Accelerating Inference at the Edge

Song Han
Assistant Professor
Massachusetts Institute of Technology

77 Massachusetts Avenue, 38-344
Cambridge, MA, 02139
https://songhan.mit.edu
MIT HAN’s Lab

- **H**: High performance, High energy efficiency Hardware
- **A**: Architectures and Accelerators for Artificial Intelligence
- **N**: Novel algorithms for Neural Networks
- **S**: Small models, Scalable Systems, and Specialized Silicon

(for inference)  (for training)

Computer Vision  Self-driving  Machine Translation  Speech Recognition

FPGA  Mobile SoC  GPU  TPU
The Virtuous Cycle of Deep Learning & Hardware

Users

Generate More

Efficient Hardware Speeds up this Virtuous Cycle

Attract More

Bigger Model

Better Accuracy

Training Data

Training Hardware

Neural Network Models CNN, RNN, LSTM...

Inference Data

Inference Hardware

Song Han, Stanford PhD thesis. https://purl.stanford.edu/qf934gh3708
• In the **post-ImageNet** era, ImageNet is becoming MNIST; we are solving more complicated AI problems using larger data sets driving the demand for more computation.

• However, we are in the **post-Moore’s Law** world where the amount of computation per unit cost and power is no longer increasing at its historic rate.
Evolution of NN Accelerators

- Computation Specialization
- Memory-centric Specialization
- SW/HW Co-design: Compression before Acceleration
- Emerging NN/Accelerator Architectures
Evolution of NN Accelerators

Computation Specialization
Part 1/4: Computation Specialization
Part 1/4: Computation Specialization

- NeuFlow [2011]
- TPU-v1 [2015]
Part 1/4: Computation Specialization

NeuFlow [Farabet, 2011]

- Specialization: Customized logic to efficiently map convolution to hardware with more parallelism
- Flexibility: runtime reconfigurable bus and operations
Part 1/4: Computation Specialization

NeuFlow [Farabet, 2011]

A Runtime Reconfigurable Dataflow Architecture

Conv

Accumulation
### Part 1/4: Computation Specialization

**NeuFlow [Farabet, 2011]**

<table>
<thead>
<tr>
<th></th>
<th>CPU (Intel DuoCore, 2.7GHz)</th>
<th>GPU (GTX480)</th>
<th>mGPU (GT335m)</th>
<th>NeuFlow on Xilinx Virtex 6</th>
<th>NeuFlow on IBM 45 nm process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real GOPs</td>
<td>1.1</td>
<td>294</td>
<td>54</td>
<td>147</td>
<td>1164</td>
</tr>
<tr>
<td>Power (W)</td>
<td>30</td>
<td>220</td>
<td>30</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>GOPs/W</td>
<td>0.04</td>
<td>1.34</td>
<td>1.8</td>
<td>14.7</td>
<td>230</td>
</tr>
</tbody>
</table>
Part 1/4: Computation Specialization

TPU [Jouppi, production in 2015, published in 2017]

Systolic Array Architecture

- Flexible: CNN/RNN/LSTM/MLP
- Latency matters
- Simple silicon, complicated compiler

Jouppi et al, TPU: In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA’17
Part 1/4: Computation Specialization

TPU [Jouppi, production in 2015, published in 2017]

Jouppi et al, TPU: In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA’17
Roof line model

TPU [Jouppi, 2017]

Jouppi et al., TPU: In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA’17
Gap between real FLOP and peak FLOP: Memory Matters

TPU [Jouppi, 2017]

Jouppi et al., TPU: In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA’17
Gap between real FLOP and peak FLOP: Memory Matters

TPU [Jouppi, 2017]

“About 35% of cycles are spent \textit{waiting for weights to load} from memory into the matrix unit …”
Evolution of NN Accelerators

Computation Specialization
Evolution of NN Accelerators

Memory-centric Specialization
Part 2/4: Memory-centric Specialization

More Tiling

Memory-centric Specialization

• Diannao Family [Chen, 2014-2016]
Part 2/4: Memory-centric Specialization

Memory-centric Specialization

- Diannao Family [Chen, 2014-2016]
- Eyeriss [Chen, 2016]

More Tiling

Better Dataflow
Part 2/4: Memory-centric Specialization

Memory-centric Specialization

- Diannao Family [Chen, 2014-2016]
- Eyeriss [Chen, 2016]
- TPU v2 [Google, 2017]
- Volta GPU [NVIDIA, 2017]
Part 2/4: Memory-centric Specialization

Memory-centric Specialization

- Diannao Family [Chen, 2014-2016]
- Eyeriss [Chen, 2016]
- TPU v2 [Google, 2017]
- Volta GPU [NVIDIA, 2017]
- Brainwave Project [Microsoft, 2017]
The spirit of the NFU is to reflect the decomposition of a layer into computational blocks of inputs/synapses and output neurons ...
Part 2/4: Memory-centric Specialization

Diannao Family [Chen, 2014-2016]

<table>
<thead>
<tr>
<th></th>
<th>Process (nm)</th>
<th>Peak Performance (GOPs)</th>
<th>Peak Power (W)</th>
<th>Area (mm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DianNao</td>
<td>65</td>
<td>452</td>
<td>0.485</td>
<td>3.02</td>
</tr>
<tr>
<td>DaDianNao</td>
<td>28</td>
<td>5585</td>
<td>15.97</td>
<td>67.73</td>
</tr>
<tr>
<td>ShiDianNao</td>
<td>65</td>
<td>194</td>
<td>0.32</td>
<td>4.86</td>
</tr>
</tbody>
</table>

