Head, Torso and Tail
Performance for modeling real data

Alex Smola
CMU ➔ Amazon

MACHINE LEARNING DEPARTMENT
Outline

• Sparse power law data
  • Generic sparsity not efficient
  • Statistical and computational advantages from modeling distributions differently

• **Memory - Recommender Systems**
  Cache head, store torso, prefetch tail

• **Model - Factorization Machines**
  Large head, tiny tail
Memory
Memory

http://www.theregister.co.uk/2016/04/04/memory_and_storage_boundary_changes/

10MB 10GB 500GB 5TB 100TB
1ns 100ns 10μs 10ms 100s
Numbers you should know

<table>
<thead>
<tr>
<th>Memory Access Type</th>
<th>Nanoseconds (ns)</th>
<th>Microseconds (us)</th>
<th>Milliseconds (msec)</th>
<th>If L1 access is 1 second</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
<td></td>
<td></td>
<td>1 sec</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
<td></td>
<td></td>
<td>14 secs</td>
</tr>
<tr>
<td>DRAM access</td>
<td>200</td>
<td></td>
<td></td>
<td>6 mins 40 secs</td>
</tr>
<tr>
<td>NVDIMM-N</td>
<td>5,000</td>
<td>5</td>
<td></td>
<td>2 hours 46 mins 40 secs</td>
</tr>
<tr>
<td>NVMe PCIe SSD write</td>
<td>30,000</td>
<td>30</td>
<td></td>
<td>16 hours 40 mins</td>
</tr>
<tr>
<td>Mangstor NX NVMeF array write</td>
<td>30,000</td>
<td>30</td>
<td></td>
<td>16 hours 40 mins</td>
</tr>
<tr>
<td>Zstor NVMe-F SSD array read</td>
<td>30,000</td>
<td>30</td>
<td></td>
<td>16 hours 40 mins</td>
</tr>
<tr>
<td>DSSD D5 NVMe-F array</td>
<td></td>
<td></td>
<td></td>
<td>55 hours 33 mins 20 secs</td>
</tr>
<tr>
<td>Mangstor NX NVMeF array read</td>
<td></td>
<td></td>
<td></td>
<td>61 hours 6 mins 40 secs</td>
</tr>
<tr>
<td>NVMe PCIe SSD read</td>
<td></td>
<td></td>
<td></td>
<td>61 hours 6 mins 40 secs</td>
</tr>
<tr>
<td>Zstor NVMe-F SSD array write</td>
<td></td>
<td></td>
<td></td>
<td>61 hours 6 mins 40 secs</td>
</tr>
<tr>
<td>Random 4K read from SSD</td>
<td></td>
<td></td>
<td></td>
<td>3 days, 11 hours, 20 mins</td>
</tr>
<tr>
<td>Sequential Read 1MB from DRAM</td>
<td>250,000</td>
<td>250</td>
<td></td>
<td>5 days, 18 hours, 53 mins, 20 secs</td>
</tr>
<tr>
<td>Round trip i datacenter</td>
<td>500,000</td>
<td>500</td>
<td>0.5</td>
<td>11 days, 13 hours, 46 mins, 40 secs</td>
</tr>
<tr>
<td>Sequential read 1MB from SSD</td>
<td>1,000,000</td>
<td>1,000</td>
<td>1</td>
<td>23 days, 3 hours, 33 mins, 20 secs</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000</td>
<td>10,000</td>
<td>10</td>
<td>231 days, 11 hours, 33 mins, 20 secs</td>
</tr>
<tr>
<td>Sequential read 1MB from disk</td>
<td>20,000,000</td>
<td>20,000</td>
<td>20</td>
<td>1 year, 97 days, 23 hours, 6 mins, 40 secs</td>
</tr>
<tr>
<td>DAS disk access</td>
<td>100,000,000</td>
<td>100,000</td>
<td>100</td>
<td>6 years, 119 days, 19 hours, 33 mins, 20 secs</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000</td>
<td>150</td>
<td>150</td>
<td>9 years, 185 days, 5 hours, 20 mins, 0 secs</td>
</tr>
<tr>
<td>SAN array access</td>
<td>300,000,000</td>
<td>300,000</td>
<td>300</td>
<td>19 years, 5 days, 10 hours, 30 mins, 0 secs</td>
</tr>
</tbody>
</table>

http://www.theregister.co.uk/2016/04/04/memory_and_storage_boundary_changes/
Use Case: Recommender Systems

