

Extreme Machine Learning with GPUs

John Canny

Computer Science Division
University of California, Berkeley
GTC, March, 2014

Big Data

Event and text data:

Microsoft

Yahoo

Ebay

Quantcast

...

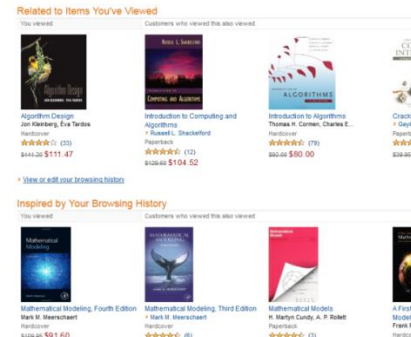
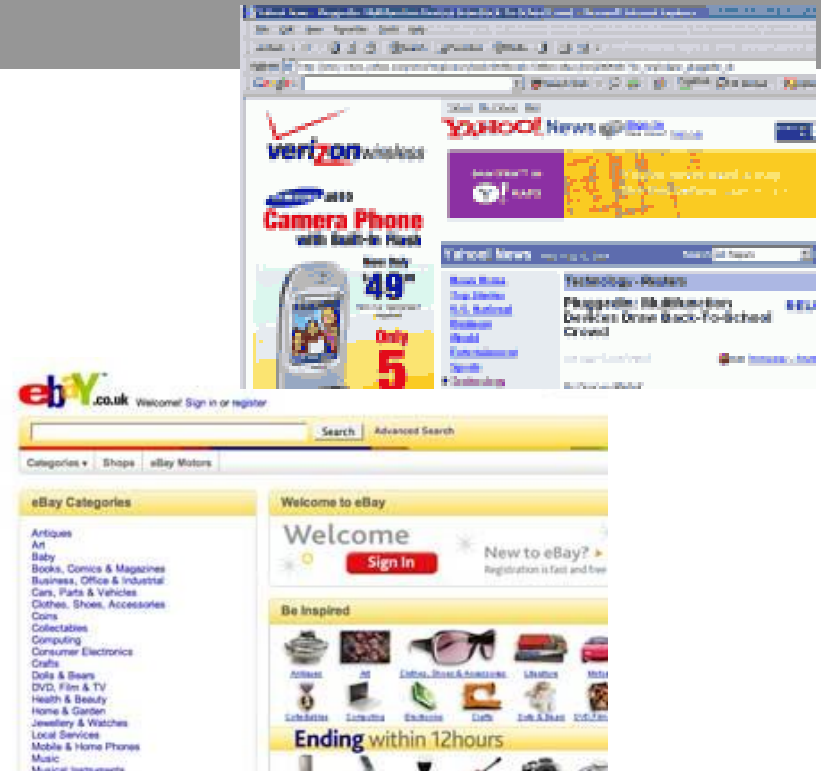
MOOC logs

Social Media

Health Data

...

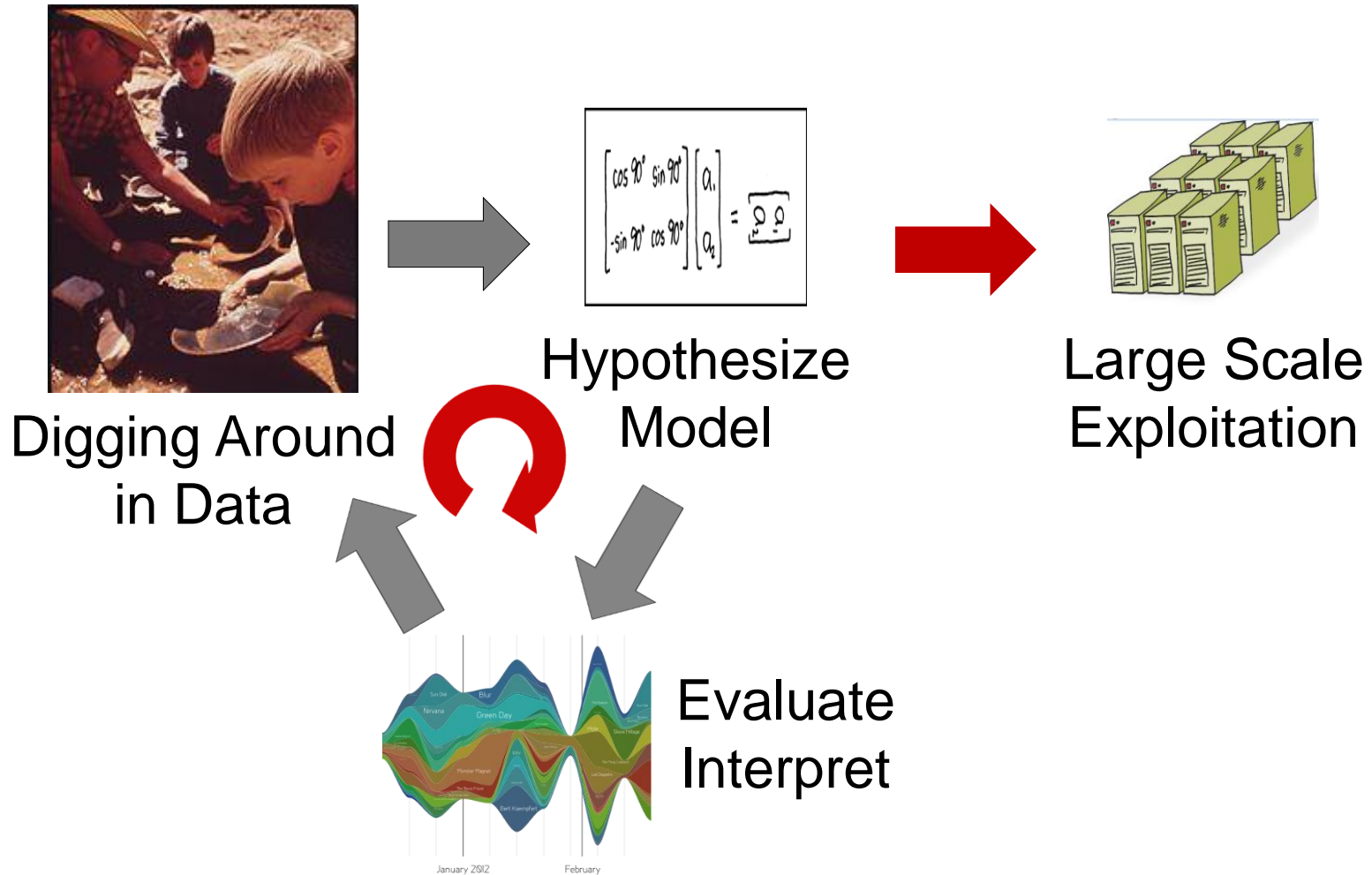
Later: Images, Video














Recommendation System

Sentiment Analysis and
Social Network Analysis

Big Data Workflow

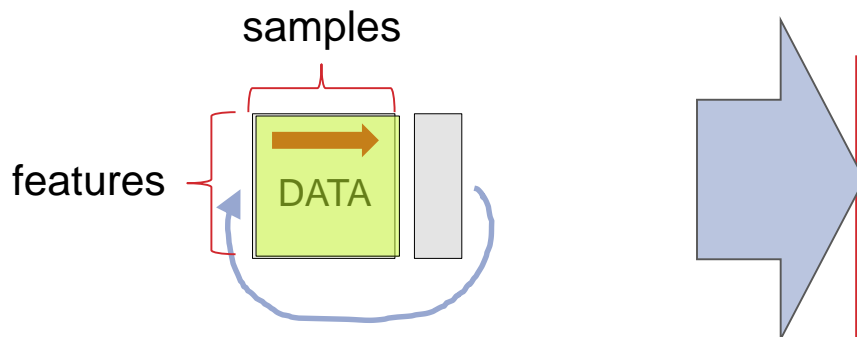


Top-10 “Big Data” Algorithms

1. Regression (logistic, linear) + Naïve Bayes 
2. Support Vector Machines 
3. Greedy Clustering (k-Means) 
4. Topic Models (Latent Dirichlet Allocation) 
5. Collaborative Filtering (Sparse Matrix Factorization) 
6. Random Forests 
7. Hidden-Markov Models 
8. Spectral Clustering 
9. Factorization Machines (Regression with Interactions) 
10. Multi-layer neural networks 
11. Natural Language Parsing 

Machine Learning for Big Data

Classical: Batch model update in memory

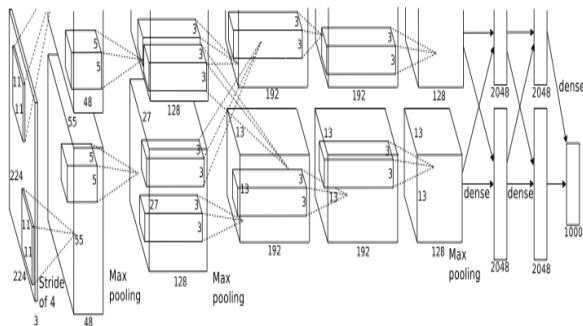


Spark: UC Berkeley

HaLoop: U. Washington

Mahout

**Deep
Learning**



Torch7: (NYU, NEC)

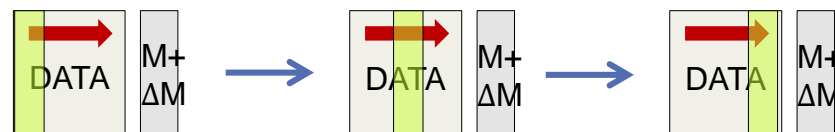
Convnet, RNNLib, Visual-RBM: Toronto

Theano: Montreal

- **Incremental-update Methods**

- Stochastic Gradient Descent (SGD)
- Gibbs Sampling (GS)