Du, Zidong, et al. “ShiDianNao: Shifting vision processing closer to the sensor.”
Part 2/4: Memory-centric Specialization

Eyeriss [Chen, 2016]

Yu-Hsin Chen, Tushar Krishna, Joel Emmer, Vivienne Sze.
“Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks.”
Part 2/4: Memory-centric Specialization

Eyeriss [Chen, 2016]

Yu-Hsin Chen, Tushar Krishna, Joel Emmer, Vivienne Sze.  
"Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks."
Part 2/4: Memory-centric Specialization

High Bandwidth Memory

High Bandwidth Memory, Instruction Specialization

All On-Chip
Evolution of NN Accelerators

CHAPTER 1. INTRODUCTION

Motivation for Model Compression:
First, a smaller model means less overhead when exporting models to clients. Take autonomous driving for example; Tesla periodically copies new models from their servers to customers' cars. Smaller models require less communication in such over-the-air (OTA) updates, making frequent updates more feasible. Another example is the Apple Store: mobile applications above 100 MB will not download until a user connects to Wi-Fi. As a result, a new feature that increases the binary size by 100MB will receive much more scrutiny than one that increases it by 10MB. Thus, putting a large DNN model in a mobile application is infeasible.

The second reason is inference speed. Many mobile scenarios require low-latency, real-time inference, including self-driving cars and AR glasses, where latency is critical to guarantee safety or user experience. A smaller model helps improve the inference speed on such devices: from the computational perspective, smaller DNN models require fewer arithmetic operations and computation cycles; from the memory perspective, smaller DNN models take less memory reference cycles. If the model is small enough it can fit in the on-chip SRAM, which is faster to access than off-chip DRAM memory.

The third reason is energy consumption. Running large neural networks requires significant memory bandwidth to fetch the weights — this consumes considerable energy and is problematic for battery-constrained mobile devices. As a result, iOS 10 requires iPhones to be plugged into chargers while performing photo analysis. Memory access dominates energy consumption. Smaller neural networks require less memory access to fetch the model, saving energy and extending battery life.

The fourth reason is cost. When deploying DNNs on Application-Specific Integrated Circuits (ASICs), a sufficiently small model can be stored on-chip directly. As smaller models require less on-chip SRAM, this permits a smaller ASIC die thus making the chip less expensive.

Smaller deep learning models are also appealing when deployed in large-scale data centers as...
Evolution of NN Accelerators

Efficient Algorithm

co-design

Domain-Specific Hardware

before pruning

pruning synapses

after pruning

Model Compression

Top Down

Hardware
CHAPTER 3. PRUNING DEEP NEURAL NETWORKS

Figure 3.1: Pruning the synapses and neurons of a deep neural network.

The phases of pruning and retraining may be repeated iteratively to further reduce network complexity. In effect, this training process learns the network connectivity in addition to the weights — this parallels the human brain development [109] [110], where excess synapses formed in the first few months of life are gradually "pruned", with neurons losing little-used connections while preserving the functionally important connections.

On the ImageNet dataset, the pruning method reduced the number of parameters of AlexNet by a factor of $9 \times (61$ to $6.7$ million), without incurring accuracy loss. Similar experiments with VGG-16 found that the total number of parameters can be reduced by $13 \times (138$ to $10.3$ million), again with no loss of accuracy. We also experimented with the more efficient fully-convolutional neural networks: GoogleNet (Inception-V1), SqueezeNet, and ResNet-50, which have zero or very thin fully connected layers. From these experiments we find that they share very similar pruning ratios before the accuracy drops: 70% of the parameters in those fully-convolutional neural networks can be pruned. GoogleNet is pruned from 7 million to 2 million parameters, SqueezeNet from 1.2 million to 0.38 million, and ResNet-50 from 25.5 million to 7.47 million, all with no loss of Top-1 and Top-5 accuracy on Imagenet.

In the following sections, we provide solutions on how to prune neural networks and how to retrain the pruned model to recover prediction accuracy. We also demonstrate the speedup and energy efficiency improvements of the pruned model when run on commodity hardware.

3.2 Pruning Methodology

Our pruning method employs a three-step process: training connectivity, pruning connections, and retraining the remaining weights. The last two steps can be done iteratively to obtain better compression ratios. The process is illustrated in Figure 3.2 and Algorithm 1.
In order to understand the 3rd generation of NN accelerator let’s switch gear to talk about deep model compression
CHAPTER 1. INTRODUCTION

Conventional
Training

Proposed
Regularized Training

Model Compression

Inference

Conventional

Proposed

Chapter 5
Han et al. ICLR’17

Chapter 3, 4
Han et al. NIPS’15
Han et al. ICLR’16

Chapter 6
Han et al. ISCA’16
Han et al. FPGA’17

Figure 1.2: Thesis contributions: regularized training, model compression, and accelerated inference.

previous accelerators running the uncompressed model. Previously proposed sparse linear algebra accelerators [30–32] do not address weight sharing, extremely narrow bits, or the activation sparsity, the other benefits of model compression. These factors motivate us to build a specialized hardware accelerator that can operate efficiently on a deeply compressed neural network.

1.2 Contribution and Thesis Outline

We optimize the efficiency of deep learning with a top-down approach from algorithm to hardware.

This thesis proposes the techniques for regularized training ⇒ model compression ⇒ accelerated inference, illustrated in Figure 1.2. The contributions of this thesis are:

• A model compression technique, called Deep Compression, that consists of pruning, trained quantization and variable length coding, which can compress DNN models by 18−49× while fully preserving the prediction accuracy.

