Bandwidth-Efficient Deep Learning
—–from Compression to Acceleration

Song Han
Assistant Professor, EECS
Massachusetts Institute of Technology
AI is Changing Our Lives

Self-Driving Car

Machine Translation

AlphaGo

Smart Robots
Models are Getting Larger

**IMAGE RECOGNITION**

16X
Model

<table>
<thead>
<tr>
<th>Year</th>
<th>Model</th>
<th>Layers</th>
<th>GFLOP</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012</td>
<td>AlexNet</td>
<td>8</td>
<td>1.4 GFLOP</td>
<td>~16%</td>
</tr>
<tr>
<td>2015</td>
<td>ResNet</td>
<td>152</td>
<td>22.6 GFLOP</td>
<td>~3.5%</td>
</tr>
</tbody>
</table>

**SPEECH RECOGNITION**

10X
Training Ops

<table>
<thead>
<tr>
<th>Year</th>
<th>Model</th>
<th>GFLOP</th>
<th>Data</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>Deep Speech 1</td>
<td>80 GFLOP</td>
<td>7,000 hrs</td>
<td>~8%</td>
</tr>
<tr>
<td>2015</td>
<td>Deep Speech 2</td>
<td>465 GFLOP</td>
<td>12,000 hrs</td>
<td>~5%</td>
</tr>
</tbody>
</table>

Dally, NIPS'2016 workshop on Efficient Methods for Deep Neural Networks

Microsoft

Baidu
The first Challenge: Model Size

Hard to distribute large models through over-the-air update
### The Second Challenge: Speed

<table>
<thead>
<tr>
<th>Model</th>
<th>Error Rate</th>
<th>Training Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet18:</td>
<td>10.76%</td>
<td>2.5 days</td>
</tr>
<tr>
<td>ResNet50:</td>
<td>7.02%</td>
<td>5 days</td>
</tr>
<tr>
<td>ResNet101:</td>
<td>6.21%</td>
<td>1 week</td>
</tr>
<tr>
<td>ResNet152:</td>
<td>6.16%</td>
<td>1.5 weeks</td>
</tr>
</tbody>
</table>

Such long training time limits ML researcher’s productivity.

Training time benchmarked with fb.resnet.torch using four M40 GPUs.
The Third Challenge: Energy Efficiency

AlphaGo: 1920 CPUs and 280 GPUs, $3000 electric bill per game

on mobile: drains battery
on data-center: increases TCO
Where is the Energy Consumed?

larger model => more memory reference => more energy
Where is the Energy Consumed?

larger model => more memory reference => more energy

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy [pJ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bit int ADD</td>
<td>0.1</td>
</tr>
<tr>
<td>32 bit float ADD</td>
<td>0.9</td>
</tr>
<tr>
<td>32 bit Register File</td>
<td>1.0</td>
</tr>
<tr>
<td>32 bit int MULT</td>
<td>3.1</td>
</tr>
<tr>
<td>32 bit float MULT</td>
<td>3.7</td>
</tr>
<tr>
<td>32 bit SRAM Cache</td>
<td>5.0</td>
</tr>
<tr>
<td><strong>32 bit DRAM Memory</strong></td>
<td><strong>640</strong></td>
</tr>
</tbody>
</table>

Relative Energy Cost

1 = 1000 ×+
larger model => more memory reference => more energy

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy [pJ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bit int ADD</td>
<td>0.1</td>
</tr>
<tr>
<td>32 bit float ADD</td>
<td>0.9</td>
</tr>
<tr>
<td>32 bit Register File</td>
<td>1</td>
</tr>
<tr>
<td>32 bit int MULT</td>
<td>3.1</td>
</tr>
<tr>
<td>32 bit float MULT</td>
<td>3.7</td>
</tr>
<tr>
<td>32 bit SRAM Cache</td>
<td>5</td>
</tr>
<tr>
<td><strong>32 bit DRAM Memory</strong></td>
<td><strong>640</strong></td>
</tr>
</tbody>
</table>

**Relative Energy Cost**

how to make deep learning more efficient?
Improve the Efficiency of Deep Learning by Algorithm-Hardware Co-Design
Application as a Black Box