Large Datasets: Mini-batch model updates



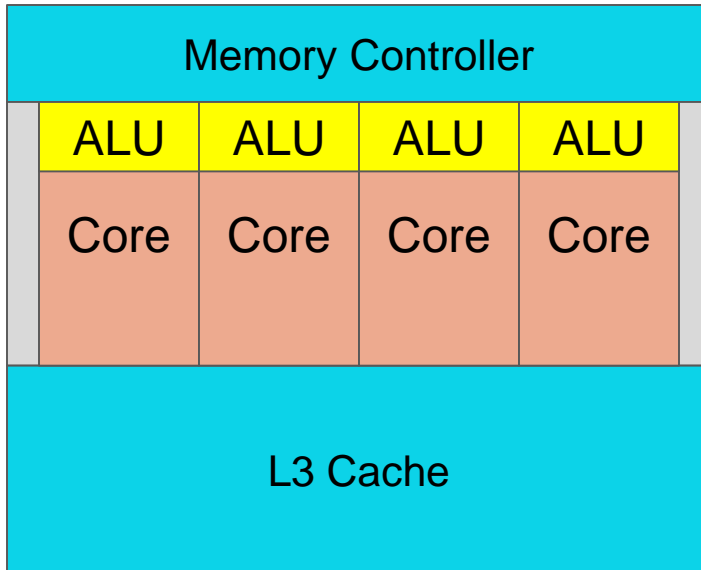
BIDMat/BIDMach: (this talk)

Downpour SGD: (Google)

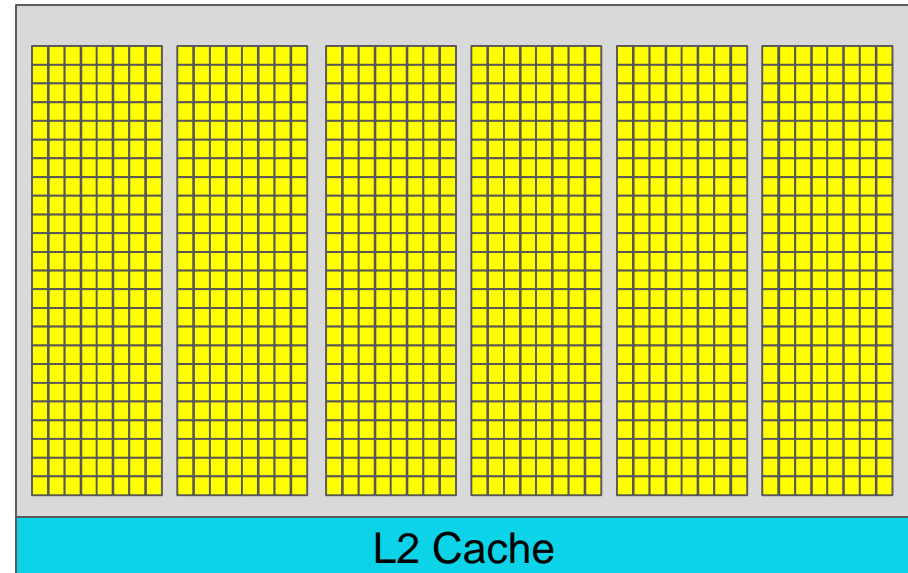
Hogwild: U. Wisc.-Madison

GPUs at a glance...

Intel® CPU

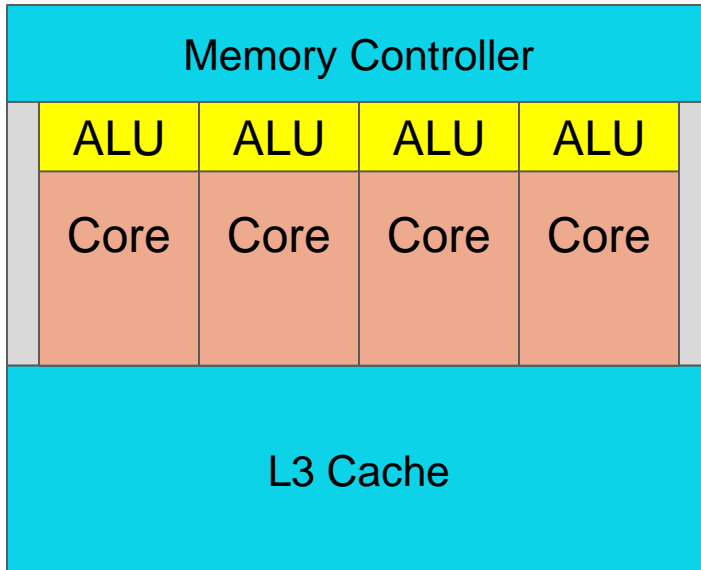


NVIDIA® GPU



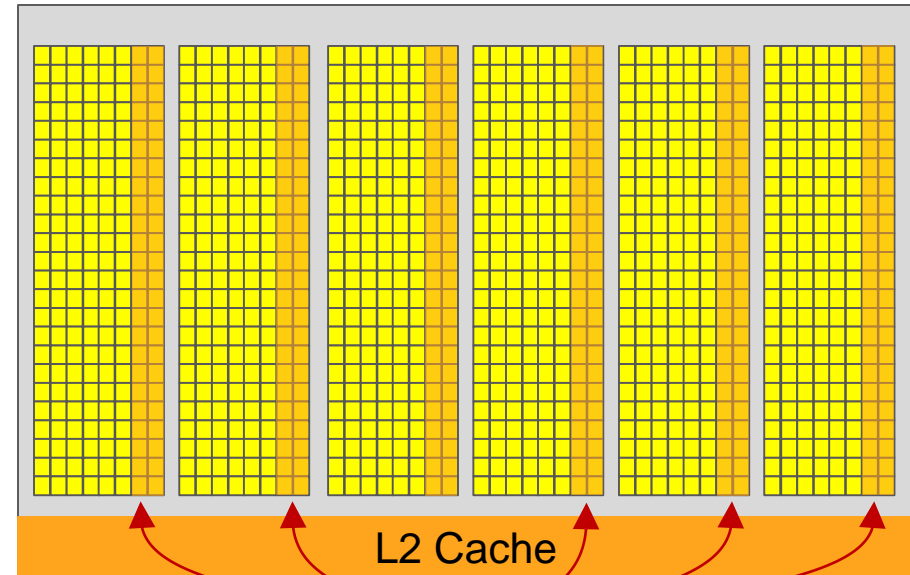
Vive La Difference !

Intel® CPU



4kB registers: 

NVIDIA® GPU



Hardware transcendentals (power series)

 4 MB register file (!)

A datapoint: NLP Parsing (Canny, Hall, Klein, EMNLP 2013)

Natural language parsing with the state-of-the-art Berkeley grammar (1100 symbols, 1.7 million rules)

End-to-End Throughput (4 GPUs):

2-2.4 Teraflops (1-1.2 B rules/sec)

CPU throughput is about **5 Mflops**.

i.e. we achieved a **0.5 million-fold speedup** on rule evaluation.

