SLIDE: In Defense of Smart Algorithms over Hardware Acceleration for Large-Scale Deep Learning Systems

Beidi Chen

Collaborators: Tharun Medini*, James Farwell†, Sameh Gobriel†, Charlie Tai †, Anshumali Shrivastava*

* Rice University, † Intel

MLSys 2020
Our SLIDE System (C++ from scratch) on a 44 core CPU beats TF on V100 (1 hours vs 3.5 hours). 100+ million parameter networks. TF on same CPU is 16 hours with all HPC optimization (Intel MKL-DNN).

3.5x faster on CPU than TF on V100 (Log Scale in Time)
The Age of Large Networks

- More Data
- Large Models
- Tons of Engineering
- Backpropagation
  (Aka Simple Gradient Descent)
Fully Connected NN

Giant Matrix Multiplication for every data point in each epoch (Forward + Backward)

\[ f(W^T x) \]
Challenges

Do we really need all the computations?

No!!

**Good News:** Only high activations are important

- Sampling few neurons in proportion of activations is enough (**Adaptive Dropouts**)
  
  (Ba et al. Neurips 13, Makhzani et al. Neurips 15)

- Relu filtered negative activations (50% sparsity by design)

- Softmax

**Bad News:** We need to compute all to identify (or sample) the high activation neurons.

**NO SAVINGS**
The Fundamental Sampling Puzzle

Given N fixed sampling weights, \( \{w_1, w_2, ..., w_N\} \).

- Task: Sample \( x_i \) with probability \( w_i \)
  - Cost of 1 sample \( O(N) \).
  - Cost of K samples \( O(N) \).

Given N time-varying sampling weights (activations) \( \{w_1^t, w_2^t, ..., w_N^t\} \).

- Task: At time \( t \), sample \( x_i \) with probability \( w_i^t \)
  - Cost of sampling \( O(N) \), at every time \( t \).
  - Last Few years of work in Locality Sensitive Hashing: If \( w_i^t = f(sim(\theta_t, x_i)) \), for a specific set of \( f \) and \( sim \), then \( O(1) \) every time after and initial preprocessing cost of \( O(N) \).
Textbook Hashing (Dictionary)

Hashing: Function $h$ that maps a given data point ($x \in R^D$) to an integer key $h : R^D \mapsto \{0, 1, 2, ..., N\}$. $h(x)$ serves as a discrete fingerprint.

Property (Ideal Hash Functions):

- If $x = y$, then $h(x) = h(y)$
- If $x \neq y$, then $h(x) \neq h(y)$
Probabilistic Fingerprinting (Hashing) (late 90s)

Hashing: Function (Randomized) $h$ that maps a given data point ($x \in R^D$) to an integer key $h : R^D \mapsto \{0, 1, 2, ..., N\}$. $h(x)$ serves as a discrete fingerprint.

Locality Sensitive Property:
- If $x = y$ $Sim(x, y)$ is high, then $h(x) = h(y)$ $Pr(h(x) = h(y))$ is high
- If $x \neq y$ $Sim(x, y)$ is low, then $h(x) \neq h(y)$ $Pr(h(x) = h(y))$ is low
Example 1: Signed Random Projection (SRP)

\[ Pr(h(x) = h(y)) = 1 - \frac{1}{\pi} \cos^{-1}(\theta) \] monotonic in \( \theta \)

A classical result from Goemans-Williamson (95)
Example 2: (Densified) Winner Take All

Original Vectors:

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0, 0, 5, 0, 0, 7, 6, 0, 0</td>
<td>0, 0, 1, 0, 0, 0, 0, 0</td>
</tr>
</tbody>
</table>

K=3

WTA hash codes: (ICCV 2011)

<table>
<thead>
<tr>
<th></th>
<th>Bin 1</th>
<th>Bin 2</th>
<th>Bin 3</th>
<th>Bin 4</th>
<th>Bin 5</th>
<th>Bin 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Θ</td>
<td>2, 1, 8</td>
<td>5, 3, 9</td>
<td>6, 2, 4</td>
<td>8, 9, 1</td>
<td>1, 7, 3</td>
<td>2, 4, 5</td>
</tr>
<tr>
<td>Θ(x)</td>
<td>0, 0, 0 (E)</td>
<td>0, 5, 0</td>
<td>7, 0, 0</td>
<td>0, 0, 0 (E)</td>
<td>0, 6, 5</td>
<td>0, 0, 0 (E)</td>
</tr>
<tr>
<td>Θ(y)</td>
<td>0, 0, 0 (E)</td>
<td>0, 1, 0</td>
<td>0, 0, 0 (E)</td>
<td>0, 0, 0 (E)</td>
<td>0, 0, 1</td>
<td>0, 0, 0 (E)</td>
</tr>
<tr>
<td>hwta(x)</td>
<td>1 (E)</td>
<td>2</td>
<td>1</td>
<td>1 (E)</td>
<td>2</td>
<td>1 (E)</td>
</tr>
<tr>
<td>hwta(y)</td>
<td>1 (E)</td>
<td>2</td>
<td>1 (E)</td>
<td>1 (E)</td>
<td>3</td>
<td>1 (E)</td>
</tr>
</tbody>
</table>