• A regularization technique, called Dense-Sparse-Dense (DSD) Training, that can regularize neural network training and prevent overfitting to improve the accuracy for a wide range of CNNs, RNNs, and LSTMs. The DSD model zoo is available online.

• An efficient hardware architecture, called “Efficient Inference Engine” (EIE), that can perform inference on the sparse, compressed DNNs and save a significant amount of memory bandwidth. EIE achieved 13× speed up and 3,400× better energy efficiency than a GPU.

All these techniques center around exploiting the sparsity in neural networks, shown in Figure 1.3.
Optimization Techniques

Loop transformations to minimize memory access*

Pruning

Compression

Winograd, Strassen and FFT

Novel layer types (squeeze, shuffle, shift)

Numerical Representations & Reducing Precision

Deep Compression Part 1: Pruning

before pruning

after pruning

pruning synapses

pruning neurons
Pruning Happens in Human Brain

50 Trillion Synapses

50 Trillion Synapses

1000 Trillion Synapses

500 Trillion Synapses

Newborn

1 year old

Adolescent

In 1979, an analysis of synapse numbers in the frontal lobe at different ages revealed a dramatic decrease in synapses during adolescence. Sleep brainwave amplitudes follow a similar pattern. Furthermore, symptoms of schizophrenia typically emerge in adolescence. In 1983, these facts together led the psychiatrist Irwin Feinberg, who was studying sleep, to propose that defects in adolescent pruning of synapses might be a cause of schizophrenia.

Schizophrenia: a long-term mental disorder of a type involving a breakdown in the relation between thought, emotion, and behavior, leading to faulty perception and inappropriate actions.
Pruning AlexNet

CONV Layer: 3x

FC Layer: 10x

Song Han, Jeff Pool, John Tran, Bill Dally, "Learning both Weights and Connections for Efficient Neural Networks" NIPS '15
Pruning Neural Networks

-0.01x^2 + x + 1

Train Connectivity

Prune Connections

Train Weights

60 Million

6M

10x less connections

Song Han, Jeff Pool, John Tran, Bill Dally, "Learning both Weights and Connections for Efficient Neural Networks" NIPS'15
Pruning Neural Networks

Train Connectivity

Parameters Pruned Away

Accuracy Loss

Song Han, Jeff Pool, John Tran, Bill Dally, "Learning both Weights and Connections for Efficient Neural Networks" NIPS'15
Pruning Neural Networks

Train Connectivity

Prune Connections

Accuracy Loss

Parameters Pruned Away

Pruning

Song Han, Jeff Pool, John Tran, Bill Dally, "Learning both Weights and Connections for Efficient Neural Networks" NIPS'15
Retrain to Recover Accuracy

Song Han, Jeff Pool, John Tran, Bill Dally, "Learning both Weights and Connections for Efficient Neural Networks" NIPS'15
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%

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

Pruning Pruning+Retraining Iterative Pruning and Retraining

Train Connectivity
Prune Connections
Train Weights

Song Han, Jeff Pool, John Tran, Bill Dally, "Learning both Weights and Connections for Efficient Neural Networks" NIPS'15
Pruning RNN and LSTM

Explain Images with Multimodal Recurrent Neural Networks, Mao et al.
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei
Show and Tell: A Neural Image Caption Generator, Vinyals et al.
Long-term Recurrent Convolutional Networks for Visual Recognition and Description, Donahue et al.
Learning a Recurrent Visual Representation for Image Caption Generation, Chen and Zitnick

*Karpathy et al "Deep Visual-Semantic Alignments for Generating Image Descriptions"

Song Han, Jeff Pool, John Tran, Bill Dally, "Learning both Weights and Connections for Efficient Neural Networks" NIPS’15
Pruning RNN and LSTM

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
Pruning Helps Transfer Learning

ImageNet

Dense Top1 Accuracy
Sparse Top1 Accuracy

69.0% 69.0%
Dense Sparse

45
Pruning Helps Transfer Learning

Dense Top1 Accuracy  Sparse Top1 Accuracy

ImageNet

Dense 69.0%  Sparse 69.0%

Flower102

Dense 84.3%  Sparse 88.3%

Transfer learning
Winograd Native Pruned Network:

really associated with the previous layer, we perform this transformed ReLU starting with the second equation.

\[
S = A^T [GgG^T] \odot [B^T dB] A
\]

\[
S = A^T [G\text{Prune}(g)G^T] \odot [B^T \text{ReLU}(d)B] A
\]

\[
S = A^T [\text{Prune}(GgG^T)] \odot [B^T \text{ReLU}(d)B] A
\]

\[
S = A^T [\text{Prune}(GgG^T)] \odot [\text{ReLU}(B^T dB)] A
\]

---

Xingyu Liu, Jeff Pool, Song Han, Bill Dally

Efficient Sparse-Winograd Convolutional Neural Networks, ICLR’18
Pruning in Winograd Domain

Figure 4: Top-1 and top-5 validation accuracy vs density for three models on a variation of ResNet-18.

- **Top-1 Test Accuracy**
  - Weight Density (%) vs Top-1 Test Accuracy for Spatial Pruning, Winograd, Winograd-ReLU, and Native Pruning.
  - Top-1 accuracies for three models are compared across different weight densities.

- **Top-5 Test Accuracy**
  - Weight Density (%) vs Top-5 Test Accuracy for Spatial Pruning, Winograd, Winograd-ReLU, and Native Pruning.
  - Top-5 accuracies for three models are compared across different weight densities.