Memory Performance

Intel® 8 core Sandy Bridge CPU

4kB registers: 5 TB/s

512K L1 Cache 1 TB/s

2 MB L2 Cache

8 MB L3 Cache 500 GB/s

10s GB Main Memory 20 GB/s

NVIDIA® GK110 GPU

4 MB register file (!) 40 TB/s

1 MB Constant Mem 13 TB/s

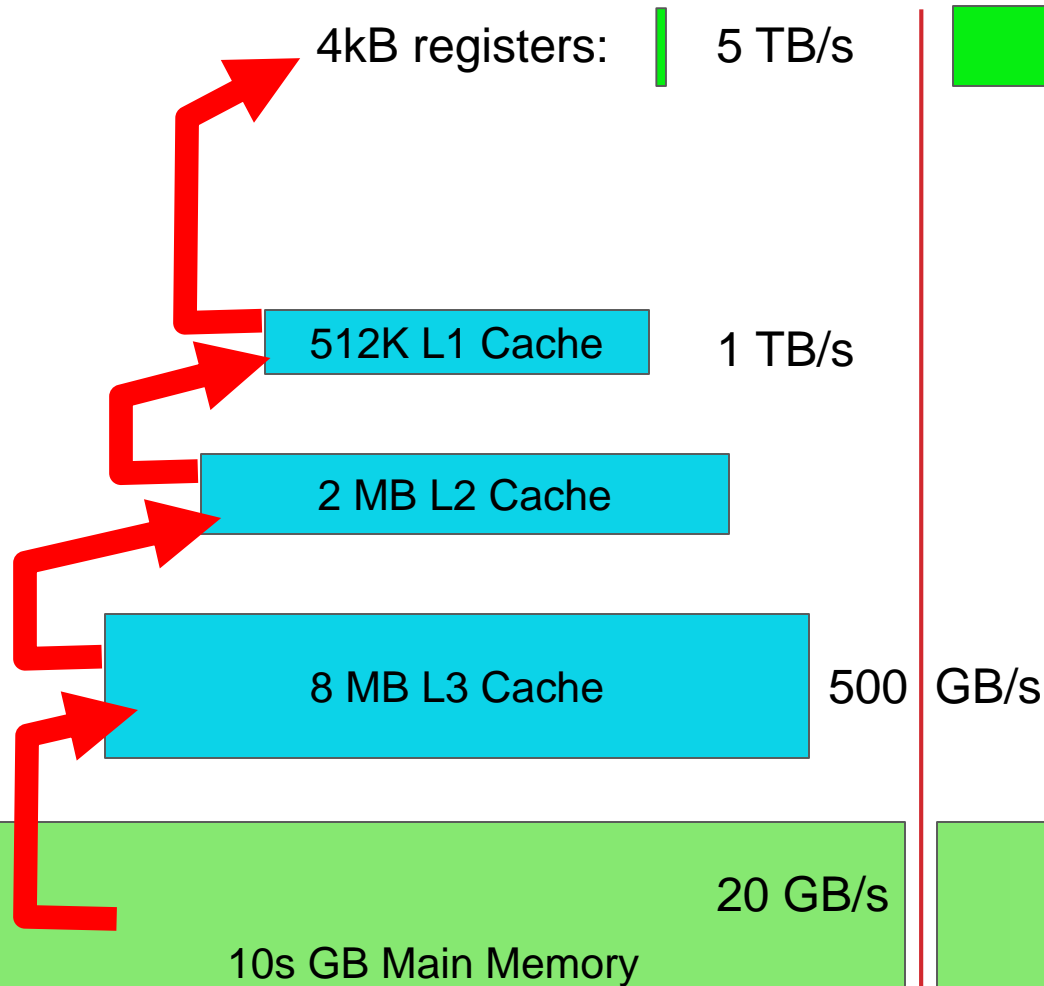
1 MB Shared Mem 1 TB/s

1.5 MB L2 Cache 500 GB/s

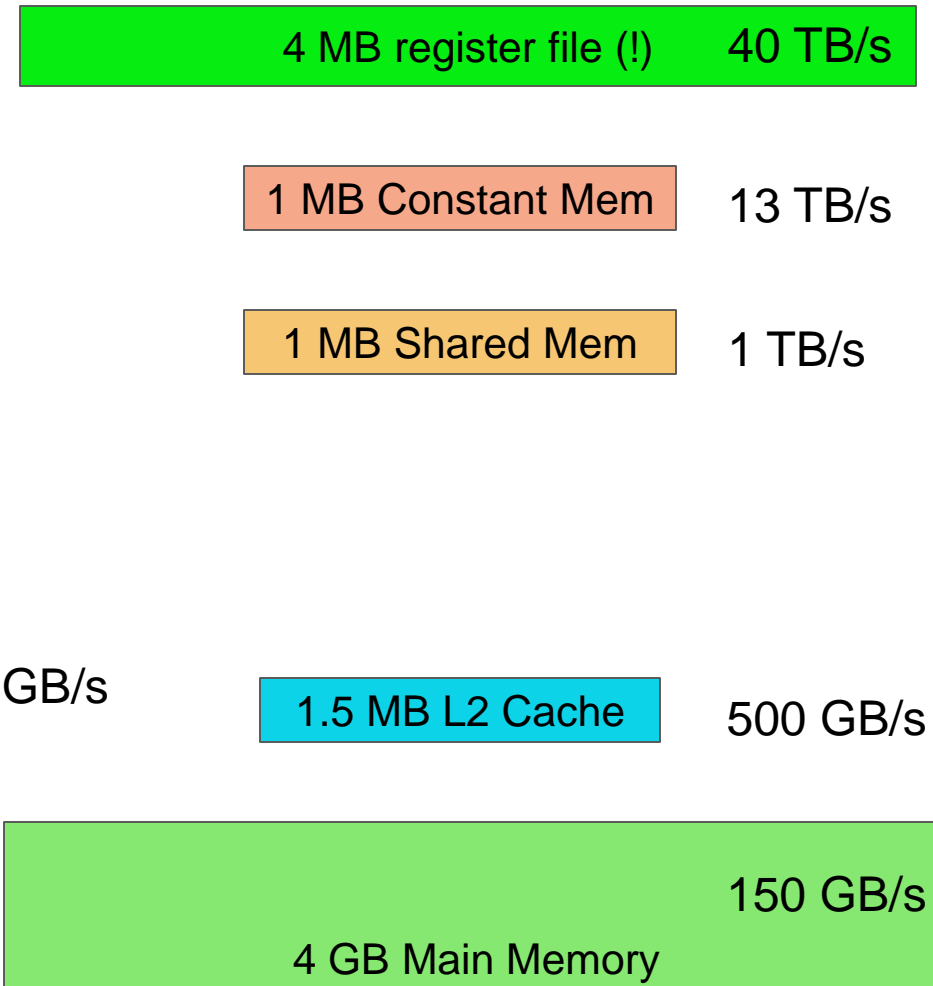
4 GB Main Memory 150 GB/s

Hi Speed CPU kernels

Intel® 8 core Sandy Bridge CPU



NVIDIA® GK110 GPU



A Strategy for Speed on GPUs

Intel® 8 core Sandy Bridge CPU

4kB registers: 5 TB/s

512K L1 Cache 1 TB/s

2 MB L2 Cache

500 GB/s

8 MB L3 Cache

10s GB Main Memory

20 GB/s

NVIDIA® GK110 GPU

4 MB register file (!) 40 TB/s

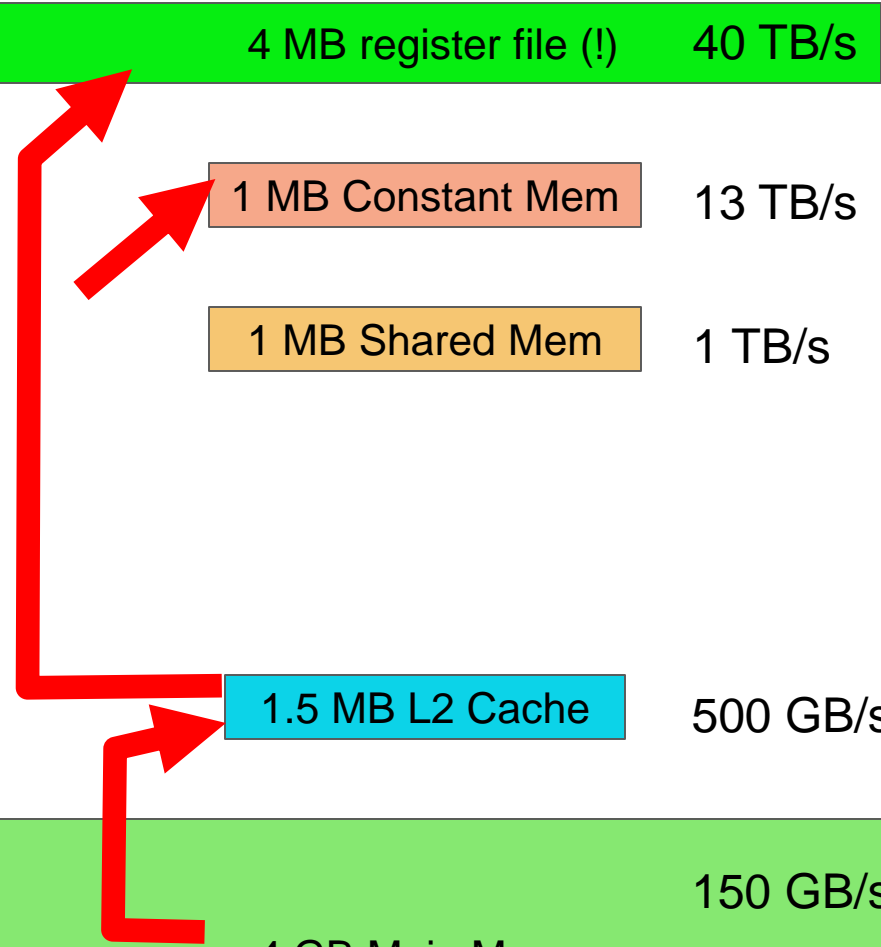
1 MB Constant Mem 13 TB/s

1 MB Shared Mem 1 TB/s

1.5 MB L2 Cache 500 GB/s

4 GB Main Memory

150 GB/s



Using Register and Constant Memory

Our goal is to *use registers to hold symbols values*, and *constant memory to hold rule weights*.

i.e. we commit to compiling the grammar into code, like this (actual GPU code):

```
float L001 = left[1][tid];  
float R031 = right[31][tid];  
float P001 = L001 * R031 * 1.338202e-001f;  
P001 += L021 * R019 * 8.32642e-003f;  
...  
atomicAdd(&parent[1][tid], P001);
```

Using Register and Constant Memory

But: Each GPU “core” has only 63 (or 255 in Titan) registers.

We have $1132 \times 3 = 3396$ symbols, a less-than-perfect fit.

Therefore we use blocking, similar to the approach used in fast CPU matrix kernels, partly inspired by:

“Usually not worth trying to cache block like you would on CPU”
– GTC 2012 Performance Analysis and Optimization ☺

i.e. we cluster the symbols into small subsets which fit into register storage, trying at the same time to balance the number of rules in each block.

