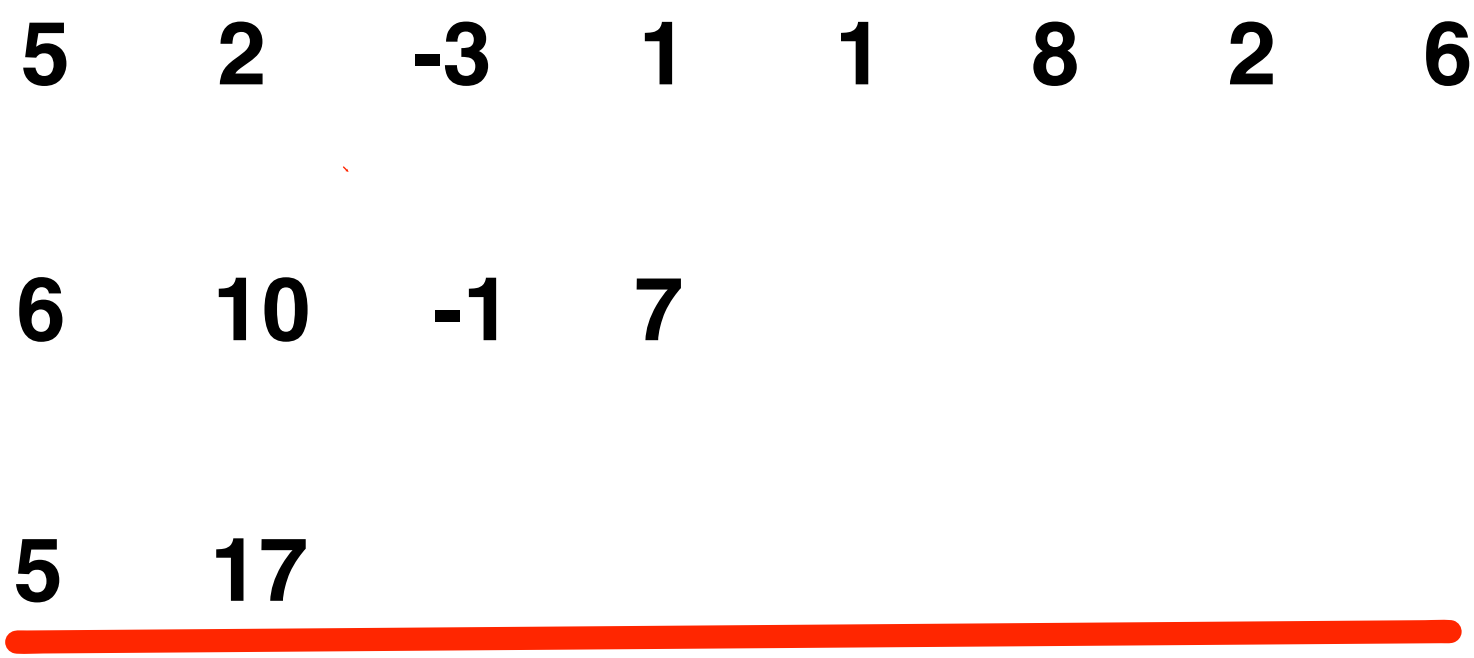
Pairwise sum



Pairwise sum



Pairwise sum



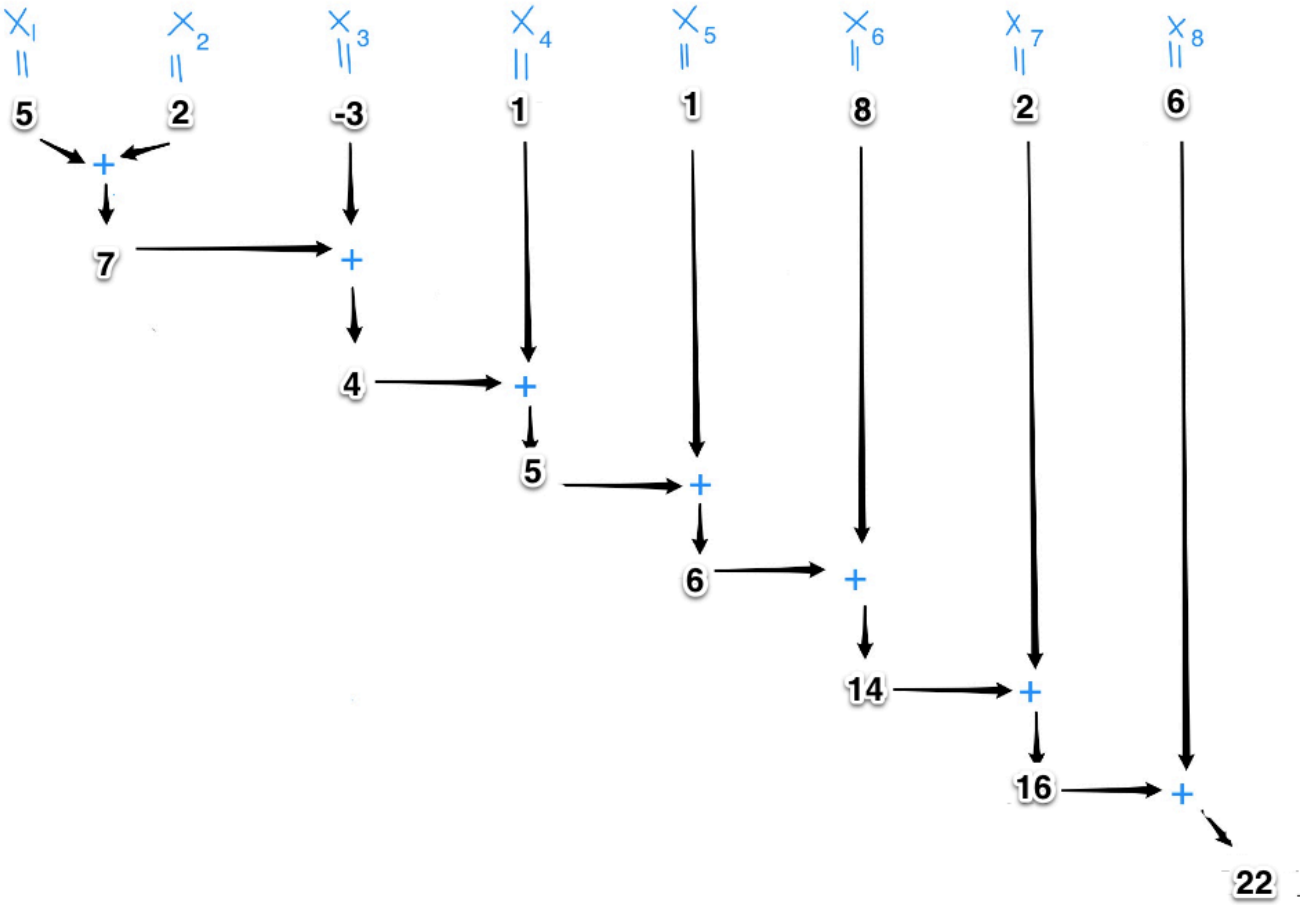
Synchronize Threads

Pairwise sum

Synchronizing threads

- **Synchronization**: waiting for all parallel tasks to reach a checkpoint before allowing any of then to continue.
  - Threads from the same block can be synchronized easily.
  - In general, do not try to synchronize threads from different blocks. It's possible, but extremely inefficient.

Compare the pairwise sum to the sequential sum



- The pairwise sum requires only $\log_2(n)$ sequential stages, while the sequential sum requires $n - 1$ sequential steps.

pairwise_sum.cu

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <cuda.h>
5  #include <cuda_runtime.h>
6
7  /*
8   * This program computes the sum of the elements of
9   * vector v using the pairwise (cascading) sum algorithm.
10  */
11
12 #define N 8 // length of vector v. MUST BE A POWER OF 2!!!
13
14 // Fill the vector v with n random floating point numbers.
15 void vfill(float* v, int n){
16   int i;
17   for(i = 0; i < n; i++){
18     v[i] = (float) rand() / RAND_MAX;
19   }
20 }
21
22 // Print the vector v.
23 void vprint(float* v, int n){
24   int i;
25   printf("v = \n");
26   for(i = 0; i < n; i++){
27     printf("%7.3f\n", v[i]);
28   }
29   printf("\n");
30 }
```

pairwise_sum.cu

```
31 // Pairwise-sum the elements of vector v and store the result in v[0].
32 __global__ void psum(float* v){
33   int t = threadIdx.x; // Thread index.
34   int n = blockDim.x; // Should be half the length of v.
35
36   while (n != 0) {
37     if(t < n)
38       v[t] += v[t + n];
39     __syncthreads();
40     n /= 2;
41   }
42 }
43
44 int main (void){
45   float *v_h, *v_d; // host and device copies of our vector,
         respectively
46
47   // dynamically allocate memory on the host for v_h
48   v_h = (float*) malloc(N * sizeof(*v_h));
49
50   // dynamically allocate memory on the device for v_d
51   cudaMalloc ((float**) &v_d, N *sizeof(*v_d));
52
53   // Fill v_h with N random floating point numbers.
54   vfill(v_h, N);
55
56   // Print v_h to the console
57   vprint(v_h, N);
```

pairwise_sum.cu