Algorithm

Hardware

Spec 2006

CPU
Open the Box before Hardware Design

Breaks the boundary between algorithm and hardware
Proposed Paradigm

Conventional

Training \(\rightarrow\) Inference

Slow \(\rightarrow\) Power-Hungry

Proposed

Regularized Training \(\rightarrow\) Model Compression \(\rightarrow\) Accelerated Inference

Fast \(\rightarrow\) Power-Efficient
How to compress a deep learning model?
Learning both Weights and Connections for Efficient Neural Networks

Han et al.
NIPS 2015
Pruning Neural Networks

before pruning

after pruning

pruning synapses

pruning neurons
Pruning Happens in Human Brain

Pruning AlexNet

**Pruning**

- Pruning AlexNet
- CONV Layer: 3x
  - conv1: 84%
  - conv2: 37%
  - conv3: 34%
  - conv4: 37%
  - conv5: 36%

**Trained Quantization**

- FC Layer: 10x
  - fc1: 10%
  - fc2: 11%
  - fc3: 26%

**Huffman Coding**

- total: 12%

[Han et al. NIPS’15]
Pruning Neural Networks

Pruning
Trained Quantization
Huffman Coding

-0.01x^2 + x + 1

Train Connectivity
Prune Connections
Train Weights

60 Million
6M
10x less connections

[Han et al. NIPS’15]
Pruning Neural Networks

[Han et al. NIPS'15]

Train Connectivity

<table>
<thead>
<tr>
<th>Parameters Pruned Away</th>
<th>Accuracy Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td>0.0%</td>
</tr>
<tr>
<td>90%</td>
<td>0.0%</td>
</tr>
<tr>
<td>80%</td>
<td>0.0%</td>
</tr>
<tr>
<td>70%</td>
<td>0.0%</td>
</tr>
<tr>
<td>60%</td>
<td>0.0%</td>
</tr>
<tr>
<td>50%</td>
<td>0.0%</td>
</tr>
<tr>
<td>40%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Pruning
Trained Quantization
Huffman Coding
Pruning Neural Networks

[Han et al. NIPS’15]

Accuracy Loss

-4.5%
-4.0%
-3.5%
-3.0%
-2.5%
-2.0%
-1.5%
-1.0%
-0.5%
0.0%
0.5%

Parameters Pruned Away

40% 50% 60% 70% 80% 90% 100%

Pruning

Train Connectivity

Prune Connections
Retrain to Recover Accuracy

- Pruning
- Pruning + Retraining

Accuracy Loss vs. Parameters Pruned Away

[Han et al. NIPS’15]

Train Connectivity
Prune Connections
Train Weights
Iteratively Retrain to Recover Accuracy

Accuracy Loss

-4.5% -4.0% -3.5% -3.0% -2.5% -2.0% -1.5% -1.0% -0.5% 0.0% 0.5%

Parameters Pruned Away

40% 50% 60% 70% 80% 90% 100%

[Han et al. NIPS'15]
Pruning RNN and LSTM

*Karpathy et al. "Deep Visual-Semantic Alignments for Generating Image Descriptions"
Pruning RNN and LSTM

[Han et al. NIPS’15]

90%

• **Original**: a basketball player in a white uniform is playing with a **ball**
• **Pruned 90%**: a basketball player in a white uniform is playing with a **basketball**

90%

• **Original**: a brown dog is running through a grassy **field**
• **Pruned 90%**: a brown dog is running through a grassy **area**

90%

• **Original**: a man is riding a surfboard on a wave
• **Pruned 90%**: a man in a wetsuit is riding a wave **on a beach**

95%

• **Original**: a soccer player in red is running in the field
• **Pruned 95%**: a man in a **red shirt and black and white** **black shirt** is running through a field
Exploring the Granularity of Sparsity that is Hardware-friendly

4 types of pruning granularity

irregular sparsity ➞ regular sparsity ➞ more regular sparsity ➞ fully-dense model

[Han et al, NIPS’15] ➞ [Molchanov et al, ICLR’17]
Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding

Han et al.
ICLR 2016
Best Paper
Trained Quantization

2.09, 2.12, 1.92, 1.87

2.0
Trained Quantization

weights (32 bit float)

cluster index (2 bit uint)

centroids

To calculate the compression rate, given \( k \) clusters, we only need \( \log_2(k) \) bits to encode the index. In general, for a network with \( n \) connections and each connection is represented with \( b \) bits, constraining the connections to have only \( k \) shared weights will result in a compression rate of:

\[
r = nb + k b
\]

For example, Figure 3 shows the weights of a single layer neural network with four input units and four output units. There are \( 4 \times 4 = 16 \) weights originally but there are only \( 4 \) shared weights: similar weights are grouped together to share the same value. Originally we need to store 16 weights each
After Trained Quantization: Discrete Weight

[Han et al. ICLR’16]
After Trained Quantization: Discrete Weight after Training

[Han et al. ICLR’16]
How Many Bits do We Need?