A Strategy for Speed on GPUs

Intel® 8 core Sandy Bridge CPU

4kB registers: 5 TB/s

512K L1 Cache 1 TB/s

2 MB L2 Cache

500 GB/s

8 MB L3 Cache

10s GB Main Memory

20 GB/s

NVIDIA® GK110 GPU

4 MB register file (!) 40 TB/s

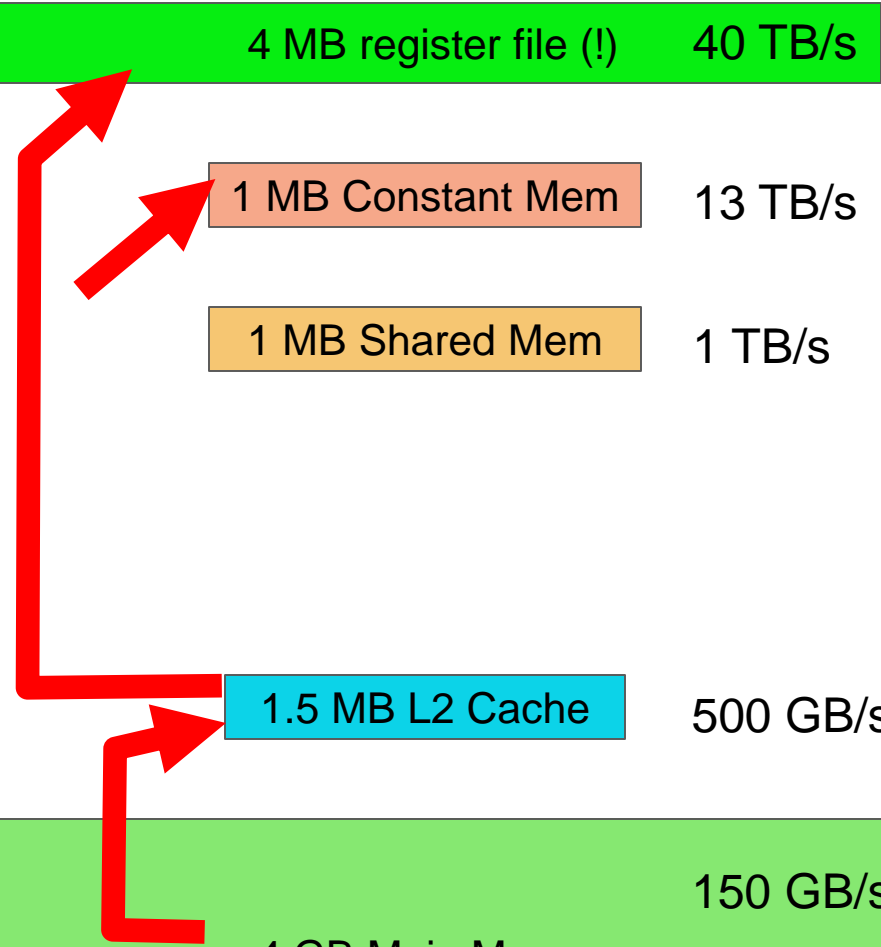
1 MB Constant Mem 13 TB/s

1 MB Shared Mem 1 TB/s

1.5 MB L2 Cache 500 GB/s

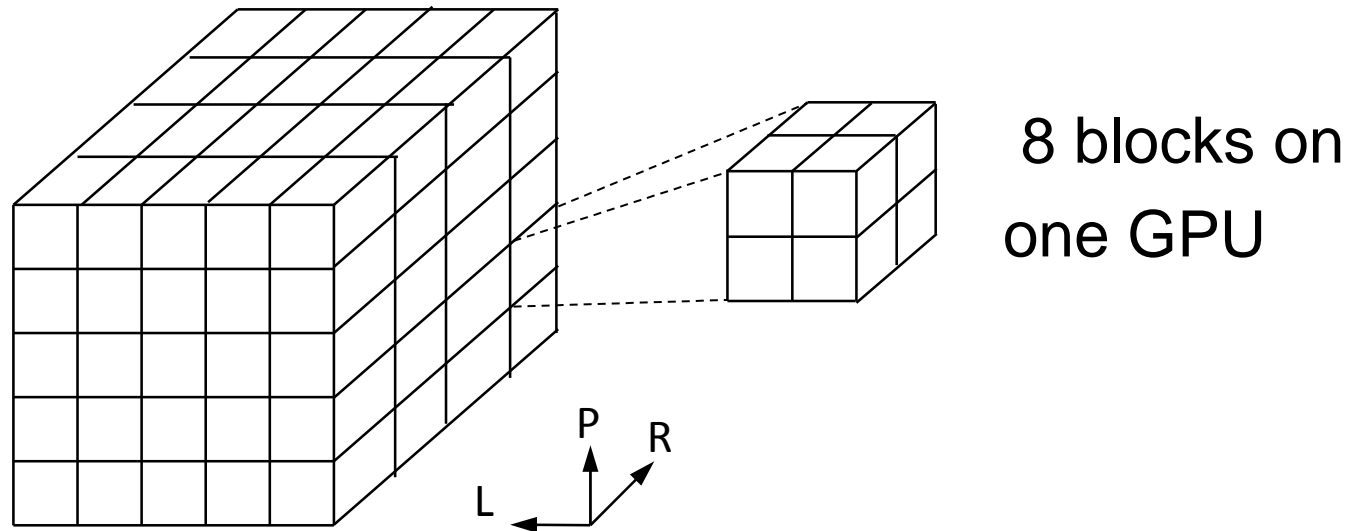
4 GB Main Memory

150 GB/s



Blocking

Align the (1132) symbols for P, L, R along the axes of a cube. We want small subcubes whose sides are roughly 50 values, that will fit in GPU register memory.



Blocks that run as separate kernels (function calls) on a GPU.

Serendipity

The compiler's version











```
float tmp = L021 * R019;  
P001 += tmp * 8.32642e-003f;  
P002 += tmp * 4.31572e-005f;  
P005 += tmp * 2.81231e-002f;
```

Compiles each rule update line into a ***single atomic multiply-add instruction***, which runs in one cycle.

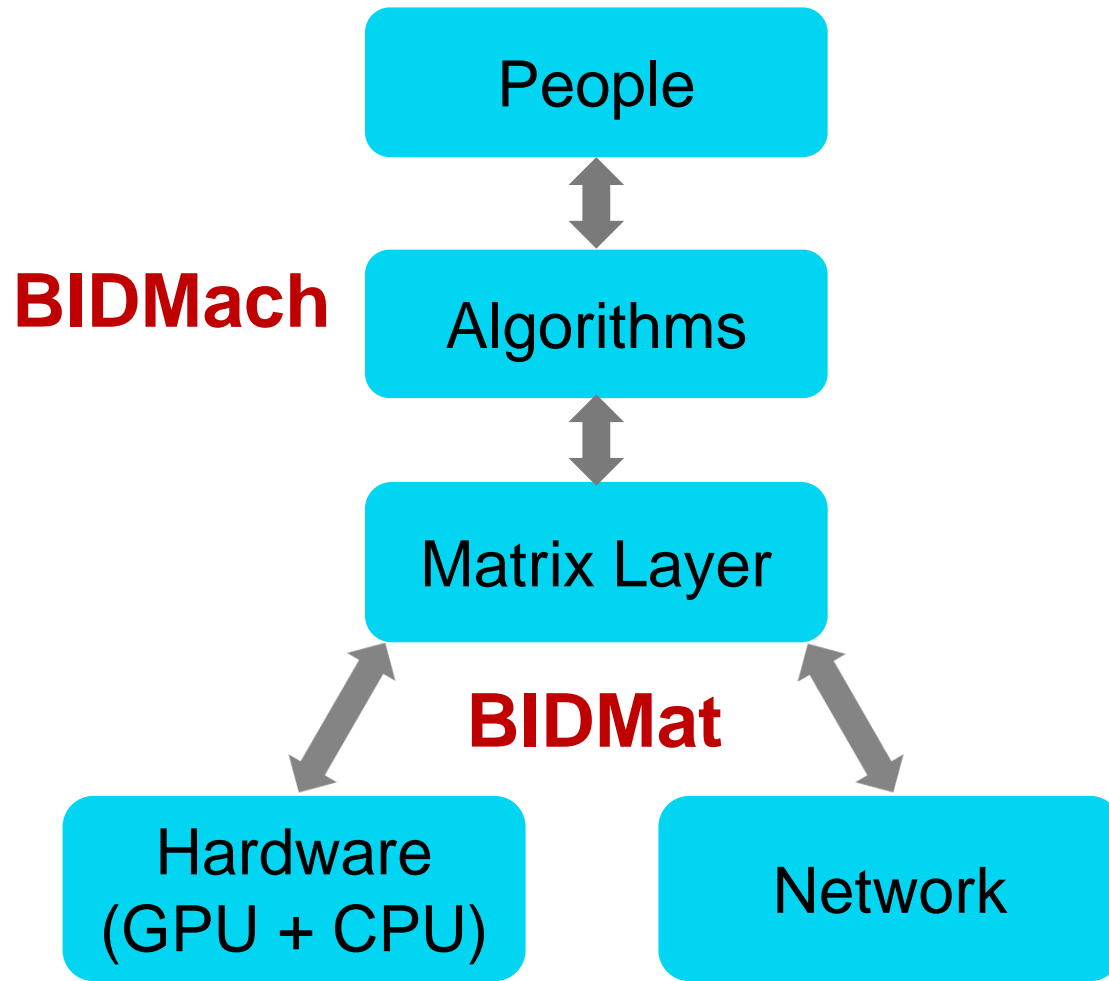
i.e. with 1.7 million rules, the compiled GPU code has about 1.7 million instructions.

It runs at about 2 cycles/rule or **1 teraflop per GPU**. This is as fast as dense matrix multiply on the GTX-680.

