A codelsss introduction to GPU parallelism

Will Landau

Iowa State University

September 23, 2013

Outline

A review of GPU parallelism

Examples of parallelism
- Vector addition
- Pairwise summation
- Matrix multiplication
- K-means clustering
- Markov chain Monte Carlo

Outline

The single instruction, multiple data (SIMD) paradigm

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

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

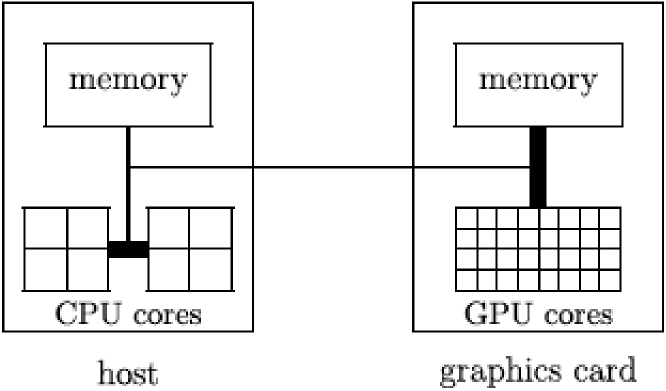
- On CPUs, the iterations of the loop run sequentially.
- With GPUs, we can easily run all 1,000,000 iterations simultaneously.

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

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

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.



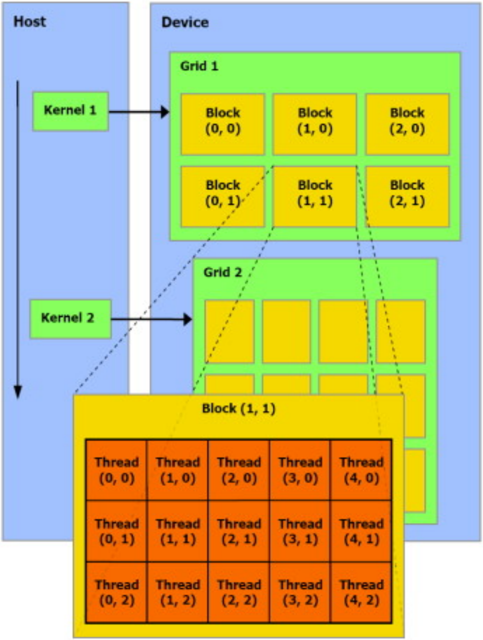
How GPU parallelism works

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: "Hey, GPU. Sum pairs of adjacent numbers. Use the array, (1, 2, 3, 4, 5, 6, 7, 8)."
  - 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	$1 + 2$	$3 + 4$	$5 + 6$	$7 + 8$

- I could have also used 1 block with 4 threads and given the threads different pairs of numbers.


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)

Outline

Vector addition

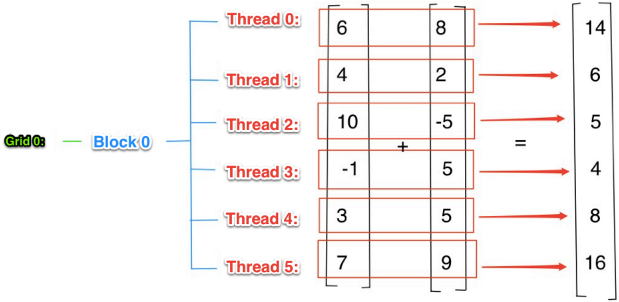
- Say I have 2 vectors,

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

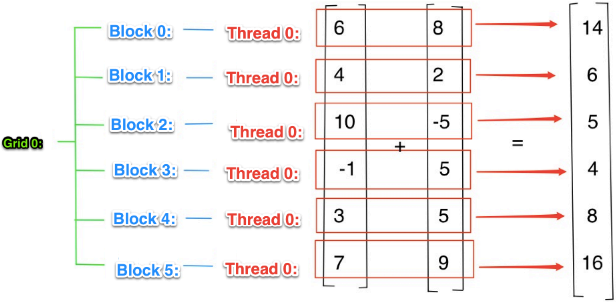
- I want to compute their component-wise sum,