[Han et al. ICLR’16]
Table 4.9: Comparison of uniform quantization and non-uniform quantization (this work) with different update methods. -c: updating centroid only; -c+l: update both centroid and label. Baseline ResNet-50 accuracy: 76.15%, 92.87%. All results are after retraining.

<table>
<thead>
<tr>
<th>Quantization Method</th>
<th>1bit</th>
<th>2bit</th>
<th>4bit</th>
<th>6bit</th>
<th>8bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform (Top-1)</td>
<td>-5.33%</td>
<td>9.33%</td>
<td>74.52%</td>
<td>75.49%</td>
<td>76.15%</td>
</tr>
<tr>
<td>Uniform (Top-5)</td>
<td>-8.29%</td>
<td>91.97%</td>
<td>92.60%</td>
<td>92.91%</td>
<td>92.91%</td>
</tr>
<tr>
<td>Non-uniform -c (Top-1)</td>
<td>24.08%</td>
<td>68.41%</td>
<td>76.16%</td>
<td>76.13%</td>
<td>76.20%</td>
</tr>
<tr>
<td>Non-uniform -c (Top-5)</td>
<td>48.57%</td>
<td>88.49%</td>
<td>92.85%</td>
<td>92.88%</td>
<td>92.90%</td>
</tr>
<tr>
<td>Non-uniform -c+l (Top-1)</td>
<td>24.71%</td>
<td>69.36%</td>
<td>76.17%</td>
<td>76.21%</td>
<td>76.19%</td>
</tr>
<tr>
<td>Non-uniform -c+l (Top-5)</td>
<td>49.84%</td>
<td>89.03%</td>
<td>92.87%</td>
<td>92.89%</td>
<td>92.90%</td>
</tr>
</tbody>
</table>

Figure 4.10: Non-uniform quantization performs better than uniform quantization.
More Aggressive Compression: Ternary Quantization

Feed Forward — Back Propagate — Inference Time

Full Precision Weight

Normalized Full Precision Weight

Quantize

Intermediate Ternary Weight

Trained Quantization

Final Ternary Weight

Loss

Ternary Weight Value

Validation

Top1 38%

Top5 12%

Top1 42.8%

Top5 19.8%
## Results: Compression Ratio

<table>
<thead>
<tr>
<th>Network</th>
<th>Original Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
<th>Original Accuracy</th>
<th>Compressed Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeNet-300</td>
<td>1070KB</td>
<td>27KB</td>
<td>40x</td>
<td>98.36%</td>
<td>98.42%</td>
</tr>
<tr>
<td>LeNet-5</td>
<td>1720KB</td>
<td>44KB</td>
<td>39x</td>
<td>99.20%</td>
<td>99.26%</td>
</tr>
<tr>
<td>AlexNet</td>
<td>240MB</td>
<td>6.9MB</td>
<td>35x</td>
<td>80.27%</td>
<td>80.30%</td>
</tr>
<tr>
<td>VGGNet</td>
<td>550MB</td>
<td>11.3MB</td>
<td>49x</td>
<td>88.68%</td>
<td>89.09%</td>
</tr>
<tr>
<td>Inception-V3</td>
<td>91MB</td>
<td>4.2MB</td>
<td>22x</td>
<td>93.56%</td>
<td>93.67%</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>97MB</td>
<td>5.8MB</td>
<td>17x</td>
<td>92.87%</td>
<td>93.04%</td>
</tr>
</tbody>
</table>

Can we make compact models to begin with?