```
58    // Write the contents of v_h to v_d
59    cudaMemcpy( v_d, v_h, N * sizeof(float), cudaMemcpyHostToDevice );
60
61    // Compute the pairwise sum of the elements of v_d and store the
          result in v_d[0].
62    psum<<< 1, N/2 >>>(v_d);
63
64    // Write the pairwise sum, v_d[0], to v_h[0].
65    cudaMemcpy(v_h, v_d, sizeof(float), cudaMemcpyDeviceToHost );
66
67    // Print the pairwise sum.
68    printf("Pairwise sum = %7.3f\n", v_h[0]);
69
70    // Free dynamically-allocated host memory
71    free(v_h);
72
73    // Free dynamically-allocated device memory
74    cudaFree(v_d);
75 }
```

Compiling and running pairwise_sum.cu

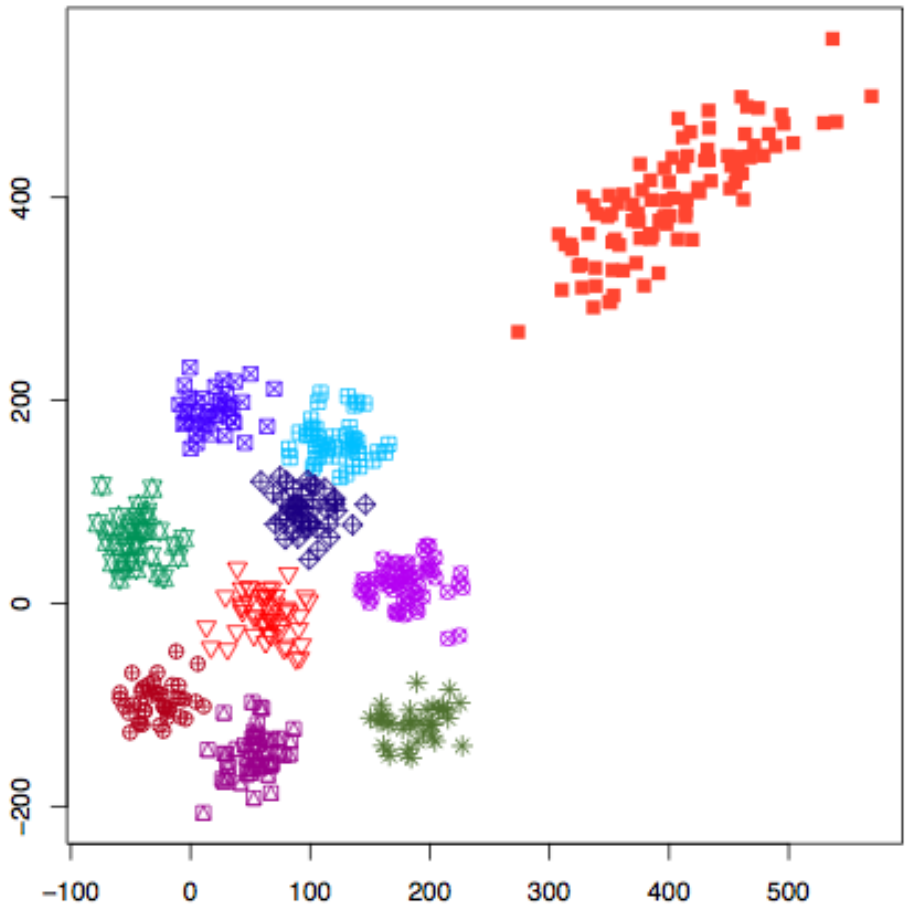
```
> nvcc pairwise_sum.cu -o pairwise_sum
> ./pairwise_sum
v =
   0.840
   0.394
   0.783
   0.798
   0.912
   0.198
   0.335
   0.768