The graphs illustrate the performance of the models with varying weight densities, showing the effectiveness of pruning in the Winograd domain. The Winograd-ReLU pruning approach consistently outperforms the other methods, achieving higher accuracies with lower weight densities compared to the baseline models.
Figure 5: Kernels of ResNet-18 Winograd-ReLU model res2a_2a layer with density of 100% (left, 87.43% top-5 accuracy), 35% (middle, 87.36% top-5 accuracy) and 15% (right, 86.57% top-5 accuracy). Positive, negative and pruned weights are in red, blue and black respectively.
Structured Sparsity

![Structured Sparsity Diagram](image)

**Figure 5:** Structured pruning at different granularity levels.

---

Huizi Mao, Song Han, Jeff Pool, Xingyu Liu, Yu Wang, Bill Dally
Exploring the Regularity of Sparse Structure in Convolutional Neural Networks
Pruning Tool is Available at DeePhi

Deep Neural Network Development Kit (DNNDK)

http://www.deephi.com/dnndk.html
DeePhi’s Pruning Tool

- 4 commands in decent_p
  - ana – analyze the network
  - prune – prune the network according to config
  - finetune – finetune the network to recovery accuracy
  - transform – transform the pruned model to regular model
<table>
<thead>
<tr>
<th>Classification Networks</th>
<th>Baseline</th>
<th>Pruning Result 1</th>
<th>Pruning Result 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Top-5</td>
<td>ΔTop5</td>
</tr>
<tr>
<td>Resnet50 [7.7G]</td>
<td></td>
<td>91.65%</td>
<td>91.23%</td>
</tr>
<tr>
<td>Inception_v2 [4.0G]</td>
<td></td>
<td>91.07%</td>
<td>90.37%</td>
</tr>
<tr>
<td>SqueezeNet [778M]</td>
<td></td>
<td>83.19%</td>
<td>82.46%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Detection Networks</th>
<th>Baseline</th>
<th>Pruning Result 1</th>
<th>Pruning Result 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mAP</td>
<td>ΔmAP</td>
</tr>
<tr>
<td>DetectNet [17.5G]</td>
<td>44.46</td>
<td>45.7</td>
<td>+1.24</td>
</tr>
<tr>
<td>SSD+VGG [117G]</td>
<td>61.5</td>
<td>62.0</td>
<td>+0.5</td>
</tr>
<tr>
<td>[A] SSD+VGG [173G]</td>
<td>57.1</td>
<td>58.7</td>
<td>+1.6</td>
</tr>
<tr>
<td>[B] Yolov2 [198G]</td>
<td>80.4</td>
<td>81.9</td>
<td>+1.5</td>
</tr>
</tbody>
</table>
Pruning for Object Detection

SSD+VGG @Deephi Surveillance 4classes

Deep Compression Tool (DECENT)
DECENT Speedup

Pruning Speedup on DPU (SSD)

fps

110
100
90
80
70
60
55
50
40
30
20
18
15
10
0

FLOPS

117G
19G
11.6G

5.7x
4x
2x DPU-4096@ZU9

Deep Compression Tool (DECENT)
### Deep Compression Part2: Trained Quantization

#### Figure 2: Representing the matrix sparsity with relative index. Padding filler zero to prevent overflow.

<table>
<thead>
<tr>
<th>weights (32 bit float)</th>
<th>cluster index (2 bit uint)</th>
<th>centroids</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.09 -0.98 1.48 0.09</td>
<td>3 0 2 1</td>
<td>3: 2.00</td>
</tr>
<tr>
<td>0.05 -0.14 -1.08 2.12</td>
<td>1 1 0 3</td>
<td>2: 1.50</td>
</tr>
<tr>
<td>-0.91 1.92 0 -1.03</td>
<td>0 3 1 0</td>
<td>1: 0.00</td>
</tr>
<tr>
<td>1.87 0 1.53 1.49</td>
<td>3 1 2 2</td>
<td>0: -1.00</td>
</tr>
</tbody>
</table>

#### Figure 3: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom).

We store the sparse structure that results from pruning using compressed sparse row (CSR) or compressed sparse column (CSC) format, which requires \(2a + n + 1\) numbers, where \(a\) is the number of non-zero elements and \(n\) is the number of rows or columns.

To compress further, we store the index difference instead of the absolute position, and encode this difference in 8 bits for conv layer and 5 bits for fc layer. When we need an index difference larger than the bound, we use the zero padding solution shown in Figure 2: in case when the difference exceeds 8, the largest 3-bit (as an example) unsigned number, we add a filler zero.

Network quantization and weight sharing further compresses the pruned network by reducing the number of bits required to represent each weight. We limit the number of effective weights we need to store by having multiple connections share the same weight, and then fine-tune those shared weights. Weight sharing is illustrated in Figure 3. Suppose we have a layer that has 4 input neurons and 4 output neurons, the weight is a \(4 \times 4\) matrix. On the top left is the \(4 \times 4\) weight matrix, and on the bottom left is the \(4 \times 4\) gradient matrix. The weights are quantized to 4 bins (denoted with 4 colors), all the weights in the same bin share the same value, thus for each weight, we then need to store only a small index into a table of shared weights. During update, all the gradients are grouped by the color and summed together, multiplied by the learning rate and subtracted from the shared centroids from last iteration. For pruned AlexNet, we are able to quantize to 8-bits (256 shared weights) for each CONV layers, and 5-bits (32 shared weights) for each FC layer without any loss of accuracy.

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:

\[
r = nb - n\log_2(k) + kb
\]

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
We store the sparse structure that results from pruning using compressed sparse row (CSR) or compressed sparse column (CSC) format, which requires $2a + n + 1$ numbers, where $a$ is the number of non-zero elements and $n$ is the number of rows or columns.