- Users $u$, movies $m$ (or projects)
- Function class
  \[ r_{um} = \langle v_u, w_m \rangle + b_u + b_m \]
- Loss function for recommendation (Yelp, Netflix)
  \[ \sum_{u \sim m} (\langle v_u, w_m \rangle + b_u + b_m - y_{um})^2 \]
Use Case: Recommender Systems

- Regularized Objective
  \[ \sum_{u \sim m} (\langle v_u, w_m \rangle + b_u + b_m + b_0 - r_{um})^2 + \frac{\lambda}{2} \left[ \|U\|_{\text{Frob}}^2 + \|V\|_{\text{Frob}}^2 \right] \]

- Update operations
  \[ v_u \leftarrow (1 - \eta_t \lambda)v_u - \eta_t w_m (\langle v_u, w_m \rangle + b_u + b_m + b_0 - r_{um}) \]
  \[ w_m \leftarrow (1 - \eta_t \lambda)w_m - \eta_t v_u (\langle v_u, w_m \rangle + b_u + b_m + b_0 - r_{um}) \]

- Very simple SGD algorithm (random pairs)
- This should be cheap ...

memory subsystem
Not so cheap …

- **Netflix contest**
  - 100M samples, 2048 dimensions, 30 steps
  - $100M \times 2048 \times 30 \times 4 \times 8\text{byte burst reads}$
  - $100M \times 30 \times 4$ random reads
- **Runtime** (1h 15 min)
  - 3300s for burst reads (@60GB/s)
  - 1200s for random reads (@10ns/read)
- **Better engineering gets 9.5 min. How?**
  
  Liu, Wang, Smola, RecSys 2015
Power laws in data

Netflix dataset

# movies

# ratings

10^4
10^3
10^2
10^1
10^0
10^1
10^2
10^3
10^4
10^5
10^6

head
torso
tail
Power laws in data

Netflix dataset

large memory footprint

lots of requests

# movies vs. # ratings

SLAM Agency

Carnegie Mellon University
Power laws in data

Netflix dataset

not too bad @ 2048
20k movies = 160MB

100M requests
Memory subsystem

- Processes
- Cache
- RAM local state
Key Ideas

- **Stratify ratings by users**
  (only 1 cache miss / read per user / out of core)
- **Keep frequent movies in cache**
  (stratify by blocks of movie popularity)
- **Avoid false sharing between sockets**
  (key cached in the wrong CPU causes miss)

<table>
<thead>
<tr>
<th>K</th>
<th>SC-SGD L1 Cache</th>
<th>SC-SGD L3 Cache</th>
<th>GraphChi L1 Cache</th>
<th>GraphChi L3 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2.84%</td>
<td>0.43%</td>
<td>12.77%</td>
<td>2.21%</td>
</tr>
<tr>
<td>256</td>
<td>2.85%</td>
<td>0.50%</td>
<td>12.89%</td>
<td>2.34%</td>
</tr>
<tr>
<td>2048</td>
<td>3.3%</td>
<td>1.7%</td>
<td>15%</td>
<td>9.8%</td>
</tr>
</tbody>
</table>
Key Ideas

GraphChi
Partitioning

<table>
<thead>
<tr>
<th>K</th>
<th>SC-SGD</th>
<th></th>
<th>GraphChi</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1 Cache</td>
<td>L3 Cache</td>
<td>L1 Cache</td>
<td>L3 Cache</td>
</tr>
<tr>
<td>16</td>
<td>2.84%</td>
<td>0.43%</td>
<td>12.77%</td>
<td>2.21%</td>
</tr>
<tr>
<td>256</td>
<td>2.85%</td>
<td>0.50%</td>
<td>12.89%</td>
<td>2.34%</td>
</tr>
<tr>
<td>2048</td>
<td>3.3%</td>
<td>1.7%</td>
<td>15%</td>
<td>9.8%</td>
</tr>
</tbody>
</table>
### Key Ideas