[Han et al. ICLR'16]
SqueezeNet

Iandola et al, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size”, arXiv 2016
## Compressing SqueezeNet

<table>
<thead>
<tr>
<th>Network</th>
<th>Approach</th>
<th>Size</th>
<th>Ratio</th>
<th>Top-1 Accuracy</th>
<th>Top-5 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>-</td>
<td>240MB</td>
<td>1x</td>
<td>57.2%</td>
<td>80.3%</td>
</tr>
<tr>
<td>AlexNet</td>
<td>SVD</td>
<td>48MB</td>
<td>5x</td>
<td>56.0%</td>
<td>79.4%</td>
</tr>
<tr>
<td>AlexNet</td>
<td>Deep Compression</td>
<td>6.9MB</td>
<td>35x</td>
<td>57.2%</td>
<td>80.3%</td>
</tr>
<tr>
<td>SqueezeNet</td>
<td>-</td>
<td>4.8MB</td>
<td>50x</td>
<td>57.5%</td>
<td>80.3%</td>
</tr>
<tr>
<td>SqueezeNet</td>
<td>Deep Compression</td>
<td>0.47MB</td>
<td>510x</td>
<td>57.5%</td>
<td>80.3%</td>
</tr>
</tbody>
</table>
Results: Speedup

Baseline:
mAP = 59.47 / 28.48 / 45.43
FLOP = 17.5G
# Parameters = 6.0M

Pruned:
mAP = 59.30 / 28.33 / 47.72
FLOP = 8.9G
# Parameters = 2.5M

1.6x speedup
Deep Compression Applied to Industry

Compression

Acceleration

Regularization
A Facebook AR prototype backed by Deep Compression

- 8x model size reduction by Deep Compression
- Run on mobile
- Source: Facebook F8’17
EIE: Efficient Inference Engine on Compressed Deep Neural Network

Han et al.
ISCA 2016
Deep Learning Accelerators

- First Wave: Compute (Neu Flow)
- Second Wave: Memory (Diannao family)
- Third Wave: Algorithm / Hardware Co-Design (EIE)

Google TPU: “This unit is designed for dense matrices. Sparse architectural support was omitted for time-to-deploy reasons. Sparsity will have high priority in future designs”
EIE: the First DNN Accelerator for Sparse, Compressed Model

Sparse Weight
90% static sparsity

Sparse Activation
70% dynamic sparsity

Weight Sharing
4-bit weights

$0 \times A = 0$

$W \times 0 = 0$

$2.09, 1.92 = \Rightarrow 2$

10x less computation
3x less computation
5x less memory footprint
8x less memory footprint

[Han et al. ISCA'16]
EIE: Parallelization on Sparsity

\[
\tilde{a} \begin{pmatrix} 0 & a_1 & 0 & a_3 \\ \end{pmatrix} \times \begin{pmatrix}
\begin{bmatrix} w_{0,0} & w_{0,1} & 0 & w_{0,3} \\ 0 & 0 & w_{1,2} & 0 \\ 0 & w_{2,1} & 0 & w_{2,3} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & w_{4,2} & w_{4,3} \\ w_{5,0} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{6,3} \\ 0 & w_{7,1} & 0 & 0
\end{bmatrix}
\end{pmatrix} = \begin{pmatrix}
\begin{bmatrix} b_0 \\ b_1 \\ -b_2 \\ b_3 \\ -b_4 \\ b_5 \\ b_6 \\ -b_7
\end{bmatrix}
\end{pmatrix} \Rightarrow \text{ReLU}
\begin{pmatrix}
\begin{bmatrix} b_0 \\ b_1 \\ 0 \\ b_3 \\ 0 \\ b_5 \\ b_6 \\ 0
\end{bmatrix}
\end{pmatrix}
\]
EIE: Parallelization on Sparsity

\[ \tilde{a} \left( \begin{array}{cccc} 0 & a_1 & 0 & a_3 \\ \end{array} \right) \times \left( \begin{array}{cccc} w_{0,0} & w_{0,1} & 0 & w_{0,3} \\ 0 & 0 & w_{1,2} & 0 \\ 0 & w_{2,1} & 0 & w_{2,3} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & w_{4,2} & w_{4,3} \\ w_{5,0} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{6,3} \\ 0 & w_{7,1} & 0 & 0 \end{array} \right) = \left( \begin{array}{c} b_0 \\ b_1 \\ -b_2 \\ b_3 \\ -b_4 \\ b_5 \\ b_6 \\ -b_7 \end{array} \right) \Rightarrow \left( \begin{array}{c} b_0 \\ b_1 \\ 0 \\ b_3 \\ 0 \\ b_5 \\ b_6 \\ 0 \end{array} \right) \]
Dataflow