To compress further, we store the index difference instead of the absolute position, and encode this difference in 8 bits for conv layer and 5 bits for fc layer. When we need an index difference larger than the bound, we use the zero padding solution shown in Figure 2: in case when the difference exceeds the largest 3-bit (as an example) unsigned number, we add a filler zero.

Weight sharing is illustrated in Figure 3. Suppose we have a layer that has 4 input neurons and 4 output neurons, the weight is a $4 \times 4$ matrix. On the top left is the $4 \times 4$ weight matrix, and on the bottom left is the $4 \times 4$ gradient matrix. The weights are quantized to 4 bins (denoted with 4 colors), all the weights in the same bin share the same value, thus for each weight, we then need to store only a small index into a table of shared weights. During update, all the gradients are grouped by the color and summed together, multiplied by the learning rate and subtracted from the shared centroids from last iteration. For pruned AlexNet, we are able to quantize to 8-bits (256 shared weights) for each CONV layers, and 5-bits (32 shared weights) for each FC layer without any loss of accuracy.

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:

$$r = nb + k$$

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...
Uniform vs non-Uniform Quantization

Different from HashNet (Chen et al., 2015) where weight sharing is determined by a hash function before the networks sees any training data, our method determines weight sharing after a network is fully trained, so that the shared weights approximate the original network.

**3.1 Weight Sharing**

We use k-means clustering to identify the shared weights for each layer of a trained network, so that all the weights that fall into the same cluster will share the same weight. Weights are not shared across layers. We partition $n$ original weights $W = \{w_1, w_2, \ldots, w_n\}$ into $k$ clusters $C = \{c_1, c_2, \ldots, c_k\}$, so as to minimize the within-cluster sum of squares (WCSS):

$$
\arg \min_C \sum_{i=1}^k \sum_{w \in c_i} |w|^2
$$

(L2)

Different from HashNet (Chen et al., 2015) where weight sharing is determined by a hash function before the networks sees any training data, our method determines weight sharing after a network is fully trained, so that the shared weights approximate the original network.

**3.2 Initialization of Shared Weights**

Centroid initialization impacts the quality of clustering and thus affects the network's prediction accuracy. We examine three initialization methods: Forgy (random), density-based, and linear initialization. In Figure 4 we plotted the original weights' distribution of conv3 layer in AlexNet (CDF in blue, PDF in red). The weights forms a bimodal distribution after network pruning. On the bottom it plots the effective weights (centroids) with 3 different initialization methods (shown in blue, red and yellow). In this example, there are 13 clusters.

- **Forgy (random)** initialization randomly chooses $k$ observations from the data set and uses these as the initial centroids. The initialized centroids are shown in yellow. Since there are two peaks in the bimodal distribution, Forgy method tend to concentrate around those two peaks.

- **Density-based** initialization linearly spaces the CDF of the weights in the y-axis, then finds the horizontal intersection with the CDF, and finally finds the vertical intersection on the x-axis, which becomes a centroid, as shown in blue dots. This method makes the centroids denser around the two peaks, but more scattered than the Forgy method.

- **Linear** initialization linearly spaces the centroids between the $[\min, \max]$ of the original weights. This initialization method is invariant to the distribution of the weights and is the most scattered compared with the former two methods.

Larger weights play a more important role than smaller weights (Han et al., 2015), but there are fewer of these large weights. Thus for both Forgy initialization and density-based initialization, very few centroids have large absolute value which results in poor representation of these few large weights.

Linear initialization does not suffer from this problem. The experiment section compares the accuracy.
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</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</td>
<td>2.39%</td>
<td>91.97%</td>
<td>92.60%</td>
<td>92.87%</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>92.91%</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>

Non-uniform quantization performs better than uniform quantization. For uniform quantization, however, all the layers of the baseline ResNet-50 can be compressed to 8 bits without losing accuracy (at 4 bits, there are about 1.6% top-1 accuracy loss when using uniform quantization). The advantage of non-uniform quantization is that it can better capture the non-uniform distribution of the weights. When the probability distribution is higher, the distance between each centroid would be closer. However, uniform quantization cannot achieve this.

Table 4.9 compares the performance of two non-uniform quantization strategies. During fine-tuning, one strategy is to only update the centroid; the other strategy is to update both the centroid and the label (the label means which centroid does the weight belong to). Intuitively, the latter case has more degree of freedom in the learning process and should give better performance. However, experiments show that the improvement is not significant, as shown in the third row and the fourth row in Table 4.9.
DeePhi’s Quantization Tool

- 4 commands in decent_q
  - quantize – quantize network
  - test – test network accuracy/mAP
  - finetune – finetune quantized network
  - deploy – generate model for DPU

- Data
  - Calibration data – quantize activation
  - Training data – further increase accuracy
DECENT Quantization Result

- Uniform quantization
  - 8bit for both weights and activation
  - A small set of images for calibration

<table>
<thead>
<tr>
<th>Networks</th>
<th>float32 baseline</th>
<th>8-bit Quantization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top1</td>
<td>Top5</td>
</tr>
<tr>
<td>Inception_v1</td>
<td>66.90%</td>
<td>87.68%</td>
</tr>
<tr>
<td>Inception_v2</td>
<td>72.78%</td>
<td>91.04%</td>
</tr>
<tr>
<td>Inception_v3</td>
<td>77.01%</td>
<td>93.29%</td>
</tr>
<tr>
<td>Inception_v4</td>
<td>79.74%</td>
<td>94.80%</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>74.76%</td>
<td>92.09%</td>
</tr>
<tr>
<td>VGG16</td>
<td>70.97%</td>
<td>89.85%</td>
</tr>
<tr>
<td>Inception-ResNet-v2</td>
<td>79.95%</td>
<td>95.13%</td>
</tr>
</tbody>
</table>

Deep Compression Tool (DECENT)
Workflow of Deep Compression Tool (DECENT)
## 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>
Pruning Accelerates Image Classification on FPGA

100% Model: 33.7 ms [3/3]  
"recreational vehicle, RV, R.V."