Back to our Top-10 list

1. Regression (logistic, linear) + Naïve Bayes 
2. Support Vector Machines 
3. Greedy Clustering (k-Means) 
4. Topic Models (Latent Dirichlet Allocation) 
5. Collaborative Filtering (Sparse Matrix Factorization) 
6. Random Forests 
7. Hidden-Markov Models 
8. Spectral Clustering 
9. Factorization Machines (Regression with Interactions) 
10. Multi-layer neural networks 

BIDMat/BIDMach architecture



A GPU-enabled Matrix Tool

BIDMAT

Written in the beautiful Scala language:

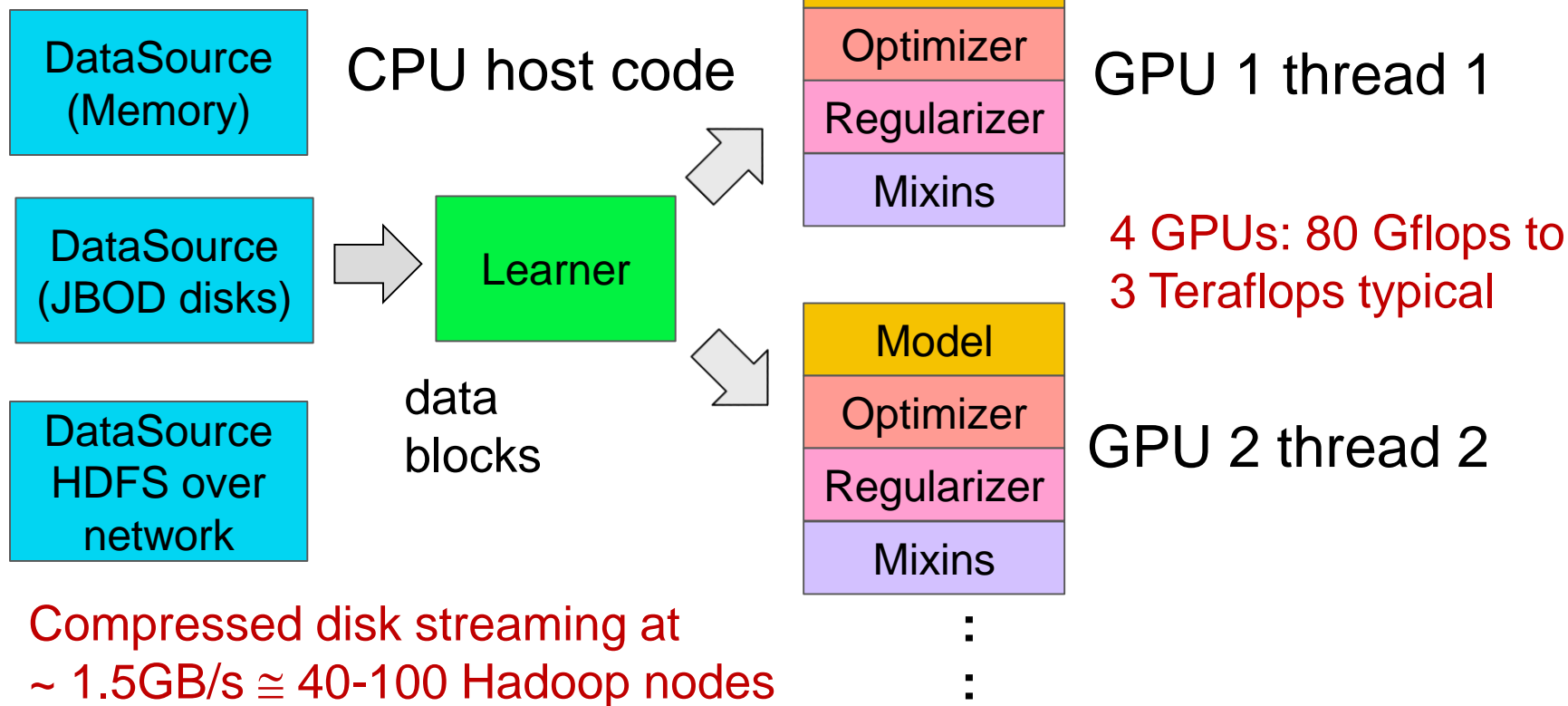
- Interpreter, w/ excellent performance
- Natural syntax $+$, $-$, $*$, \circ , \bullet , \otimes etc and high-level expressivity
- CPU and GPU backend (generics)
- Hardware acceleration – **many custom GPU kernels**
- Easy threading (Actors)
- Java VM + Java codebase – runs on Hadoop, Spark
- Good text processing, integrated XML interpreter

Inspired by Matlab, R, SciPy

A modular learning API

Zhao+Canny
SIAM DM 13, KDD 13, BIGLearn 13

BIDMACH



BIDMach sample code

Latent Dirichlet Allocation Model:

```
def eStep(sdata:Mat, user:Mat):Unit = {  
  for (i <- 0 until opts.uiters) {  
    val preds = SDDMM(modelmat, user, sdata)  
    val unew = user ° (mm * (sdata / preds)) + opts.alpha  
    user <-- exppsi(unew)  
  }  
}
```

BIDMach

Every Learner can:

- Run Sparse or Dense input matrices
- Run on GPU or CPU
- Run on single or multiple GPUs
- Use in-memory or disk data sources (matrix caching)
- Run on single or multiple network nodes*

BIDMach Performance

Performance dominated by a few kernels:

Dense-dense MM – sgemm (for dense input data)

Sparse-dense MM and filtered MM (for sparse inputs)

Almost all learners achieve end-to-end performance of:

- 20-40 Gflops (for sparse input data)
- 1-3 Tflops (for dense input data)

Tested K-means, LDA, ALS, on Mahout, Scikit-Learn, Vowpal Wabbit, Mlbase, with MKL acceleration if possible.

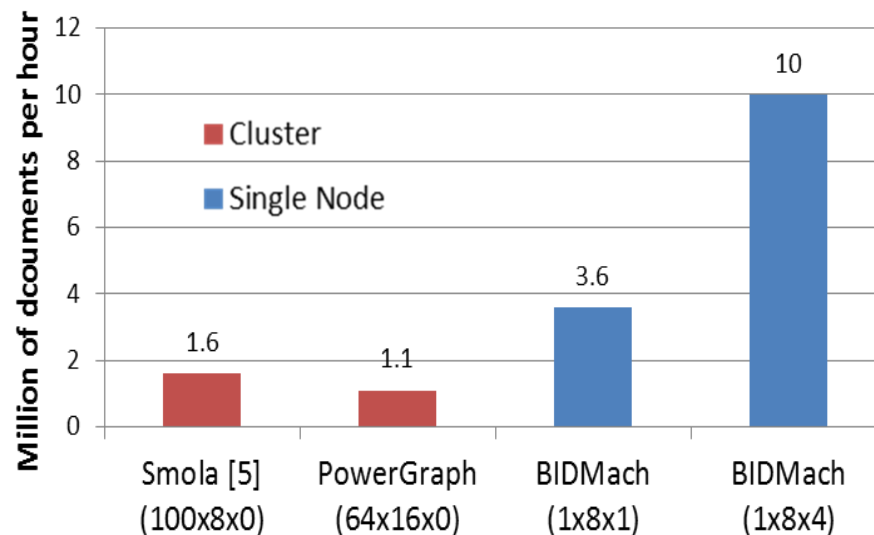
Speedups 100x to several 1000x.

Benchmarks

Variational Latent Dirichlet Allocation

(N hosts x N cores x N GPUs)

LDA M docs/hour



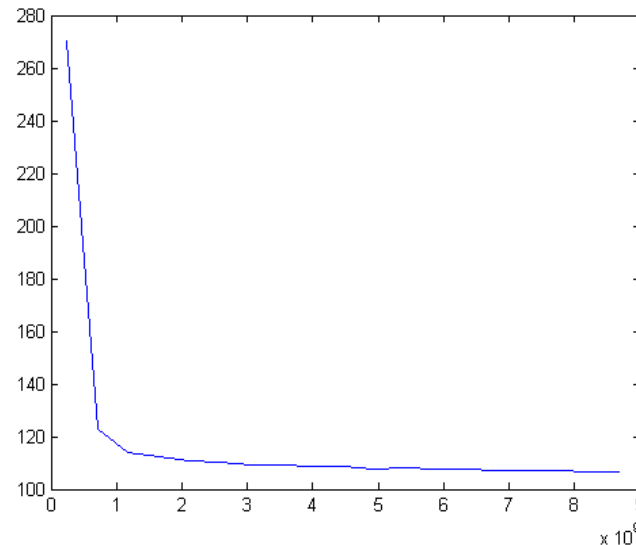
i.e. 10x improvement for the single-node implementation vs. 64-node cluster, or 500x in per-node throughput.

Avg end-to-end throughput with 4 GPUs is 80 Gflops.

Benchmarks

Variational Latent Dirichlet Allocation (256 dims)

LDA convergence on **1 Terabyte** of Twitter data



We have run this algorithm up to 10 TB, $\sim 10^{16}$ floating point operations, on a single PC with GTX-680s.

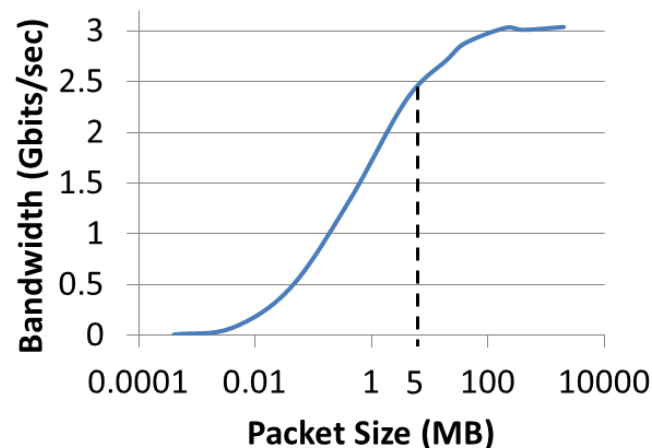
This is the largest calculation on commodity hardware that we know of.