DWTA hash codes: (UAI 2018)

<table>
<thead>
<tr>
<th></th>
<th>ΗDWTA(x)</th>
<th>ΗDWTA(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1+3*C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2+3*C</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2+1*C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2+2*C</td>
<td>2</td>
</tr>
</tbody>
</table>

Yagnik (ICCV11), Chen and Shrivastava (UAI 18)
Probabilistic Hash Tables

**Given:** $Pr_h[h(x) = h(y)] = f(sim(x, y))$, $f$ is monotonic.

- Given query, if $h_1(q) = 11$ and $h_2(q) = 01$, then probe bucket with index 1101. It is a good bucket!!
- (Locality Sensitive) $h_i(q) = h_i(x)$ noisy indicator of high similarity.
- Doing better than random!!
LSH for Search (Known)

Theory

• Super-linear $O(N^{1+\rho})$ memory
• Sub-linear query time, $O(N^\rho)$
• $\rho < 1$ but generally large (close to 1) and often hard to determine

Practical Issues

• Needs lot of hash tables and distance computations for good accuracy on near-neighbors
• Buckets can be quite heavy. Poor randomness, or unfavorable data distributions
New View: Data Structures for Efficient Sampling!

Is LSH really a search algorithm?

- Given the query $\theta_t$, LSH samples $x_i$ from the dataset, with probability
  \[ w_i^t = 1 - (1 - p(x_i, \theta_t)^K)^L \]
- $w_i^t$ is proportional to $p(x_i, \theta_t)^K$ and the same similarity of $x_i, \theta_t$
- LSH is considered a black box for nearest-neighbor search. It is not!!
LSH as Samplers

We can pre-process the dataset $D$, such that

• Given any query $q$, we can sample $x \in D$ with probability
  $Const \times [1 - (1 - p(q, x)^K)^L]$ in KL hash computation and L bucket probes.

• Even $K = 1, L = 1$ is adaptive. So $O(1)$ time adaptive.

• **Adaptive**: $x$ is sampled with higher probability than $y$
  • if and only if $\text{sim}(q, x) > \text{sim}(q, y)$

We can exactly compute the sampling probability.

• $Const = \text{No of elements sampled} / \text{No of elements in Buckets}$
  (Chen et al. NeurIPS 2019)

Sufficient for Importance Sampling Estimations. Sampling cost $O(1)$. 
**SLIDE: Sub-Linear Deep learning Engine**

Step 1 – Build the hash tables by processing the weights of the hidden layers (*initialization*).

**Subtlety:** Neurons (vectors) in hash tables are not the data vectors. Reorganizing neurons.
SLIDE: Sub-LLinear Deep learning Engine

Step 2 – Hash the input to any given layer using its randomized hash function.
**SLIDE: Sub-Linear Deep learning Engine**

Step 3 – Query the hidden layer's hash table(s) for the active set using integer fingerprint.

Sample neurons in proportion to their activations.
**SLIDE: Sub-Linear Deep learning Engine**

Step 4 – Perform forward and back propagation only on the nodes in the **active set**.

Computation is in the same order of active neurons.
SLIDE: Sub-LInear Deep learning Engine

Step 5 – Update hash tables by rehashing the updated node weights.

Computation is in the same order of active neurons.
We can go very sparse if Adaptive

- Reduce both training and inference cost by 95%!
- Significantly more for larger networks. (The wider the better)
- 2 Hidden Layers
- 1000 Nodes Per Layer
Sparsity + Randomness \( \rightarrow \) Asynchronous Updates

- 3 Hidden Layers
- 1000 Nodes Per Layer
Less Computations + Asynchronous Parallelism

• Each update is computationally very small (100x+ reduction in computation and energy)

• Updates are near-independent, very low chance of conflict. Hence, parallel SGD!
SLIDE: Sub-Linear Deep learning Engine
Parallelism with OpenMP

<table>
<thead>
<tr>
<th>Node</th>
<th>Batchsize</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Active Inputs:**
  - 1
  - 0
  - 1
  - ......

- **Activation for each Inputs:**
  - 0.1
  - 0.2
  - 0.5
  - ......

- **Accumulated Gradients:**
  - 0.3
  - 0.8
  - 0.7
  - ......

- **Weights:**
  - -0.3
  - 0.8
  - -0.5
  - ......

Parallel across training samples in a batch (Extreme sparsity and randomness in gradient updates)

Thanks to the theory of **HOGWILD**!