Before Compression  
30 FPS

40% Model: 16.2 ms [3/3]  
"recreational vehicle, RV, R.V."

After 2.5X Compression  
62 FPS
Pruning Accelerates Object Detection on FPGA

Before Compression
16FPS@120GOP

After 10X Compression
125FPS@11.5GOP
Discovering the Philosophy behind Deep Learning Computing
Now we switch gear to the 3rd generation of NN accelerator: Accelerating the Compressed, Sparse NN Models

Chapter 1. Introduction

Model Compression

Accelerated Inference

Regularized Training

Chapter 5
Han et al. ICLR’17

Chapter 3, 4
Han et al. NIPS’15
Han et al. ICLR’16

Chapter 6
Han et al. ISCA’16
Han et al. FPGA’17

Efficient Methods and Hardware for Deep Learning

Song Han, Stanford PhD thesis.
Part 3/4: Accelerating Compressed Model

Figure 3.1: Pruning the synapses and neurons of a deep neural network.

In effect, this training process learns the network connectivity in addition to the weights — this parallels the human brain development [109] [110], where excess synapses formed in the first few months of life are gradually “pruned”, with neurons losing little-used connections while preserving the functionally important connections.

On the ImageNet dataset, the pruning method reduced the number of parameters of AlexNet by a factor of $9 \times 61$ to 6.7 million, without incurring accuracy loss. Similar experiments with VGG-16 found that the total number of parameters can be reduced by $13 \times 138$ to 10.3 million, again with no loss of accuracy. We also experimented with the more efficient fully-convolutional neural networks: GoogleNet (Inception-V1), SqueezeNet, and ResNet-50, which have zero or very thin fully connected layers. From these experiments we find that they share very similar pruning ratios before the accuracy drops: 70% of the parameters in those fully-convolutional neural networks can be pruned. GoogleNet is pruned from 7 million to 2 million parameters, SqueezeNet from 1.2 million to 0.38 million, and ResNet-50 from 25.5 million to 7.47 million, all with no loss of Top-1 and Top-5 accuracy on ImageNet.

In the following sections, we provide solutions on how to prune neural networks and how to retrain the pruned model to recover prediction accuracy. We also demonstrate the speedup and energy efficiency improvements of the pruned model when run on commodity hardware.

3.2 Pruning Methodology

Our pruning method employs a three-step process: training connectivity, pruning connections, and retraining the remaining weights. The last two steps can be done iteratively to obtain better compression ratios. The process is illustrated in Figure 3.2 and Algorithm 1.

SW/HW co-design: Compression before Acceleration

- EIE [Han, 2016]
- ESE [Han, 2017]
- SCNN [Parashar, 2017]
- DLA [NVIDIA, 2017]
Part 3/4: Accelerating Compressed Model

EIE [Han, 2016]

\[ \vec{a} \begin{bmatrix} 0 & 0 & a_2 & 0 & a_4 & a_5 & 0 & a_7 \end{bmatrix} \]

\[ \times \]

\[ \begin{bmatrix} w_{0,0} & 0 & w_{0,2} & 0 & w_{0,4} & w_{0,5} & 0 & w_{0,6} \\ 0 & w_{1,1} & 0 & w_{1,3} & 0 & w_{1,4} & 0 & w_{1,6} \\ 0 & 0 & w_{2,2} & 0 & w_{2,4} & 0 & w_{2,7} \\ 0 & w_{3,1} & 0 & 0 & w_{3,5} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{4,1} & 0 & w_{4,4} & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{5,4} & 0 & 0 & w_{5,7} \\ 0 & 0 & 0 & 0 & 0 & w_{6,4} & 0 & w_{6,6} \\ 0 & 0 & 0 & 0 & 0 & 0 & w_{7,4} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & w_{8,7} \\ w_{9,0} & 0 & 0 & 0 & 0 & 0 & 0 & w_{9,6} \\ 0 & 0 & 0 & 0 & 0 & w_{10,4} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & w_{11,4} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & w_{12,7} \\ w_{12,0} & 0 & w_{12,2} & 0 & 0 & w_{12,5} & 0 & w_{12,7} \\ w_{13,0} & 0 & w_{13,2} & 0 & 0 & 0 & w_{13,6} & 0 \\ 0 & 0 & 0 & 0 & w_{14,2} & w_{14,4} & w_{14,5} & 0 \\ 0 & 0 & 0 & 0 & w_{15,2} & w_{15,3} & 0 & w_{15,5} \end{bmatrix} \]

\[ = \]

\[ \begin{bmatrix} b_0 \\ b_1 \\ -b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ -b_7 \\ b_8 \\ b_9 \\ -b_{10} \\ b_{10} \\ b_{11} \\ -b_{12} \\ b_{12} \\ b_{13} \\ b_{14} \end{bmatrix} \]

\[ \Rightarrow \]