MapReduce Version

Variational Latent Dirichlet Allocation (256 dims)

But you can do this on a big MapReduce Cluster, right?

- No-one has
- Probably not
- The common MapReduce implementations (Hadoop, Spark, Powergraph*) don't scale. i.e. The communication time stops decreasing and starts increasing past a certain point, on this example about 20 machines.

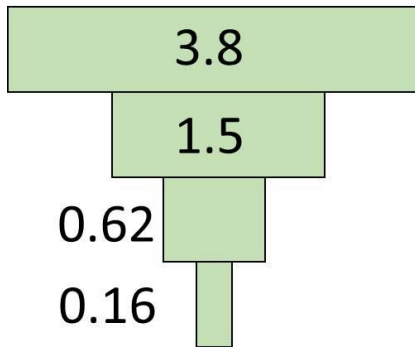


Kylix: A Scalable, Sparse Allreduce

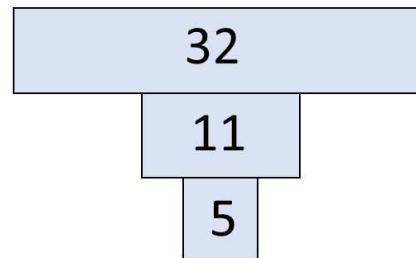
(Forthcoming paper)

- Total communication across all layers a small constant larger than the top layer, which is close to optimal.
- Communication volume across layers has a characteristic Kylix shape.

Twitter (8x4x2)



Yahoo (16x4)

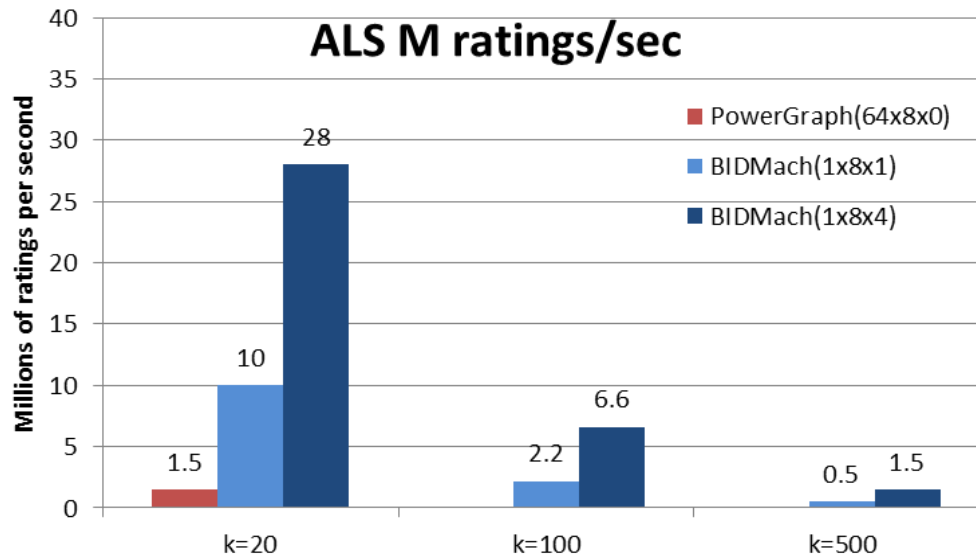


Learner Output

```
1.00%, ll=-4.985, gf=71.878, secs=116.9, GB=10.04, MB/s=85.87, GPUmem=0.57, 0.57, 0.57, 0.57
2.00%, ll=-4.852, gf=67.469, secs=254.9, GB=20.54, MB/s=80.56, GPUmem=0.57, 0.57, 0.57, 0.57
3.00%, ll=-4.824, gf=68.385, secs=379.8, GB=31.00, MB/s=81.62, GPUmem=0.57, 0.57, 0.57, 0.57
4.00%, ll=-4.803, gf=68.469, secs=517.2, GB=42.27, MB/s=81.73, GPUmem=0.57, 0.57, 0.57, 0.57
5.00%, ll=-4.787, gf=69.333, secs=639.4, GB=52.91, MB/s=82.74, GPUmem=0.57, 0.57, 0.57, 0.57
6.00%, ll=-4.784, gf=69.589, secs=768.7, GB=63.84, MB/s=83.04, GPUmem=0.57, 0.57, 0.57, 0.57
7.00%, ll=-4.784, gf=70.226, secs=892.2, GB=74.77, MB/s=83.80, GPUmem=0.57, 0.57, 0.57, 0.57
8.00%, ll=-4.762, gf=70.415, secs=1023.6, GB=86.00, MB/s=84.02, GPUmem=0.57, 0.57, 0.57, 0.57
9.00%, ll=-4.765, gf=70.492, secs=1135.5, GB=95.50, MB/s=84.10, GPUmem=0.57, 0.57, 0.57, 0.57
10.00%, ll=-4.761, gf=70.488, secs=1260.1, GB=105.97, MB/s=84.10, GPUmem=0.57, 0.57, 0.57, 0.57
11.00%, ll=-4.762, gf=70.346, secs=1373.9, GB=115.29, MB/s=83.92, GPUmem=0.57, 0.57, 0.57, 0.57
12.00%, ll=-4.758, gf=70.087, secs=1496.1, GB=125.09, MB/s=83.61, GPUmem=0.57, 0.57, 0.57, 0.57
13.00%, ll=-4.760, gf=69.812, secs=1621.2, GB=135.01, MB/s=83.28, GPUmem=0.57, 0.57, 0.57, 0.57
14.00%, ll=-4.756, gf=69.549, secs=1752.5, GB=145.40, MB/s=82.97, GPUmem=0.57, 0.57, 0.57, 0.57
15.00%, ll=-4.753, gf=69.229, secs=1890.2, GB=156.12, MB/s=82.59, GPUmem=0.57, 0.57, 0.57, 0.57
16.00%, ll=-4.748, gf=68.930, secs=2016.9, GB=165.87, MB/s=82.24, GPUmem=0.57, 0.57, 0.57, 0.57
17.00%, ll=-4.752, gf=68.697, secs=2136.9, GB=175.16, MB/s=81.97, GPUmem=0.57, 0.57, 0.57, 0.57
18.00%, ll=-4.749, gf=68.411, secs=2275.6, GB=185.74, MB/s=81.62, GPUmem=0.57, 0.57, 0.57, 0.57
19.00%, ll=-4.759, gf=68.125, secs=2426.5, GB=197.24, MB/s=81.29, GPUmem=0.57, 0.57, 0.57, 0.57
20.00%, ll=-4.751, gf=67.889, secs=2573.0, GB=208.40, MB/s=80.99, GPUmem=0.57, 0.57, 0.57, 0.57
21.00%, ll=-4.740, gf=67.661, secs=2718.3, GB=219.43, MB/s=80.72, GPUmem=0.57, 0.57, 0.57, 0.57
22.00%, ll=-4.760, gf=67.407, secs=2855.3, GB=229.62, MB/s=80.42, GPUmem=0.57, 0.57, 0.57, 0.57
23.00%, ll=-4.760, gf=67.179, secs=2986.0, GB=239.29, MB/s=80.14, GPUmem=0.57, 0.57, 0.57, 0.57
24.00%, ll=-4.755, gf=66.968, secs=3132.1, GB=250.21, MB/s=79.89, GPUmem=0.57, 0.57, 0.57, 0.57
25.00%, ll=-4.756, gf=66.776, secs=3266.1, GB=260.16, MB/s=79.66, GPUmem=0.57, 0.57, 0.57, 0.57
```

Benchmarks

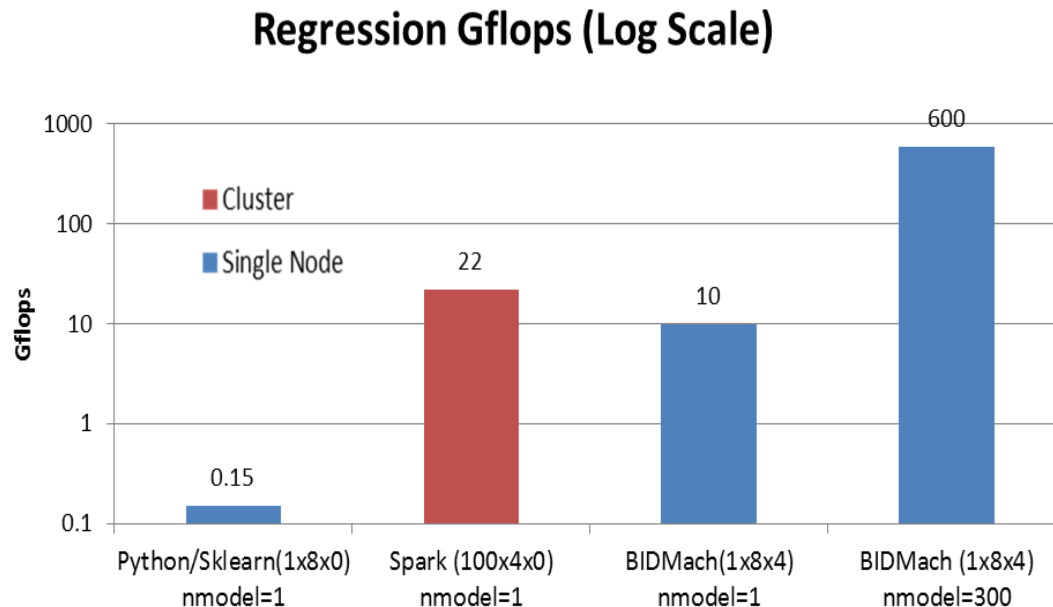
Alternating Least Squares: (synthetic Netflix Data)



- i.e. order of magnitude speedup for single-node vs. 64-node cluster, or 1000x speedup in per-node throughput.
- Uses the SDDMM matrix primitive and interleaved conjugate gradient updates (KDD 2013 paper).
- About 80 Gflops end-to-end throughput w/ 4 GPUs.