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

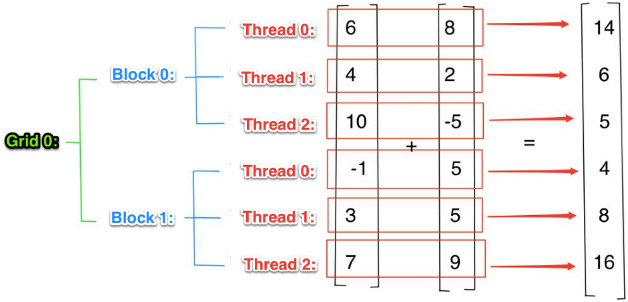
Vector addition



Vector addition



Vector addition



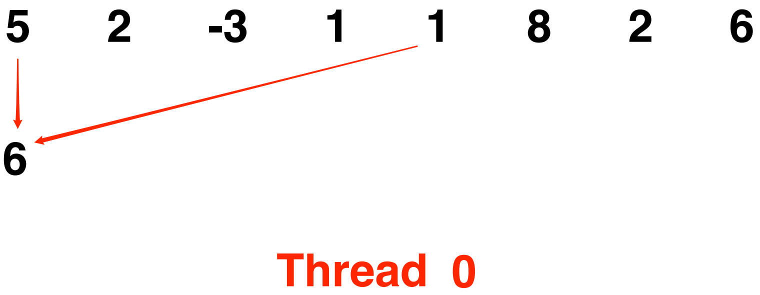
Pairwise summation

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

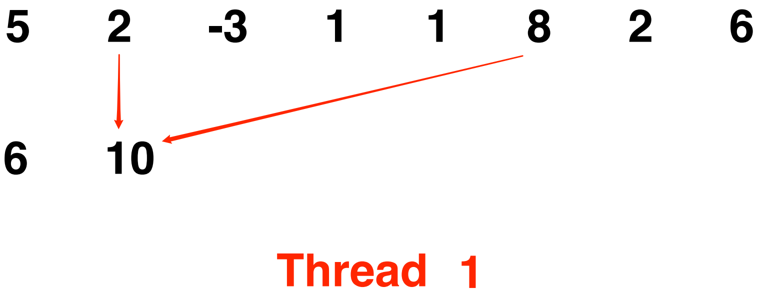
$$(5, 2, -3, 1, 1, 8, 2, 6)$$

using 1 block of 4 threads.