SC-SGD partitioning

<table>
<thead>
<tr>
<th>K</th>
<th>SC-SGD</th>
<th>GraphChi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1 Cache</td>
<td>L1 Cache</td>
</tr>
<tr>
<td></td>
<td>L3 Cache</td>
<td>L3 Cache</td>
</tr>
<tr>
<td>16</td>
<td>2.84%</td>
<td>12.77%</td>
</tr>
<tr>
<td></td>
<td>0.43%</td>
<td>2.21%</td>
</tr>
<tr>
<td>256</td>
<td>2.85%</td>
<td>12.89%</td>
</tr>
<tr>
<td></td>
<td>0.50%</td>
<td>2.34%</td>
</tr>
<tr>
<td>2048</td>
<td>3.3%</td>
<td>15%</td>
</tr>
<tr>
<td></td>
<td>1.7%</td>
<td>9.8%</td>
</tr>
</tbody>
</table>
Speed (c4.8xlarge)

Netflix - 100M, 15 iterations

Yahoo - 250M, 30 iterations
Convergence

- GraphChi blocks (users, movies) into random groups
- This slows down convergence
Sanity Check
What to expect

- **c4.8xlarge**
  - 200 GFlops for dense SGEMM (geekbench)
  - Much less for SpMV (probably < 10 GFlops)
- **M40 GPU**
  - 6.5 TFlops for dense SGEMM (NVIDIA)
  - TensorFlow sparse autoencoder 10 GFlops
  - Amazon DSSTNE autoencoder 64 GFlops

https://medium.com/@scottlegrand/first-dsstne-benchmarks-tldr-almost-15x-faster-than-tensorflow-393dbeb80c0f#.tev6o3lgm

Scott LeGrand’s MovieLens benchmark (embedding, 1024d, 3 dense sigmoid layers)
We get 40 GFlops (200 peak)

- 100M rating pairs
- 2048 dimensions
- 20s / iteration
- $10^8 \times 2048 \times 2 \times 2 / 20s$
Takeaway

• Many sparse codes ignore that **data is nice**
  • Head in cache
  • Torso in RAM
  • Tail out of core (or regularize away)
• **Whole system**
  • Larger footprint to use SSD/RAM/Cache
  • Extend to GPUs (host RAM is just another caching layer)
  • Automatic layout and detection
A Linear Model is not enough
Factorization Machines

- Linear Model
  \[ f(x) = \langle w, x \rangle \]

- Polynomial Expansion (Rendle, 2012)
  \[ f(x) = \langle w, x \rangle + \sum_{i<j} x_i x_j \text{tr} \left( V_i^{(2)} \otimes V_j^{(2)} \right) + \]
  \[ \sum_{i<j<k} x_i x_j x_k \text{tr} \left( V_i^{(3)} \otimes V_j^{(3)} \otimes V_k^{(3)} \right) + \ldots \]
Factorization Machines

- Special cases
  - Recommender systems
    \[ x = (\text{user}, \text{movie}) \]
  - Feature based methods
    \[ x = (\text{user}, \text{movie}, \text{features}) \]
  - Time-dependent recommendation
    \[ x = (\text{user}, \text{movie}, \text{time}) \]
  - Advertising
    \[ x = (\text{sparse features}) \]
Prefetching to the rescue

- Most keys are infrequent (power law distribution)
- Prefetch the embedding vectors for a minibatch from parameter server
- Compute gradients and push to server
  - Variable dimensionality embedding
  - Enforcing sparsity (ANOVA style)
  - Adaptive gradient normalization
  - Frequency adaptive regularization (CF style)
ParameterServer 101

bipartite design

high cross section BW
**Parameter Server 101**

- Clients have local parameter view
- Servers have shard of parameter space
- Client-server synchronization
  - Reconciliation protocol
  - Synchronization schedule
  - Load distribution algorithm