\[ \tilde{a} \begin{pmatrix} 0 & a_1 & 0 & a_3 \end{pmatrix} \times \begin{pmatrix} w_{0,0} & w_{0,1} & 0 & w_{0,3} \\ 0 & 0 & w_{1,2} & 0 \\ 0 & w_{2,1} & 0 & w_{2,3} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & w_{4,2} & w_{4,3} \\ w_{5,0} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{6,3} \\ 0 & w_{7,1} & 0 & 0 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ -b_2 \\ b_3 \\ -b_4 \\ b_5 \\ -b_6 \\ -b_7 \end{pmatrix} \]

ReLU \Rightarrow \begin{pmatrix} b_0 \\ b_1 \\ 0 \\ b_3 \\ 0 \\ b_5 \\ b_6 \\ 0 \end{pmatrix}

rule of thumb:
\[ 0 \times A = 0 \quad W \times 0 = 0 \]
EIE Architecture

Weight decode

Address Accumulate

rule of thumb: $0 \times A = 0 \quad W \times 0 = 0 \quad 2.09, 1.92 \Rightarrow 2$
Post Layout Result of EIE

1. Post layout result
2. Throughput measured on AlexNet FC-7
The compressed DNN model is produced as described in the benchmark, we used cuBLAS GEMV to implement the sparsity, this requires a dense DNN accelerator 3TOP/s to fill a 23,961 (3.76%) filler cell and the activation: codebook lookup and address accumulation (in parallel), shift and add, and output activation write. Activation registers) while ActRW selects an SRAM row, and the low v = b(0...p) selects an SRAM column. The arithmetic unit receives a p = = 0...NT (in 4-bit encoded form, it is first expanded to a 16-bit fixed-point number via a table look up. A bypass path is provided in the 16-bit fixed-point number via a table look up. A bypass path is provided in the compressed sparse model. CPU socket and DRAM power are as reported by the NTLSTM NT-Wd VGG-7 Alex-8 Layer 1x 5x 16x 0.3x 18x 9x 24x 210x 10x 78x 8x 1x 507x 248x 507x 1x 6x 14 9x 24 9 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9x 6x 14 9
The compressed DNN model is produced as described in components [26], [27] to report the AP+DRAM power for the class processor, that has been used in NVIDIA Digits Deep Learning. We use the cuSPARSE CSRMV kernel, which is optimized for sparse matrices. We store the sparse matrix in CSR format, and use the original dense layer for the compressed sparse layer. For the compressed sparse model, we use a state-of-the-art GPU for deep learning as our baseline. We use the original dense model and MKL SPBLAS CSRMV for the uncompressed DNN model.

Figure 7. Energy efficiency of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

- CPU: 1x
- CPU Dense (Baseline): 3x
- CPU Compressed: 248x
- GPU Dense: 15%
- GPU Compressed: 10%
- mGPU Dense: 10%
- mGPU Compressed: 37.5%
- EIE: 11%

Table III shows the speedup of different models. We compare EIE with three different models: AlexNet (Alex-7), VGG-16, and LSTM.

- Alex-6: 34,522x
- Alex-7: 61,533x
- Alex-8: 14,826x
- VGG-6: 78x
- VGG-7: 101x
- VGG-8: 61x
- NT-We: 15x
- NT-Wd: 15x
- NT-LSTM: 204x

Geo Mean: 24,207x

Figure 5 shows the layout (after place-and-route) of one PE in EIE under TSMC 45nm process. The PE contains a central control unit, 64 activation registers, and a 2KB activation SRAM. The activation register file holds 64 16-bit activations. Each activation register file is divided into two sets of 32 activations. There are two modes in the central control unit: broadcast mode and local non-zero mode.