(Recht et al. Neurips 11)
Flexible choices of Hash Functions

SLIDE supports four different LSH hash functions

- Simhash (cosine similarity)
- Winner-take-all Hashing (order)
- Densified Winner-take-all Hashing (for sparse data)*
- Minhash (jaccard similarity)

Easily add more!
Design Choices for Speed

• Vanilla sub-sampling:
  - choose sub-samples uniformly

• Top K sub-sampling:
  - rank samples and choose topk

• Hard Thresholding sub-sampling:
  - choose sub-samples that occur > threshold times
Micro-Architecture Optimization

Cache Optimization

Transparent Hugepages

Vector Processing

Software Pipelining and Prefetching
Looks Good on Paper. Does it change anything?

Baseline
State-of-the-art optimized Implementations
• TF on Intel Xeon E5-2699A v4 @ 2.40GHz CPU (FMA, AVX, AVX2, SSE4.2)
• TF on NVIDIA Tesla V100 (32GB)

VS.
SLIDE on Intel Xeon E5-2699A v4 @ 2.40GHz CPU (FMA, AVX, AVX2, SSE4.2)
• TF on NVIDIA Tesla V100 (32GB)
Datasets

<table>
<thead>
<tr>
<th></th>
<th>Delicious-200K</th>
<th>Amazon-670K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Dim</td>
<td>782,585</td>
<td>135,909</td>
</tr>
<tr>
<td>Feature Sparsity</td>
<td>0.038 %</td>
<td>0.055 %</td>
</tr>
<tr>
<td>Label Dim</td>
<td>205,443</td>
<td>670,091</td>
</tr>
<tr>
<td>Training Size</td>
<td>196,606</td>
<td>490,449</td>
</tr>
<tr>
<td>Testing Size</td>
<td>100,095</td>
<td>153,025</td>
</tr>
</tbody>
</table>

Network Architectures (Fully Connected)

- Delicious-200K 782,585 $\Rightarrow$ 128 $\Rightarrow$ 205,443 (126 million parameters)
- Amazon-670K 135,909 $\Rightarrow$ 128 $\Rightarrow$ 670,091 (103 million parameters)
Performance

Delicious-200K

Accuracy

Time (s)

Amazon-670K

Accuracy

Time (s)
Performance compared to sampled softmax

Graphs showing accuracy over time for Delicious-200K and Amazon-670K datasets. The graphs compare the performance of SLIDE CPU and TF-GPU SSM algorithms.
Performance @ Different Batchsizes

Amazon-670K, Batch_Size=64

Amazon-670K, Batch_Size=128

Amazon-670K, Batch_Size=256
Asynchronous Parallelism gets best scalability

Table: Core Utilization

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tensorflow-CPU</td>
<td>45%</td>
<td>35%</td>
<td>32%</td>
</tr>
<tr>
<td>SLIDE</td>
<td>82%</td>
<td>81%</td>
<td>85%</td>
</tr>
</tbody>
</table>
Inefficiency Diagnosis

**Tensorflow-CPU Inefficiencies Ratio in CPU Usage**

- Front-End Bound
- Memory Bound
- Retiring Bound
- Core Bound

**SLIDE Inefficiencies Ratio in CPU Usage**

- Front-End Bound
- Memory Bound
- Retiring Bound
- Core Bound
# Impact of HugePages

<table>
<thead>
<tr>
<th>Metric</th>
<th>Without Hugepages</th>
<th>With Hugepages</th>
</tr>
</thead>
<tbody>
<tr>
<td>dTLB load miss rate</td>
<td>5.12%</td>
<td>0.25%</td>
</tr>
<tr>
<td>iTLB load miss rate</td>
<td>56.12%</td>
<td>20.96%</td>
</tr>
<tr>
<td>PTW dTLB-miss</td>
<td>7.74%</td>
<td>0.72%</td>
</tr>
<tr>
<td>PTW iTLB-miss</td>
<td>0.02%</td>
<td>0.015%</td>
</tr>
<tr>
<td>RAM read dTLB-miss</td>
<td>3,062,039/s</td>
<td>749,485/s</td>
</tr>
<tr>
<td>RAM read iTLB-miss</td>
<td>12,060/s</td>
<td>11,580/s</td>
</tr>
<tr>
<td>PageFault</td>
<td>32,548/s</td>
<td>26,527/s</td>
</tr>
</tbody>
</table>
Conclusion: From Matrix Multiplication to (few) Hash Lookups

• Standard
  • Operation
    • Matrix Multiply
  • Pros
    • Hardware Support
  • Cons
    • Expensive O(N^3)
    • Can only scale with hardware.
    • Energy

• SLIDE
  • Operations
    • Compute Random Hashes of Data
    • Hash lookups, Sample and Update. (Decades of work in Databases)
    • Very Few Multiplication (100x+ reduction)
  • Pros
    • Energy (IoT), Latency
    • Asynchronous Parallel Gradient updates
    • Simple Hash Tables
    • Larger Network → More Savings
  • Cons
    • Random Memory Access (but parallel SGD)
Future Work

• Distributed SLIDE

• SLIDE on more complex architectures like CNN/RNN
References


Thanks!!!
Welcome to stop by Poster #7