**Smola & Narayanamurthy, 2010, VLDB**
**Gonzalez et al., 2012, WSDM**
**Dean et al., 2012, NIPS**
**Shervashidze et al., 2013, WWW**

Google, Baidu, Facebook, Amazon, Yahoo, Microsoft

**put(keys, values, clock)**
**get(keys, values, clock)**
Killing Features

- Need high dimensions only when we have enough data
- Most features are sparse. Hence kill embedding
- Use sparsity in linear model to guide nonlinear part

feature frequency

head

saved RAM
torso
tail
Figure 2: Using different adaptive memory constraints when varying the embedding dimension. Left: Criteo data. Right: CTR data.

A more interesting observation is that these memory adaptive constraints do not affect the test accuracy. To the contrary, we even see a slight improvement when the dimension $k$ is greater than 8 for CTR data. The reason could be that the model capacity control is of great importance when the dimension $k$ is large. And these memory adaptive constraints can provide additional capacity control besides the $\ell^2$ and $\ell^1$ regularizers.

5.3 Fixed-point Compression

We evaluate lossy fixed-point compression for data communication. By default, both the model and gradient entries are represented as 32 bit floats. In this experiment, we compress these values to lower precision integers. More specifically, given a bin size $b$ and number of bits $n$, we represent $x$ by the following $n$-bit integer $z = j x b \cdot 2^n k + z$, where $j$ is a Bernoulli random variable chosen such as to ensure that $E[z] = 2^n x b$. 

(Criteo 1TB)
Faster Solver (small Criteo)

(a) Total data sent by workers in one iteration. The compression rates from 4-byte to 1-byte are 4.2x and 2.9x for Criteo2 and CTR2, respectively.

(b) The relative test logloss comparing to no fixed-point compression.

Figure 3: Compressing model and gradient using the fixed-point compression, where 4-byte means using the default 32-bit floating-point format.

We implemented the fixed-point compression as a user-defined filter in the parameter server framework. Since multiple numbers are communicated in each round, we choose $b$ to be the absolute maximum value of these numbers. In addition, we used the key caching and lossless data compression (via LZ4) filters.

We observed different effects of accuracy on these two datasets: CTR2 is robust to the number precision, while Criteo2 has a 6% increase of logloss if only using 1-byte presentation. However, a medium compression rate even improves the model accuracy. This might be because the lossy compression acts as a regularization to the objective function.

5.4 Comparison with LibFM

To our best knowledge, there is no publicly released distributed FM solver. Hence we only compare DiFacto to the popular single machine package LibFM developed by Renngle [14]. We only report results on Criteo1 and CTR1 on a single machine, since LibFM fails on the other two larger datasets. We perform a similar grid search of the hyperparameters as we did for DiFacto. As LibFM only uses single thread, we run DiFacto with 1 worker and 1 server in sequential execution order. We also report the performance using 10 workers and 10 servers on a single machine for reference.

The results are shown in Figure 4. As can be seen, DiFacto converges significantly faster than LibFM, it uses 2 times fewer iterations to reach the best model. This is because the adaptive learning rate used in DiFacto better models the data sparsity and the adaptive regularization and constraints can further accelerate the convergence. In particular, the latter results in a lower test logloss on the CTR1 dataset, where the number of features exceeds the number of examples, requiring improved capacity control.

Also note that DiFacto with a single worker is twice slower than LibFM per iteration. This is because the data communicated to the workers is much larger than the model size. LibFM died on large models.
There is a longstanding suspicion that the convergence of the objective logloss on test datasets is below 0.5%, which is the difference in model accuracy. In other words, the relative difference between the test logloss is within 0.5%, which is the difference in model accuracy. In other words, the relative difference between the test logloss is within 0.5%, which is the difference in model accuracy. In other words, the relative difference between the test logloss is within 0.5%, which is the difference in model accuracy.
Summary

- Sparse power law data
- Generic sparsity not efficient
- Statistical and computational advantages from modeling distributions differently
- Memory
  Cache head, store torso, prefetch tail
- Model
  Large head, tiny tail (safe screening rules)

I am hiring - alex@smola.org