Pairwise sum =   5.029
```

Reductions and scans

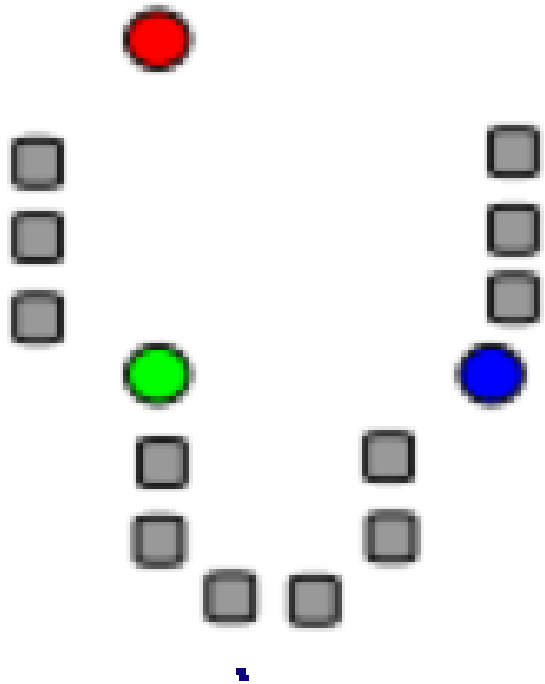
- Reductions
  - Pairwise sum and pairwise multiplication are examples of reductions.
  - **Reduction**: an algorithm that applies some binary operation on a vector to produce a scalar.
- Scans
  - **Scan (prefix sum)**: an operation on a vector that produces a sequence of partial reductions.
  - Example: computing the sequence of partial sums in pairwise fashion.

Lloyd's K-means algorithm

- Cluster N vectors in Euclidian space into K groups.

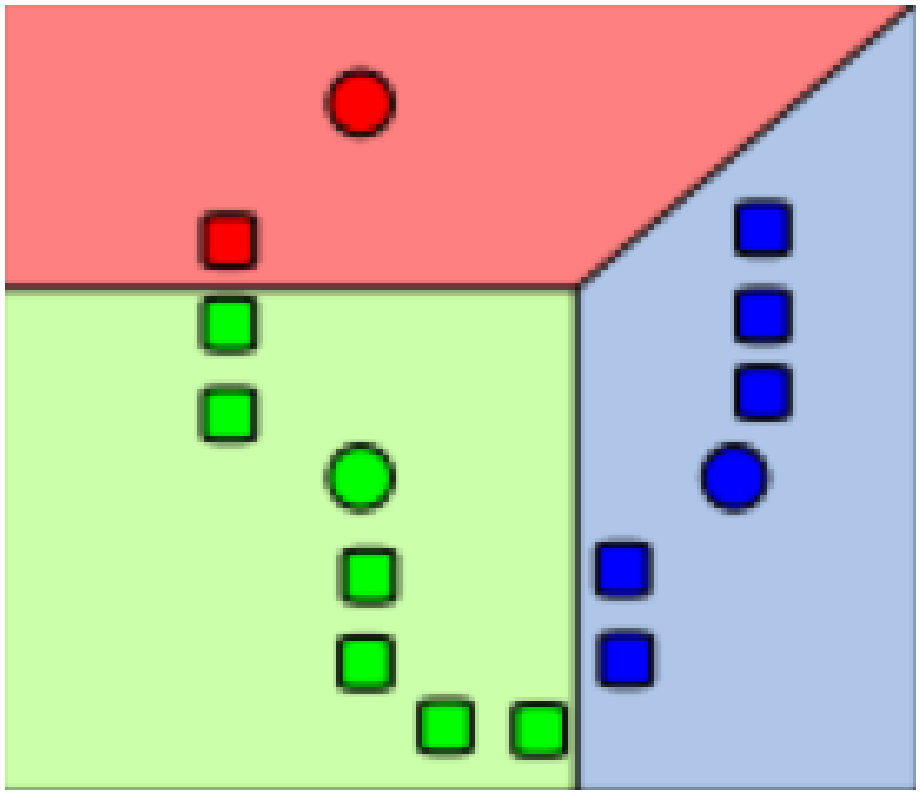


Step 1: choose initial cluster centers.

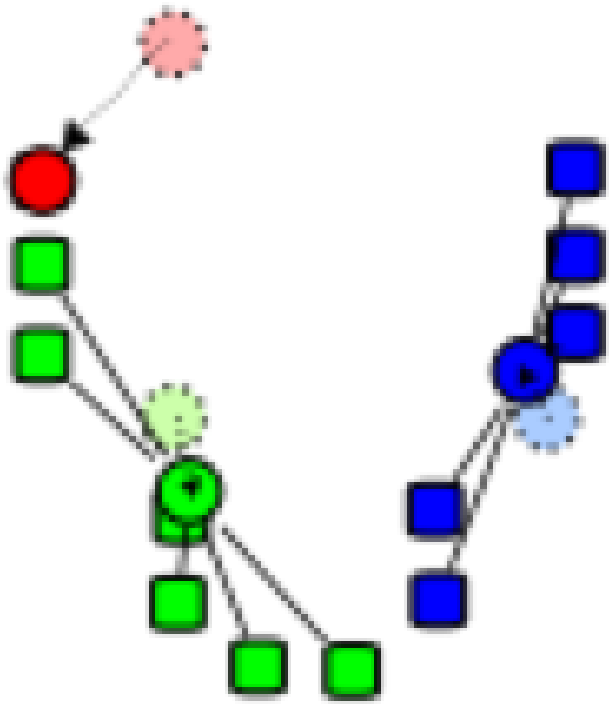


- The circles are the cluster means, the squares are the data points, and the color indicates the cluster.

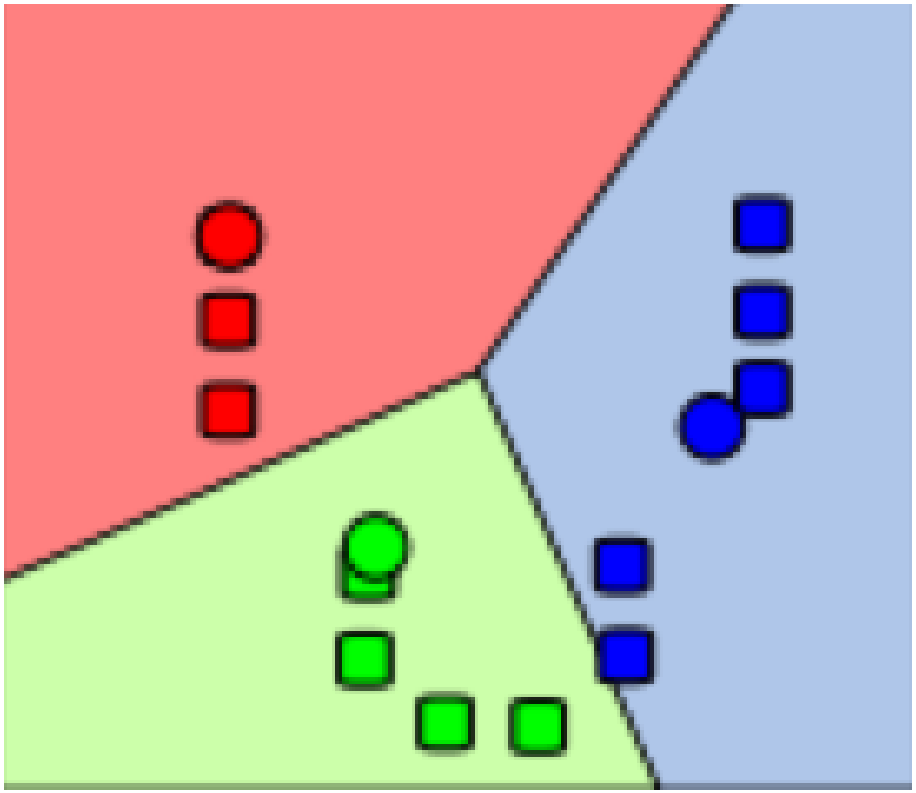
Step 2: assign each data point (square) to its closest center (circle).



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)
```