Benchmarks

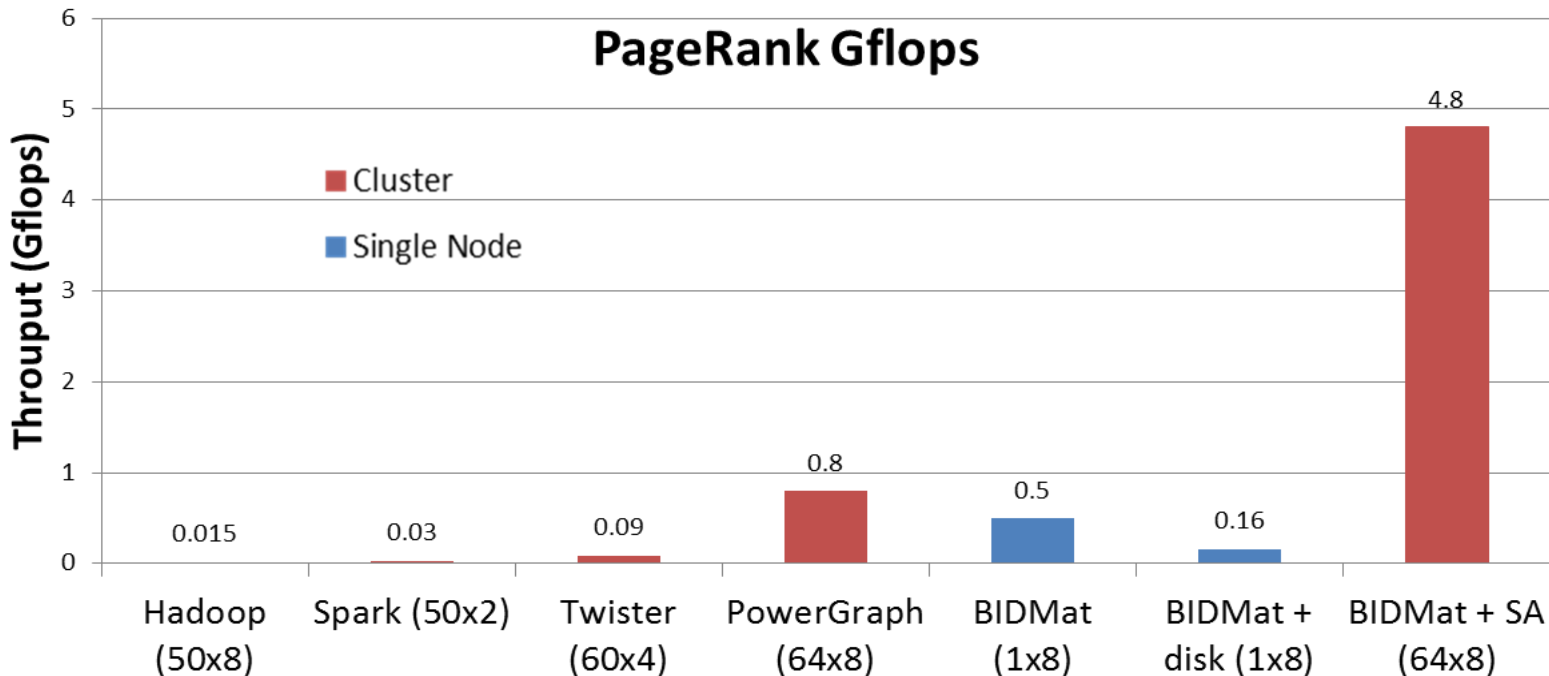
Logistic Regression: (100GB Twitter)



- i.e. single-node implementation takes 2x time, and has 50x the per-node throughput.
- But for multi-model regression (many different targets), BIDMach achieves 50x the throughput (one node vs 100), and 5000x the per-node throughput.

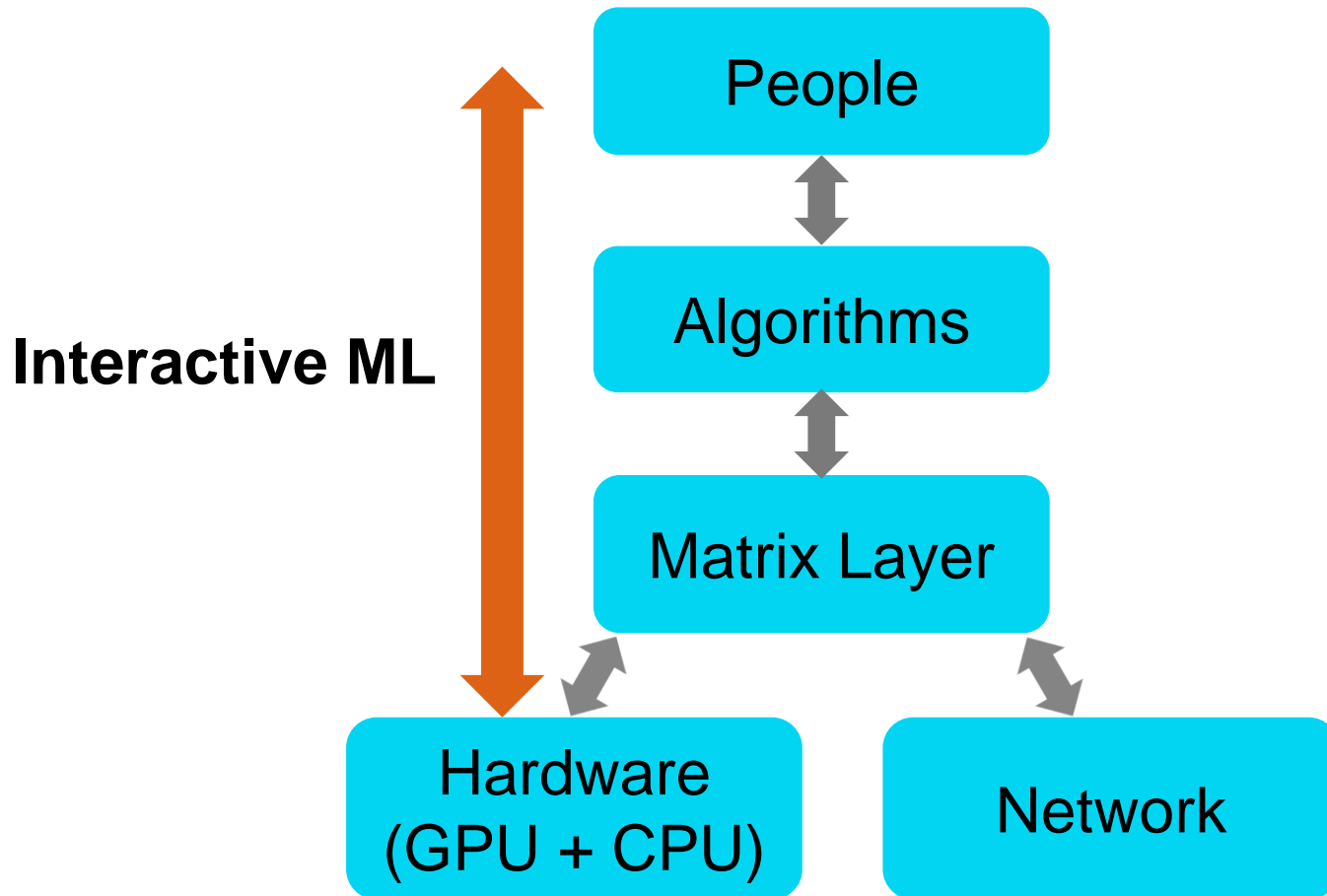
Benchmarks

Pagerank Iteration (using Sparse Allreduce)



i.e. for in-memory data, single-node performance is comparable with a 64-node cluster (about 40x faster in per-node throughput)

Toward Interactive Machine Learning



Gibbs Sampling

The most general method for inference on probabilistic graphical models:

- **Simple** to specify and implement
- **Flexible** (grouping, ordering)
- **Unbiased**
- Allows estimation of **arbitrary statistics**

But:

- Slow!!
- Hard to do parameter optimization

EM and Cooled Gibbs Sampling

EM: Separate parameters from other latent variables: joint is $P(X, \Theta)$, maximize $P(\Theta)$ and compute expected log likelihood.

Standard Gibbs: blocked sample from $P(X \mid \Theta)$ and $P(\Theta \mid X)$

Cooled Gibbs: sample from $P(X_1, \dots, X_k \mid \Theta)$ and $P(\Theta \mid X_1, \dots, X_k)$ for independent groups X_i

The X_i have the same conditional distribution as before.