Pairwise summation



Pairwise summation



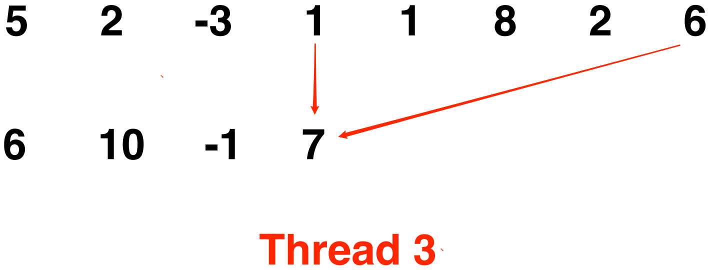
Pairwise summation

5 2 -3 1 1 8 2 6

6 10 -1

Thread 2

Pairwise summation



Pairwise summation

5 2 -3 1 1 8 2 6

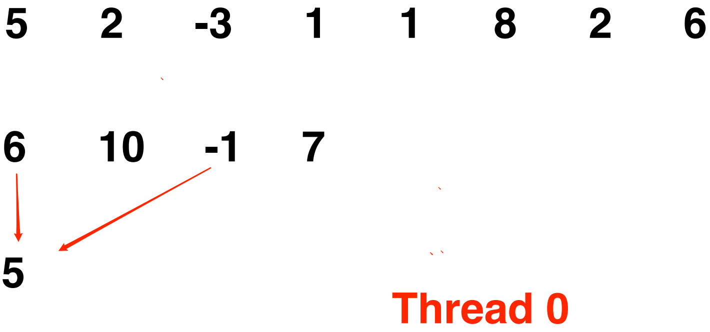
6 10 -1 7

Synchronize threads

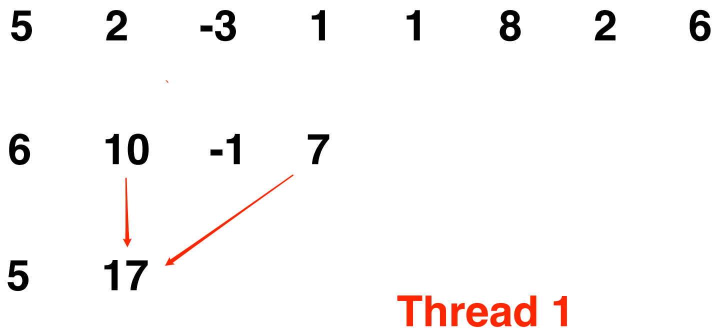
Synchronizing threads

- **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.

Pairwise summation



Pairwise summation



Pairwise summation

5 2 -3 1 1 8 2 6

6 10 -1 7

5 17

Synchronize Threads

Pairwise summation

5 2 -3 1 1 8 2 6

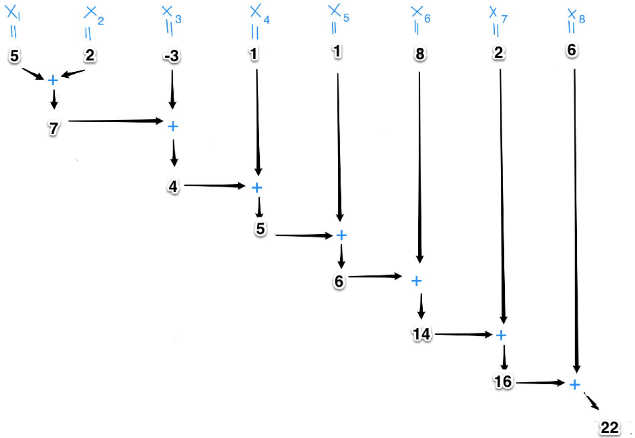
6 10 -1 7

5 17

22

Thread 0

Compare the pairwise sum to the sequential sum



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

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.

Matrix multiplication

- Take an $m \times n$ matrix, $A = (a_{ij})$, and an $n \times p$ matrix, $B = (b_{jk})$. Compute $C = A \cdot B$:
  - Write A in terms of its rows: $A = \begin{bmatrix} a_{1.} \\ \vdots \\ a_{m.} \end{bmatrix}$ where $a_{i.} = \begin{bmatrix} a_{i1} & \cdots & a_{in} \end{bmatrix}$.
  - Write B in terms of its columns: $B = \begin{bmatrix} b_{.1} & \cdots & b_{.p} \end{bmatrix}$ where $b_{.k} = \begin{bmatrix} b_{1k} \\ \vdots \\ b_{nk} \end{bmatrix}$
  - Compute $C = A \cdot B$ by taking the product of each row of A with each column of B:

$$C = A \cdot B = \begin{bmatrix} (a_{1.} \cdot b_{.1}) & \cdots & (a_{1.} \cdot b_{.p}) \\ \vdots & \ddots & \vdots \\ (a_{m.} \cdot b_{.1}) & \cdots & (a_{m.} \cdot b_{.p}) \end{bmatrix}$$

Parallelizing matrix multiplication

- Entry (i, k) of matrix C is

$$\begin{aligned} c_{ik} &= \underbrace{a_{i1}b_{1k}} + \underbrace{a_{i2}b_{2k}} + \cdots + \underbrace{a_{in}b_{nk}} \\ &= \quad c_{i1k} \quad + \quad c_{i2k} \quad + \cdots + \quad c_{ink} \end{aligned}$$

- Assign block (i, k) to compute c_{ik}.
  1. Spawn n threads.
  2. Tell the j'th thread to compute $c_{ijk} = a_{ij} \cdot b_{jk}$.
  3. Synchronize threads to make sure we have finished calculating $c_{i1k}, c_{i2k}, \ldots, c_{ink}$ before continuing.
  4. Compute $c_{ik} = \sum_{j=1}^{n} c_{ijk}$ as a pairwise sum.