To measure the area, power and critical path delay, we used Cacti [25] to get SRAM area and compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler (ICC). We used Cacti [25] to get SRAM area and C++ code for the CACTI compiler ( ICC ).
Comparison: Throughput

Throughput (Layers/s in log scale)

- Core-i7 5930k 22nm CPU
- TitanX 28nm GPU
- Tegra K1 28nm mGPU
- A-Eye 28nm FPGA
- DaDianNao 28nm ASIC
- TrueNorth 28nm ASIC
- EIE 45nm ASIC 64PEs
- EIE 28nm ASIC 256PEs

Compression
Acceleration
Regularization

[Han et al. ISCA'16]
Comparison: Energy Efficiency

Energy Efficiency (Layers/J in log scale)

- CPU
- GPU
- mGPU
- FPGA
- ASIC (DaDianNao, TrueNorth, EIE)

[Han et al. ISCA'16]
Feedforward => Recurrent neural network?
ESE Architecture

[Han et al. FPGA'17]
ESE is now on AWS Marketplace with Xilinx VU9P FPGA
ESE is now on AWS Marketplace with Xilinx VU9P FPGA

### End-to-End Automatic Speech Recognition

**Preprocess**
- **Feature Extraction**

**Recognition**
- **Decoder with Neural Network**
- **Language Model (not necessary)**

**Speech Files (*.wav)**

**Character Labels**

$i_t = g(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$

$f_t = g(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$

$o_t = g(W_{ox}x_t + W_{oh}h_{t-1} + b_o)$

$c_{in_t} = \tanh(W_{cx}x_t + W_{ch}h_{t-1} + b_c)$

$c_t = f_t \cdot c_{in_t} + i_t \cdot c_{in_t}$

$h_t = o_t \cdot \tanh(c_t)$

**Abilities:**

We already have a demo based on LibriSpeech 1000h dataset under Baidu DeepSpeech2 framework, which could show the entire flow of our algorithm, software and hardware co-design (containing pruning, quantization, compilation and FPGA inference);
ESE is now on AWS Marketplace with Xilinx VU9P FPGA

![ASR System Diagram]

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>Only CPU on AWS</th>
<th>CPU + FPGA on AWS</th>
<th>CPU + GPU P4 locally (cudnn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decoding</td>
<td>0.09</td>
<td>0.08</td>
<td>0.40</td>
</tr>
<tr>
<td>Softmax</td>
<td>0.07</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>Fully connected</td>
<td>0.11</td>
<td>0.73</td>
<td>0.12</td>
</tr>
<tr>
<td>LSTM</td>
<td>118.31</td>
<td>16.97</td>
<td>38.58</td>
</tr>
<tr>
<td>CNN</td>
<td>14.16</td>
<td></td>
<td>1.21</td>
</tr>
<tr>
<td>Feature extraction</td>
<td>2.78</td>
<td>2.73</td>
<td>1.95</td>
</tr>
<tr>
<td><strong>E2E time</strong></td>
<td><strong>135.61</strong></td>
<td><strong>20.59</strong></td>
<td><strong>42.35</strong></td>
</tr>
</tbody>
</table>

while Figure 1 and Algorithm 1. The progression of weight distribution is plotted in Figure 2.

Our DSD training employs a three-step process: dense, sparse, dense. Each step is illustrated in

found consistent performance gains over its comparable counterpart for image classification, image

model with sparsity-constrained optimization, and finally increases the model capacity by restoring

novel training strategy that starts from a dense model from conventional training, then regularizes

original size should have the capacity to achieve higher accuracy. This shows the inadequacy of

the same accuracy as the redundant uncompressed model, one hypothesis is that the model of the

size by 35x-49x or more without hurting prediction accuracy. Compression without losing accuracy

under-fitting and a high bias. Bias and variance are hard to optimize at the same time.

In contrast, simply reducing the model capacity would lead to the other extreme, causing a machine

goto

eend

S

eend

Output :

Initialization:

$W_0 = \text{initialize the mask by sorting and keeping the Top-k weights.}$

$\text{Sparse Phase}$

$W_t^{(t+1)} = W_t^{(t)} - \lambda \eta \nabla_{W_t^{(t)}} f(W_t^{(t)} | \mathbf{X}, \mathbf{y})$

$\text{Dense Pruning}$

$S_{t+1} = \text{setting the control registers.}$

$\text{Sparsity Constraint}$

$W_t^{(t)} = \text{end}$

$\text{Sparse}$

$W_t^{(t)} = \text{not converged}$

$\text{Increase Model Capacity Re-Dense}$

$W_t^{(t)} = \text{not converged}$

$\text{Dense}$

$W_t^{(t)} = \text{not converged}$

$\text{Dense}$

$W_0 = \text{under review as a conference paper at ICLR 2017}$

Thank you!

songhan@mit.edu