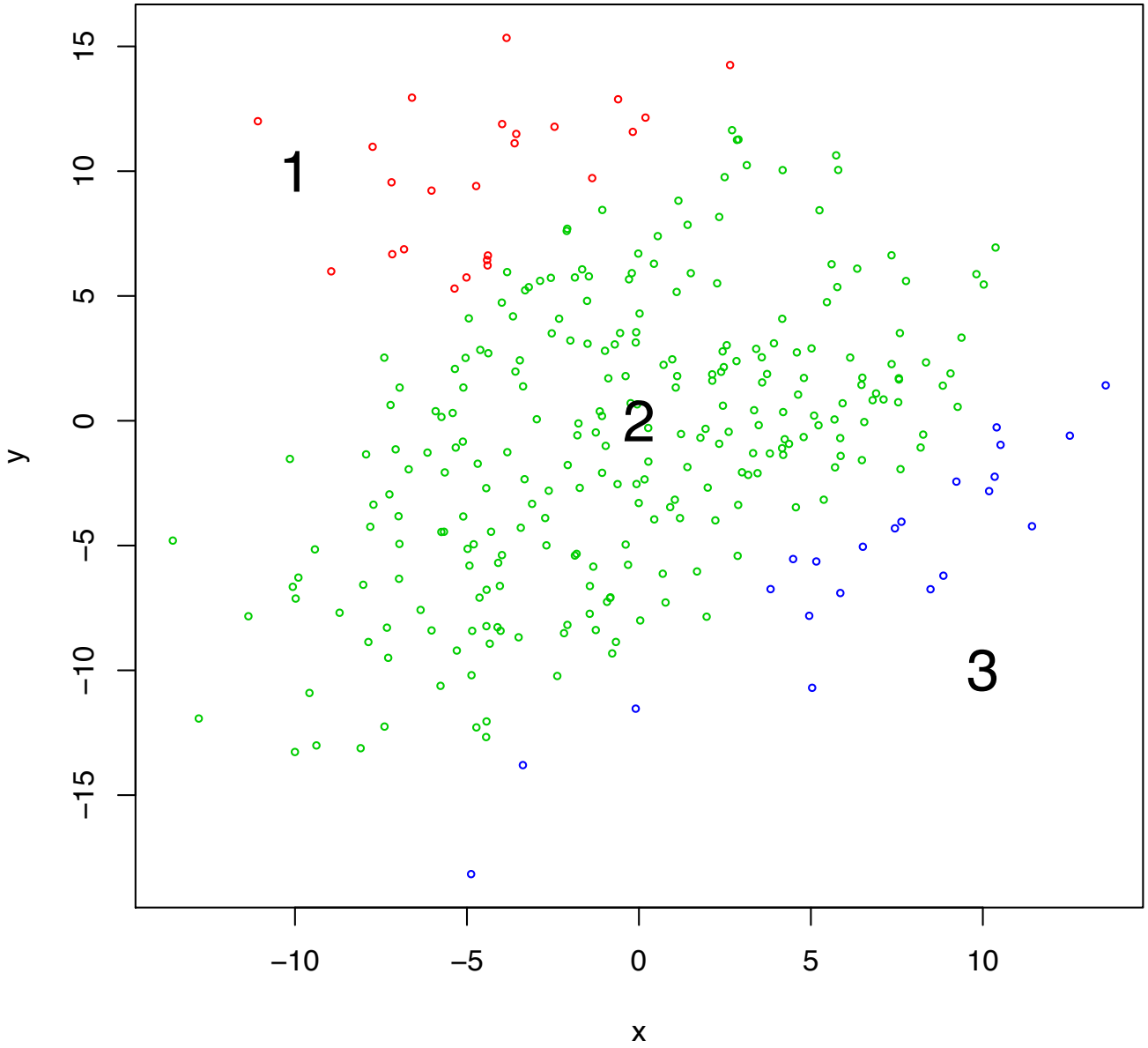
Compile, test, and run.