Matrix multiplication

- Say I want to compute $A \cdot B$, where:

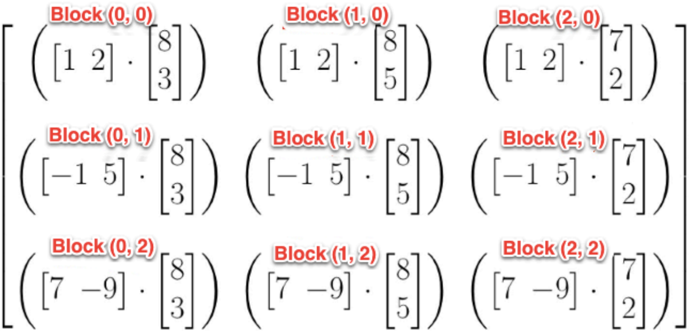
$$A = \begin{bmatrix} 1 & 2 \\ -1 & 5 \\ 7 & -9 \end{bmatrix} \quad B = \begin{bmatrix} 8 & 8 & 7 \\ 3 & 5 & 2 \end{bmatrix}$$

- I write the multiplication as an array of products:

$$C = \begin{bmatrix} \left(\begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix} \right) & \left(\begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 5 \end{bmatrix} \right) & \left(\begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 7 \\ 2 \end{bmatrix} \right) \\ \left(\begin{bmatrix} -1 & 5 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix} \right) & \left(\begin{bmatrix} -1 & 5 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 5 \end{bmatrix} \right) & \left(\begin{bmatrix} -1 & 5 \end{bmatrix} \cdot \begin{bmatrix} 7 \\ 2 \end{bmatrix} \right) \\ \left(\begin{bmatrix} 7 & -9 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix} \right) & \left(\begin{bmatrix} 7 & -9 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 5 \end{bmatrix} \right) & \left(\begin{bmatrix} 7 & -9 \end{bmatrix} \cdot \begin{bmatrix} 7 \\ 2 \end{bmatrix} \right) \end{bmatrix}$$

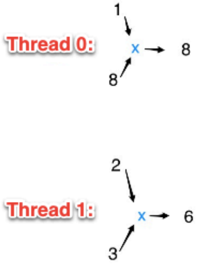
Matrix multiplication

- We don't need to synchronize blocks because they operate independently.

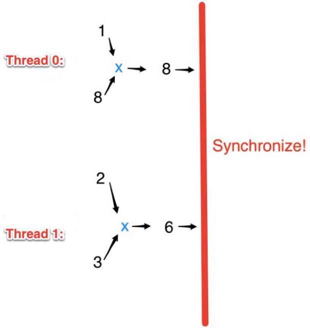


Matrix multiplication

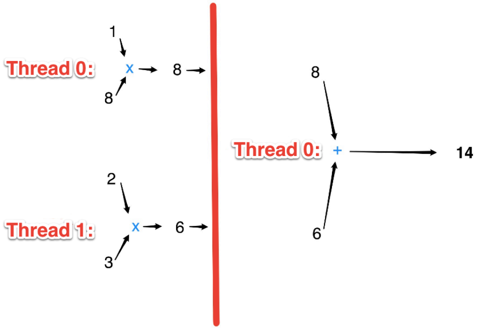
- Consider block (0, 0), which computes $\begin{bmatrix} 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 3 \end{bmatrix}$