ReLU

Save Memory by Deep Compression
10x-30x BW saving

Virtual Weight

<table>
<thead>
<tr>
<th>W_{a1}</th>
<th>W_{a2}</th>
<th>W_{a3}</th>
<th>W_{a4}</th>
<th>W_{a5}</th>
<th>W_{a6}</th>
<th>W_{a7}</th>
<th>W_{a8}</th>
<th>W_{a9}</th>
<th>W_{a10}</th>
<th>W_{a11}</th>
<th>W_{a12}</th>
<th>W_{a13}</th>
<th>W_{a14}</th>
<th>W_{a15}</th>
<th>W_{a16}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Relative Row Index

| 0 | 3 | 4 | 6 | 8 | 10 | 11 | 13 |

Column Pointer

| 0 | 3 | 4 | 6 | 8 | 10 | 11 | 13 |

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, Bill Dally
“EIE: Efficient Inference Engine on Compressed Deep Neural Network”, ISCA’16
Part 3/4: Accelerating Compressed Model

EIE [Han, 2016]

Save Memory by Deep Compression saving 10x-30x BW

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, Bill Dally
“EIE: Efficient Inference Engine on Compressed Deep Neural Network”, ISCA’16
Part 3/4: Accelerating Compressed Model

EIE [Han, 2016]

EIE introduces methods to distribute the storage and the computation across PEs. EIE exploits the dynamic sparsity of activations to save computation and better energy savings compared to accessing from a memory. EIE is the first accelerator for sparse and compressed deep neural networks. EIE is 24,000 times more efficient than a CPU and GPU, respectively. The contributions of EIE include:

1. **Sparse Activation:** Compressed DNN Model
2. **Sparse Weight:** Compressed DNN Model
3. **Coded Weights:** Compressed DNN Model
4. **Leading Zero Detect:** Compressed DNN Model
5. **Arithmetic Unit:** Compressed DNN Model
6. **Rounding:** Compressed DNN Model
7. **SRAM Bank:** Compressed DNN Model
8. **Sparse Matrix Access:** Compressed DNN Model
9. **Sparse Matrix:** Compressed DNN Model
10. **Act Value:** Compressed DNN Model

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, Bill Dally
“EIE: Efficient Inference Engine on Compressed Deep Neural Network”, ISCA’16
Part 3/4: Accelerating Compressed Model

EIE [Han, 2016]
Part 3/4: Accelerating Compressed Model

EIE [Han, 2016]

We use NVIDIA GeForce GTX Titan X GPU, which has been used in NVIDIA Digits Deep Learning benchmark. AlexNet, VGGNet, and NeuralTalk model zoos; The compressed sparse model. CPU socket and DRAM power consumption, that has been used in NVIDIA Digits Deep Learning benchmark. AlexNet, VGGNet, and NeuralTalk model zoos are hierarchically distributed to each PE. To take advantage of the input vector sparsity, we use leading non-zero detection logic to select the first non-zero result. Each activation is broadcast back to all the PEs via a separate activation register file that accommodates the entries fetched on each SRAM read. The high 13 bits of this PE's slice of column vector contain one 4-bit element of the sparse matrix while any) of this PE's slice of column vector has a length greater than 4K, the M

3) Mobile GPU.

2) GPU.

1) CPU.

Figure 6. Speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

We use NVIDIA GeForce GTX Titan X GPU, which has been used in NVIDIA Digits Deep Learning benchmark. AlexNet, VGGNet, and NeuralTalk model zoos; The compressed sparse model. CPU socket and DRAM power consumption, that has been used in NVIDIA Digits Deep Learning benchmark. AlexNet, VGGNet, and NeuralTalk model zoos are hierarchically distributed to each PE. To take advantage of the input vector sparsity, we use leading non-zero detection logic to select the first non-zero result. Each activation is broadcast back to all the PEs via a separate activation register file that accommodates the entries fetched on each SRAM read. The high 13 bits of this PE's slice of column vector contain one 4-bit element of the sparse matrix while any) of this PE's slice of column vector has a length greater than 4K, the M

3) Mobile GPU.

2) GPU.

1) CPU.

Figure 6. Speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

We use NVIDIA GeForce GTX Titan X GPU, which has been used in NVIDIA Digits Deep Learning benchmark. AlexNet, VGGNet, and NeuralTalk model zoos; The compressed sparse model. CPU socket and DRAM power consumption, that has been used in NVIDIA Digits Deep Learning benchmark. AlexNet, VGGNet, and NeuralTalk model zoos are hierarchically distributed to each PE. To take advantage of the input vector sparsity, we use leading non-zero detection logic to select the first non-zero result. Each activation is broadcast back to all the PEs via a separate activation register file that accommodates the entries fetched on each SRAM read. The high 13 bits of this PE's slice of column vector contain one 4-bit element of the sparse matrix while any) of this PE's slice of column vector has a length greater than 4K, the M

3) Mobile GPU.

2) GPU.

1) CPU.

Figure 6. Speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

We use NVIDIA GeForce GTX Titan X GPU, which has been used in NVIDIA Digits Deep Learning benchmark. AlexNet, VGGNet, and NeuralTalk model zoos; The compressed sparse model. CPU socket and DRAM power consumption, that has been used in NVIDIA Digits Deep Learning benchmark. AlexNet, VGGNet, and NeuralTalk model zoos are hierarchically distributed to each PE. To take advantage of the input vector sparsity, we use leading non-zero detection logic to select the first non-zero result. Each activation is broadcast back to all the PEs via a separate activation register file that accommodates the entries fetched on each SRAM read. The high 13 bits of this PE's slice of column vector contain one 4-bit element of the sparse matrix while any) of this PE's slice of column vector has a length greater than 4K, the M

3) Mobile GPU.

2) GPU.

1) CPU.