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

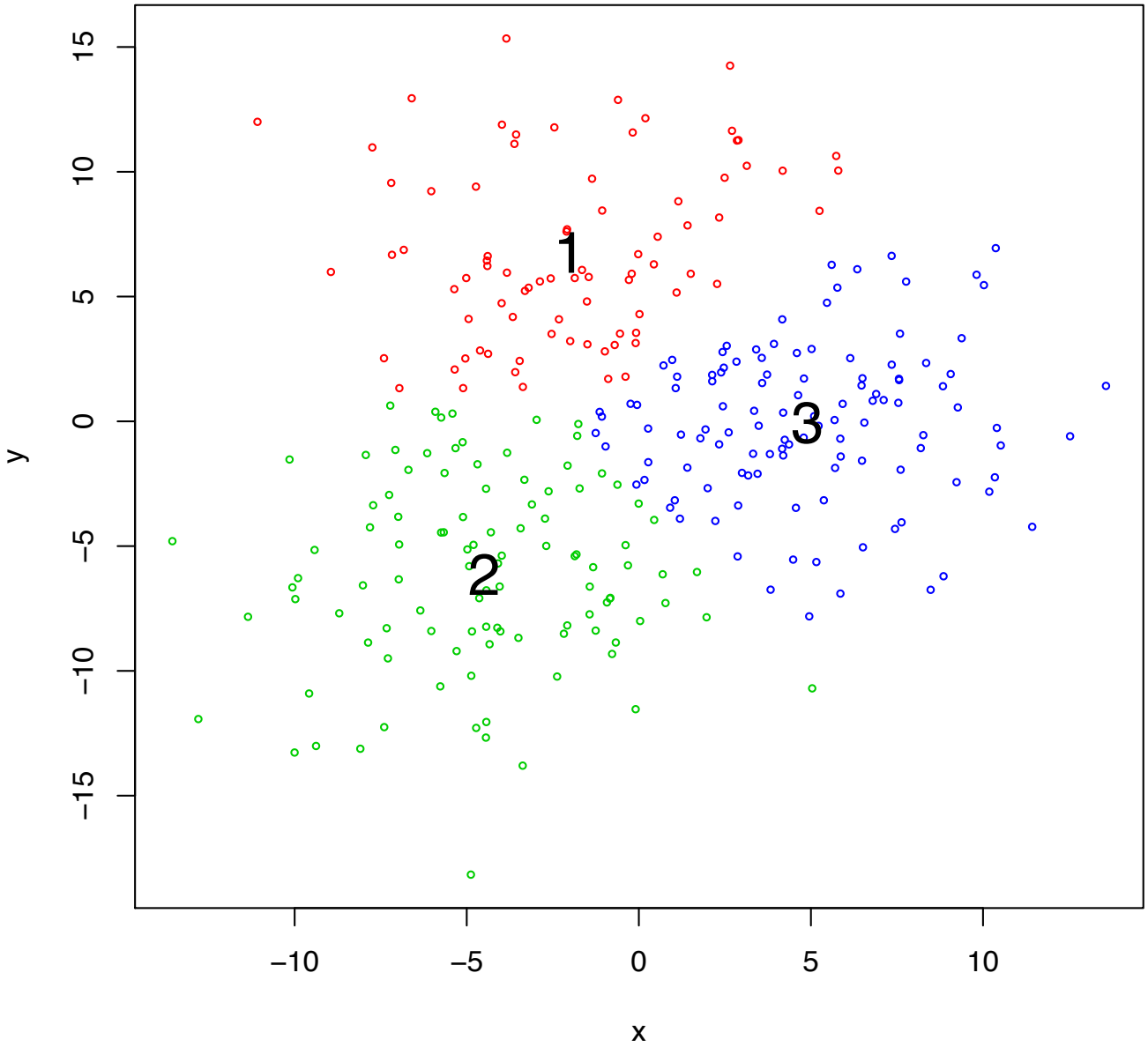
- Run for real.

```
> ./gpu_kmeans
```

Initial clustering: 300 points, 3 clusters



Final clustering after 10 iterations



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.

```
> valgrind ./gibbs_metropolis smallData.txt 10
==12942== Memcheck, a memory error detector
==12942== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==12942== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==12942== Command: ./gibbs_metropolis smallData.txt 10
==12942==
==12942== Warning: set address range perms: large range [0x800000000, 0x2100000000) (noaccess)
==12942== Warning: set address range perms: large range [0x2100000000, 0x2800000000) (noaccess)
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
```

