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

Digging Around in Data → Hypothesize Model → Evaluate Interpret → Large Scale Exploitation
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

- **Samples**
- **Features**

- **DATA**

**Incremental-update Methods**

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

**Large Datasets:** Mini-batch model updates

- **Spark:** UC Berkeley
- **HaLoop:** U. Washington
- **Mahout**

**Deep Learning**

- **Torch7:** (NYU, NEC)
- **Convnet, RNNLib, Visual-RBM:** Toronto
- **Theano:** Montreal

**BIDMat/BIDMach:** (this talk)

**Downpour SGD:** (Google)

**Hogwild:** U. Wisc.-Madison
GPUs at a glance…

Intel® CPU

Memory Controller

<table>
<thead>
<tr>
<th>ALU</th>
<th>ALU</th>
<th>ALU</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>Core</td>
<td>Core</td>
<td>Core</td>
</tr>
</tbody>
</table>

L3 Cache

NVIDIA® GPU

L2 Cache
Vive La Difference!

Intel® CPU

Memory Controller

<table>
<thead>
<tr>
<th>ALU</th>
<th>ALU</th>
<th>ALU</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>Core</td>
<td>Core</td>
<td>Core</td>
</tr>
</tbody>
</table>

L3 Cache

4kB registers:

NVIDIA® GPU

L2 Cache

Hardware transcendentalists (power series)

4 MB register file (!)
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

<table>
<thead>
<tr>
<th>Intel® 8 core Sandy Bridge CPU</th>
<th>NVIDIA® GK110 GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>4kB registers: 5 TB/s</td>
<td>4 MB register file (1) 40 TB/s</td>
</tr>
<tr>
<td>512K L1 Cache 1 TB/s</td>
<td>1 MB Constant Mem 13 TB/s</td>
</tr>
<tr>
<td>2 MB L2 Cache</td>
<td>1 MB Shared Mem 1 TB/s</td>
</tr>
<tr>
<td>8 MB L3 Cache 500 GB/s</td>
<td>1.5 MB L2 Cache 500 GB/s</td>
</tr>
<tr>
<td>10s GB Main Memory 20 GB/s</td>
<td>4 GB Main Memory 150 GB/s</td>
</tr>
</tbody>
</table>
Hi Speed CPU kernels

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
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: 20 GB/s
- 10s GB Main Memory: 150 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
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):

```c
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);
```
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: 20 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
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.
The compiler’s version

```c
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.
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

- People
- Algorithms
- Matrix Layer
- Hardware (GPU + CPU)
- Network
A GPU-enabled Matrix Tool

Written in the beautiful Scala language:

- Interpreter, w/ excellent performance
- Natural syntax +,-,*, °,●,⊗ 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

DataSource (Memory)

DataSource (JBOD disks)

DataSource HDFS over network

CPU host code

Learner

data blocks

Model
Optimizer
Regularizer
Mixins

Model
Optimizer
Regularizer
Mixins

GPU 1 thread 1

4 GPUs: 80 Gflops to 3 Teraflops typical

GPU 2 thread 2

Compressed disk streaming at
~ 1.5GB/s ≈ 40-100 Hadoop nodes
Latent Dirichlet Allocation Model:

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

- Cluster
- Single Node

<table>
<thead>
<tr>
<th>System</th>
<th>Million documents per hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smola [5] (100x8x0)</td>
<td>1.6</td>
</tr>
<tr>
<td>PowerGraph (64x16x0)</td>
<td>1.1</td>
</tr>
<tr>
<td>BIDMach (1x8x1)</td>
<td>3.6</td>
</tr>
<tr>
<td>BIDMach (1x8x4)</td>
<td>10</td>
</tr>
</tbody>
</table>

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.
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.

![Graph showing bandwidth vs packet size](image-url)
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)  

<table>
<thead>
<tr>
<th>Layer 1</th>
<th>Layer 2</th>
<th>Layer 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.8</td>
<td>1.5</td>
<td>0.62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.16</td>
</tr>
</tbody>
</table>

Yahoo (16x4)  

<table>
<thead>
<tr>
<th>Layer 1</th>
<th>Layer 2</th>
<th>Layer 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>Percentage</td>
<td>ll</td>
<td>gf</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>1.00%</td>
<td>-4.985</td>
<td>71.878</td>
</tr>
<tr>
<td>2.00%</td>
<td>-4.852</td>
<td>67.469</td>
</tr>
<tr>
<td>3.00%</td>
<td>-4.824</td>
<td>68.385</td>
</tr>
<tr>
<td>4.00%</td>
<td>-4.803</td>
<td>68.469</td>
</tr>
<tr>
<td>5.00%</td>
<td>-4.787</td>
<td>69.333</td>
</tr>
<tr>
<td>6.00%</td>
<td>-4.784</td>
<td>69.589</td>
</tr>
<tr>
<td>7.00%</td>
<td>-4.784</td>
<td>70.226</td>
</tr>
<tr>
<td>8.00%</td>
<td>-4.762</td>
<td>70.415</td>
</tr>
<tr>
<td>9.00%</td>
<td>-4.762</td>
<td>70.492</td>
</tr>
<tr>
<td>10.00%</td>
<td>-4.761</td>
<td>70.415</td>
</tr>
<tr>
<td>11.00%</td>
<td>-4.762</td>
<td>70.346</td>
</tr>
<tr>
<td>12.00%</td>
<td>-4.758</td>
<td>70.087</td>
</tr>
<tr>
<td>13.00%</td>
<td>-4.760</td>
<td>69.812</td>
</tr>
<tr>
<td>14.00%</td>
<td>-4.756</td>
<td>69.549</td>
</tr>
<tr>
<td>15.00%</td>
<td>-4.753</td>
<td>69.229</td>
</tr>
<tr>
<td>16.00%</td>
<td>-4.748</td>
<td>68.930</td>
</tr>
<tr>
<td>17.00%</td>
<td>-4.752</td>
<td>68.697</td>
</tr>
<tr>
<td>18.00%</td>
<td>-4.749</td>
<td>68.411</td>
</tr>
<tr>
<td>19.00%</td>
<td>-4.759</td>
<td>68.125</td>
</tr>
<tr>
<td>20.00%</td>
<td>-4.751</td>
<td>67.889</td>
</tr>
<tr>
<td>21.00%</td>
<td>-4.740</td>
<td>67.661</td>
</tr>
<tr>
<td>22.00%</td>
<td>-4.760</td>
<td>67.407</td>
</tr>
<tr>
<td>23.00%</td>
<td>-4.760</td>
<td>67.179</td>
</tr>
<tr>
<td>24.00%</td>
<td>-4.755</td>
<td>66.968</td>
</tr>
<tr>
<td>25.00%</td>
<td>-4.756</td>
<td>66.776</td>
</tr>
</tbody>
</table>
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.
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

Interactive ML

- People
- Algorithms
- Matrix Layer
- Hardware (GPU + CPU)
- Network
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**: 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 $P(X_1, \ldots, X_k \mid \Theta)$ and $P(\Theta \mid X_1, \ldots, X_k)$ 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 $\Theta$
What cooling does:
Likelihood function in model parameter space (peaks are good models)
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.
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/BIDDdata/BIDMat
github.com/BIDDdata/BIDMach

BSD-style open source libs and dependencies,
Amazon AMI for test-driving…
http://bid2.berkeley.edu/bid-data-project/overview/