Matrix multiplication

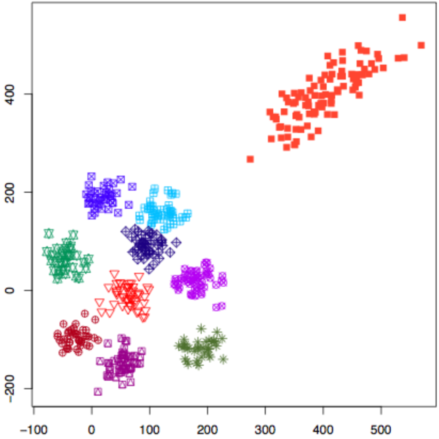


Matrix multiplication

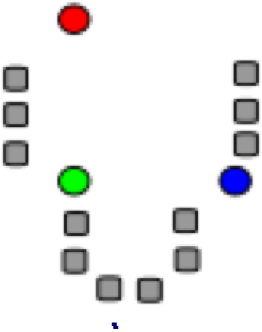


Lloyd's K-means algorithm

- Cluster N vectors in Euclidian space into K groups.

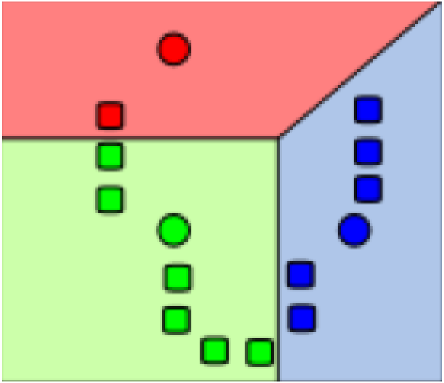


Step 1: choose initial cluster centers.

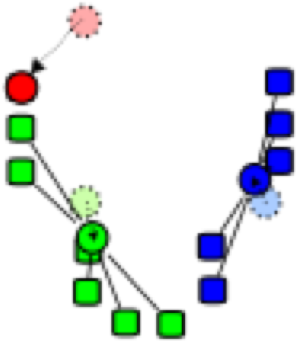


- The circles are the cluster means, the squares are the data points, and the color indicates the cluster.

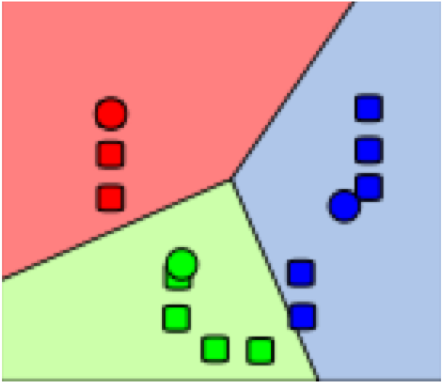
Step 2: assign each data point (square) to its closest center (circle).



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.
  - 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.

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:

$$
\begin{aligned}
y_k &\overset{\text{ind}}{\sim} \text{Poisson}(n_k \cdot \theta_k) \\
\theta_k &\overset{\text{iid}}{\sim} \text{Gamma}(\alpha,\ \beta) \\
\alpha &\sim \text{Uniform}(0, a_0) \\
\beta &\sim \text{Uniform}(0, b_0)
\end{aligned}
$$

  - Also assume α and 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.

Preview: a bare bones CUDA C workflow

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void some_kernel(...){...}

int main (void){
  // Declare all variables.
  ...
  // Allocate host memory.
  ...
  // Dynamically allocate device memory for GPU
     results.
  ...
  // Write to host memory.
  ...
  // Copy host memory to device memory.
  ...
```

Preview: a bare bones CUDA C workflow

```
  // Execute kernel on the device.
  some_kernel<<< num_blocks, num_theads_per_block
      >>>(...);

  // Write GPU results in device memory back to
     host memory.
  ...
  // Free dynamically-allocated host memory
  ...
  // Free dynamically-allocated device memory
  ...
}
```

Outline

Resources

1. J. Sanders and E. Kandrot. *CUDA by Example.* Addison-Wesley, 2010.
2. Prof. Jarad Niemi's STAT 544 lecture notes.

That's all for today.

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