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
```

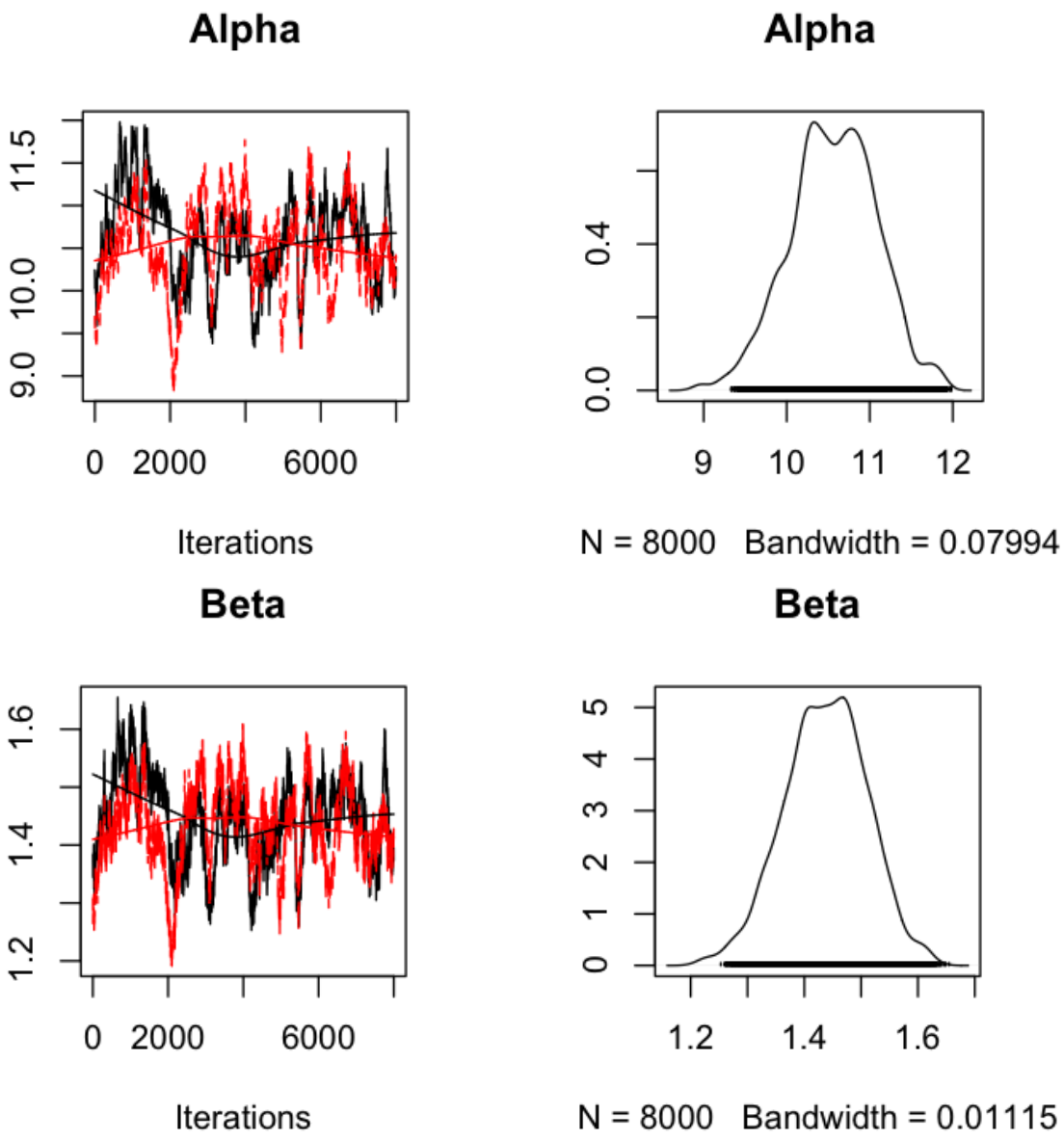
Diagnostics: Gelman-Rubin potential scale reduction factor

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

- 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



Example: `ols.cu`

- I will attempt to solve the least squares problem,

$$y = X\beta + \varepsilon$$

by computing the solution,

$$\widehat{\beta} = (X^T X)^{-1} X^T y$$

Example: ols.cu

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <cuda_runtime.h>
5  #include <cublas_v2.h>
6  #include <cula.h>
7  #include <math.h>
8
9  #define I(i, j, ld) j * ld + i
10
11 #define CUDA_CALL(x) {if((x) != cudaSuccess){ \
12    printf("CUDA error at %s:%d\n",__FILE__,__LINE__); \
13    printf("  %s\n", cudaGetErrorString(cudaGetLastError())); \
14    exit(EXIT_FAILURE);}}
15
16 float rnorm(){
17    float r1 = ((float) rand()) / ((float) RAND_MAX);
18    float r2 = ((float) rand()) / ((float) RAND_MAX);
19    return sqrt( -2 * log(r1) ) * cos(2 * 3.1415 * r2);
20 }
21
22 int main(){
23    int i, j;
24    int n = 10;
25    int p = 3;
26    int* ipiv;
27    float k;
28    float *X, *XtX, *XtY, *beta, *Y, *dX, *dXtX, *dXtY, *dbeta, *dY;
```

Example: ols.cu

```
29    float *a, *b;
30    a = (float*) malloc(sizeof(*X));
31    b = (float*) malloc(sizeof(*X));
32    *a = 1.0;
33    *b = 0.0;
34
35    cublasHandle_t handle;
36    cublasCreate(&handle);
37
38    X = (float*) malloc(n * p * sizeof(*X));
39    XtX = (float*) malloc(p * p * sizeof(*X));
40    XtY = (float*) malloc(p * sizeof(*X));
41    beta = (float*) malloc(p * sizeof(*X));
42    Y = (float*) malloc(n * sizeof(*X));
43
44    CUDA_CALL(cudaMalloc((void**) &ipiv, p * p * sizeof(*ipiv)));
45    CUDA_CALL(cudaMalloc((void**) &dX, n * p * sizeof(*X)));
46    CUDA_CALL(cudaMalloc((void**) &dXtX, p * p * sizeof(*X)));
47    CUDA_CALL(cudaMalloc((void**) &dXtY, p * sizeof(*X)));
48    CUDA_CALL(cudaMalloc((void**) &dbeta, p * sizeof(*X)));
49    CUDA_CALL(cudaMalloc((void**) &dY, n * sizeof(*X)));
```

Example: ols.cu

```
51    printf("Y\t\tX\n");
52    for(i = 0; i < n; i++){
53      k = (float) i;
54      X[I(i, 0, n)] = 1.0;
55      X[I(i, 1, n)] = k / 10.0;
56      X[I(i, 2, n)] = k * k / 10.0;
57      Y[i] = (k - 5.0) * (k - 2.3) / 3.0 + rnorm();
58
59      printf("%0.2f\t\t", Y[i]);
60      for(j = 0; j < p; j++){
61        printf("%0.2f\t", X[I(i, j, n)]);
62      }
63      printf("\n");
64    }
65    printf("\n");
66
67    CUDA_CALL(cudaMemcpy(dX, X, n * p * sizeof(float),
            cudaMemcpyHostToDevice));
68    CUDA_CALL(cudaMemcpy(dY, Y, n * sizeof(float), cudaMemcpyHostToDevice)
            );
69
70    cublasSgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N, p, p, n,
71      a, dX, n, dX, n, b, dXtX, p);
72
73    CUDA_CALL(cudaMemcpy(XtX, dXtX, p * p * sizeof(float),
            cudaMemcpyDeviceToHost));
```

Example: ols.cu

```
74    printf("XtX\n");
75    for(i = 0; i < p; i++){
76      for(j = 0; j < p; j++){
77        printf("%0.2f\t", XtX[I(i, j, p)]);
78      }
79      printf("\n");
80    }
81    printf("\n");
```