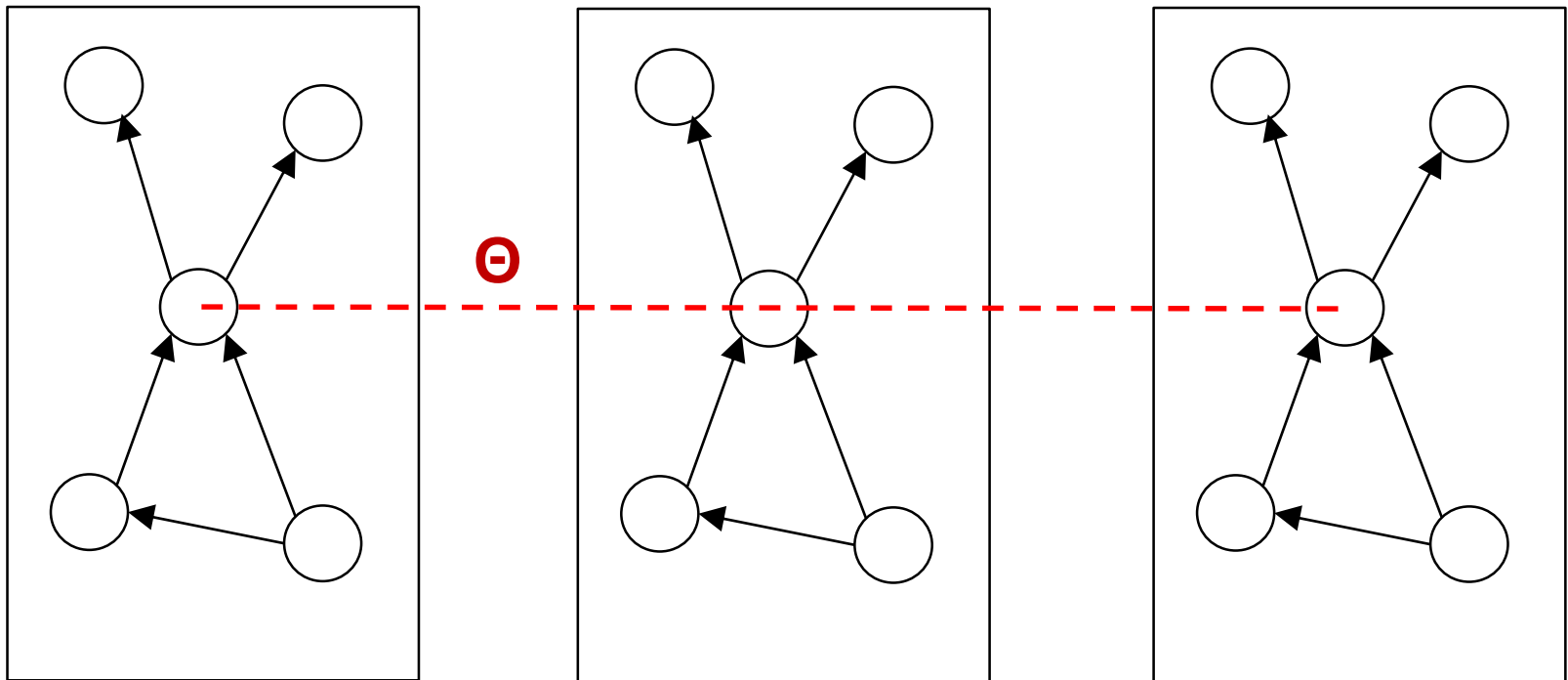
parameters now $\sim P^k(\Theta)$, i.e. the parameter distribution cooled to $T=1/k$.

The samples X_i can often be computed very fast.

EM and Cooled Gibbs Sampling

In the language of graphical models:

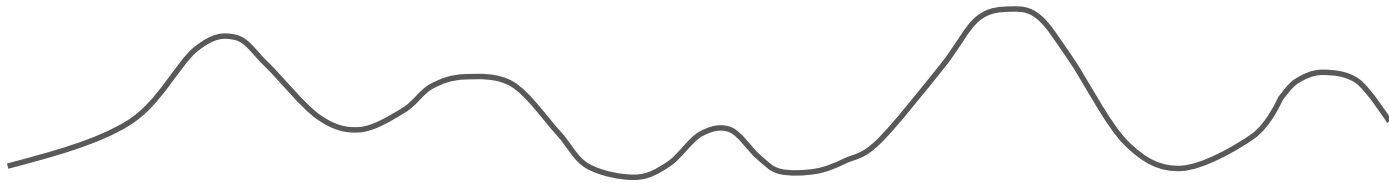
Run independent simulations with tied parameters Θ



EM and Cooled Gibbs Sampling

What cooling does:

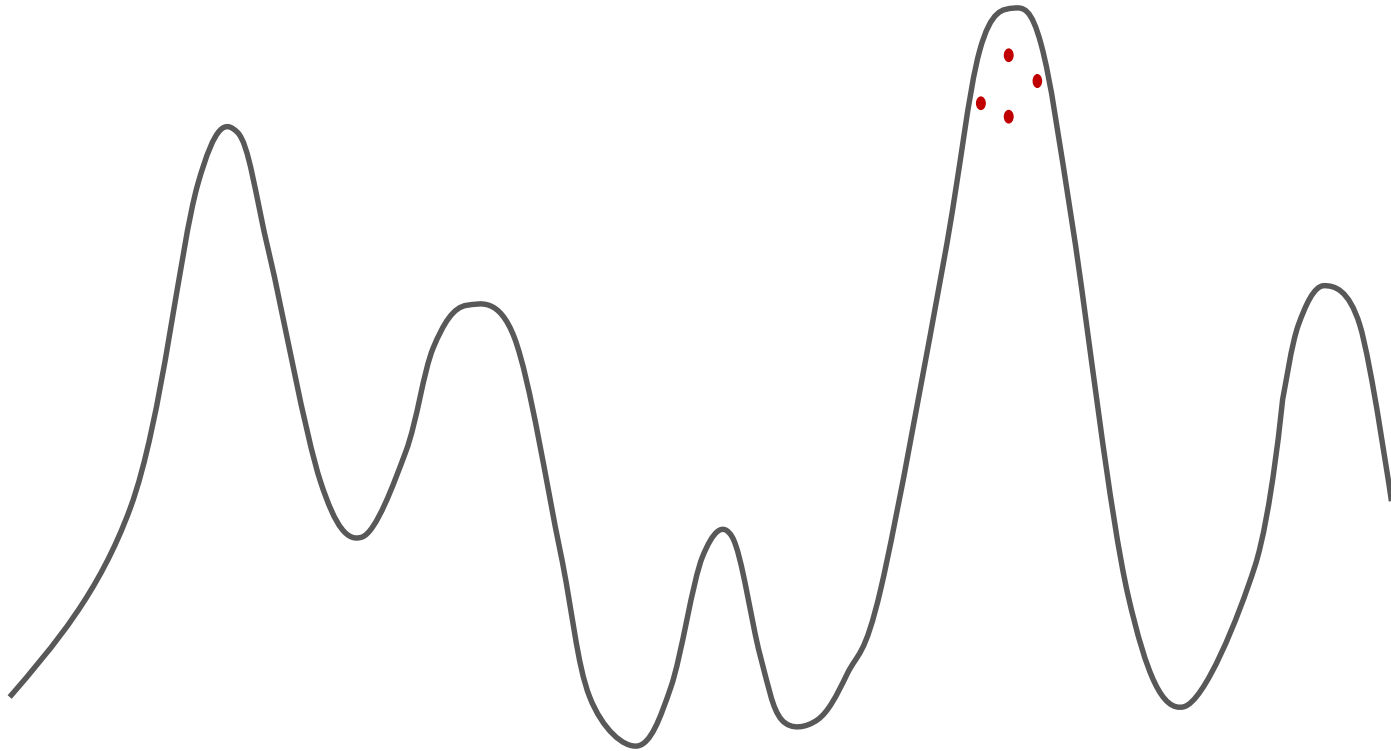
Likelihood function in model parameter space (peaks are good models)



EM and Cooled Gibbs Sampling

What cooling does:

Likelihood function in model parameter space (peaks are good models)



Cooled Gibbs Sampling

The “fastest” version of this sampler represents a collection of samples by its average.

For some models, e.g. LDA, other factor models, the fastest sampler is also exact.

The fast sampler gives a **two order-of-magnitude** speedup for inference on LDA models.

We can use both samplers on general graphical models:

- Run the fast, cooled sampler to convergence.
- Run the exact cooled sampler for a few iterations.

Toward Interactive Modeling



We can control the temperature of individual parameters in a model, and use this for human-supervised search. See Biye's poster.

Future

“Caffeinated” BIDMach:

- Wrapping a DNN toolkit called CAFFE with a Java native API

Genomics Module:

- Very fast, bit-level edit distance (2 Tcups)
- Sorting (the new hashing)
- Probabilistic alignment/assembly
- Cleaving, reversing, filtering,...

Summary

- You can achieve order-of-magnitude speedups for general machine learning through roofline design (BIDMach).
- With GPU acceleration, you gain a further order of magnitude.
- You can scale the performance of GPU-accelerated ML, but not with current MapReduce frameworks.
- Exciting possibilities for fundamental improvements in ML through deep codesign (model compilation, cooled sampling).

Software

Code:

github.com/BIDData/BIDMat

github.com/BIDData/BIDMach

BSD-style open source libs and dependencies,

Amazon AMI for test-driving...

<http://bid2.berkeley.edu/bid-data-project/overview/>