Output of code so far

```
> nvcc -I /usr/local/cula/include -L /usr/local/cula/lib64 -lcula_core -
    lcula_lapack -lcublas -lcudart ols.cu -o ols
> ./ols
Y X
3.37 1.00 0.00 0.00
1.94 1.00 0.10 0.10
0.44 1.00 0.20 0.40
-0.30 1.00 0.30 0.90
-2.08 1.00 0.40 1.60
-0.84 1.00 0.50 2.50
-0.18 1.00 0.60 3.60
3.40 1.00 0.70 4.90
5.51 1.00 0.80 6.40
7.39 1.00 0.90 8.10

XtX
10.00 4.50 28.50
4.50 2.85 20.25
28.50 20.25 153.33
```

Example: `ols.cu`

- We have $X^T X$, but which we need to invert in order to compute our solution,

$$\widehat{\beta} = (X^T X)^{-1} X^T y$$

- But CUBLAS can only invert triangular matrices!

Enter CULA: CUDA LAPACK

```
82    culaInitialize();
83
84    culaDeviceSgetrf(p, p, dXtX, p, ipiv);
85    culaDeviceSgetri(p, dXtX, p, ipiv);
86
87    CUDA_CALL(cudaMemcpy(XtX, dXtX, p * p * sizeof(float),
          cudaMemcpyDeviceToHost));
88
89    printf("XtX^(-1)\n");
90    for(i = 0; i < p; i++){
91      for(j = 0; j < p; j++){
92        printf("%0.2f\t", XtX[I(i, j, p)]);
93      }
94      printf("\n");
95    }
96    printf("\n");
97
98    cublasSgemm(handle, CUBLAS_OP_T, CUBLAS_OP_N, p, 1, n,
99      a, dX, n, dY, n, b, dXtY, p);
100
101   cublasSgemv(handle, CUBLAS_OP_N, p, p,
102     a, dXtX, p, dXtY, 1, b, dbeta, 1);
103
104   CUDA_CALL(cudaMemcpy(beta, dbeta, p * sizeof(float),
          cudaMemcpyDeviceToHost));
105
106   printf("CUBLAS/CULA matrix algebra parameter estimates:\n");
107   for(i = 0; i < p; i++){
108     printf("beta_%i = %0.2f\n", i, beta[i]);
109   }
110   printf("\n");
```

CULA's culaSgels() does least squares for you

```
111     culaSgels('N', n, p, 1, X, n, Y, n);
112
113     printf("culaSgels Parameter estimates:\n");
114     for(i = 0; i < p; i++){
115       printf("beta_%i = %0.2f\n", i, Y[i]);
116     }
117     printf("\n");
118
119     culaShutdown();
120     cublasDestroy(handle);
121
122     free(a);
123     free(b);
124     free(X);
125     free(XtX);
126     free(XtY);
127     free(beta);
128     free(Y);
129
130     CUDA_CALL(cudaFree(dX));
131     CUDA_CALL(cudaFree(dXtX));
132     CUDA_CALL(cudaFree(dXtY));
133     CUDA_CALL(cudaFree(dbeta));
134     CUDA_CALL(cudaFree(dY));
135 }
```

Rest of the output

```
19 XtX^(-1)
20 0.62  -2.59  0.23
21 -2.59  16.55  -1.70
22 0.23  -1.70  0.19
23
24 CUBLAS/CULA matrix algebra parameter estimates:
25 beta_0 = 3.78
26 beta_1 = -25.53
27 beta_2 = 3.36
28
29 culaSgels Parameter estimates:
30 beta_0 = 3.78
31 beta_1 = -25.53
32 beta_2 = 3.36
```

Resources

1. J. Sanders and E. Kandrot. CUDA by Example. Addison-Wesley, 2010.
2. D. Kirk, W.H. Wen-mei, and W. Hwu. *Programming massively parallel processors: a hands-on approach.* Morgan Kaufmann, 2010.
3. A. Lee, C. Yau, M.B. Giles, A. Doucet, and C.C. Holmes. On the utility of graphics cards to perform massively parallel simulation of advanced monte carlo methods. *Journal of Computational and Graphical Statistics*, 19(4): 769-789, 2010.
4. M.A. Suchard, Q. Wang, C. Chan, J. Frelinger, A. Cron, and M. West. Understanding gpu programming for statistical computation: Studies in massively parallel mixtures. *Journal of Computational and Graphical Statistics.* 19(2): 419-438, 2010.

Outline

GPUs, parallelism, and why we care

CUDA and our CUDA systems

GPU computing with R

How GPU parallelism works

Examples

- Vector addition
- Pairwise sum
- K-means clustering
- Markov chain Monte Carlo
- Ordinary least squares

Resources

5. http://docs.nvidia.com/cuda/index.html
6. https://developer.nvidia.com/gpu-accelerated-libraries
7. Dr. Ranjan Maitra's STAT 580 notes.
8. Dr. Jarad Niemi's STAT 544 notes.
9. Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis*. 2nd ed. Chapman & Hall/CRC, 2004.
10. George Marsaglia and Wai Wan Tsang. "A Simple Method for Generating Gamma Variables." *ACM Transactions on Mathematical Software* 26(3), Sept 2000. 363-372.

Example code

1. vectorsums.cu
2. pairwise_sum.cu
3. kmeans.zip
4. mcmc.zip
5. ols.cu

That's all.

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