Figure 6. Speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

We use NVIDIA GeForce GTX Titan X GPU, which has been used in NVIDIA Digits Deep Learning benchmark. AlexNet, VGGNet, and NeuralTalk model zoos; The compressed sparse model. CPU socket and DRAM power consumption, that has been used in NVIDIA Digits Deep Learning benchmark. AlexNet, VGGNet, and NeuralTalk model zoos are hierarchically distributed to each PE. To take advantage of the input vector sparsity, we use leading non-zero detection logic to select the first non-zero result. Each activation is broadcast back to all the PEs via a separate activation register file that accommodates the entries fetched on each SRAM read. The high 13 bits of this PE's slice of column vector contain one 4-bit element of the sparse matrix while any) of this PE's slice of column vector has a length greater than 4K, the M

3) Mobile GPU.

2) GPU.

1) CPU.

Figure 6. Speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.
Part 3/4: Accelerating Compressed Model

EIE [Han, 2016]

![Speedup and Energy Efficiency Graph]

Geo Mean Over AlexNet, VGG, NeuralTalk

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, Bill Dally
“EIE: Efficient Inference Engine on Compressed Deep Neural Network”, ISCA’16
Part 3/4: Accelerating Compressed Model

ESE [Han, 2017]

Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, Bill Dally, 
ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA, FPGA'17
Part 3/4: Accelerating Compressed Model

ESE [Han, 2017]

DeePhi Descartes Efficient Speech Recognition Engine

DDESE is an efficient end-to-end automatic speech recognition (ASR) engine with the deep learning acceleration solution of algorithm, software and hardware co-design (containing pruning, quantization, compilation and FPGA inference) by DeePhi.

Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, Bill Dally, ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA, FPGA'17
Part 3/4: Accelerating Compressed Model

ESE [Han, 2017]

Part 3/4: Accelerating Compressed Model

ESE [Han, 2017]

Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, Bill Dally, ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA, FPGA'17
Part 3/4: Accelerating Compressed Model

SCNN [Parashar, 2017]

- Only compute where both operands are nonzero

![Diagram showing the computation process for SCNN](image)

Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, Bill Dally

Part 3/4: Accelerating Compressed Model

Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, Bill Dally
Part 3/4: Accelerating Compressed Model

SCNN [Parashar, 2017]

Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, Bill Dally
Part 3/4: Accelerating Compressed Model

DLA [NVIDIA, 2017]

- Weight Sparsity: no wire toggle for 0 weight, save energy
- Winograd Transformation
- Opensourced at http://nvdla.org
Evolution of NN Accelerators

Memory-centric Specialization

SW/HW Co-design: Compression before Acceleration

Computation Specialization
Evolution of NN Accelerators

- Memory-centric Specialization
- SW/HW co-design: Compression before Acceleration
- Computation Specialization
- AlexNet, VGG
Evolution of NN Accelerators

Memory-centric Specialization

Computation Specialization

SW/HW co-design: Compression before Acceleration

GoogleNet, ResNet

AlexNet, VGG
Evolution of NN Accelerators

SW/HW co-design: Compression before Acceleration

Memory-centric Specialization

Computation Specialization

Sparse NN

GoogleNet, ResNet

AlexNet, VGG

The End?
"Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning."

—— Winston Churchill, 1942
CHAPTER 1. INTRODUCTION

given the computation pattern of deep learning and achieved higher efficiency compared with CPUs and GPUs. The first wave of accelerators efficiently implemented the computational primitives for neural networks [16,18,24]. Researchers then realized that memory access is more expensive and critically needs optimization, so the second wave of accelerators efficiently optimized memory transfer and data movement [19–23]. These two generations of accelerators have made promising progress in improving the speed and energy efficiency of running DNNs.

However, both generations of deep learning accelerators treated the algorithm as a black box and focused on only optimizing the hardware architecture. In fact, there is plenty of room at the top by optimizing the algorithm. We found that DNN models can be significantly compressed and simplified before touching the hardware; if we treat these DNN models merely as a black box and hand them directly to hardware, there is massive redundancy in the workload. However, existing hardware accelerators are optimized for uncompressed DNN models, resulting in huge wastes of computation cycles and memory bandwidth compared with running on compressed DNN models. We therefore need to co-design the algorithm and the hardware.

In this dissertation, we co-designed the algorithm and hardware for deep learning to make it run faster and more energy-efficiently. We developed techniques to make the deep learning workload more efficient and compact to begin with and then designed the hardware architecture specialized for the optimized DNN workload. Figure 1.1 illustrates the design methodology of this thesis. Breaking
Evolution of NN Accelerators

Memory-centric Specialization

Computation Specialization

SW/HW co-design: Compression before Acceleration

Sparse NN

GoogleNet, ResNet

AlexNet, VGG
Evolution of NN Accelerators

Emerging NN/Accelerator Architectures and Methodologies
Part 4/4: HW Architecture/NN Architecture Co-design

- Squeezelerator [Kwon, 2018]
- DeePhi’s DPU-v2 [DeePhi, 2018]
- ShiftNet Accelerator [Wu, 2018]

Emerging NN/Accelerator Architectures

SqueezeNext ⟷ Squeezelerator [Kwon, 2018]
Depth-wise Convolution ⟷ DeePhi’s DPU-v2 [DeePhi, 2018]
ShiftNet ⟷ ShiftNet Accelerator [Wu, 2018]
Part 4/4: HW Architecture/NN Architecture Co-design

SqueezeNext, Squeezelerator [Gholami, Kwon, 2018]

Illustration of a ResNet block on the left, a SqueezeNet block in the middle, and a SqueezeNext (SqNxt) block on the right. SqueezeNext uses a two-stage bottleneck module to reduce the number of input channels to the $3 \times 3$ convolution. The latter is further decomposed into separable convolutions to further reduce the number of parameters (orange parts), followed by a $1 \times 1$ expansion module.

The spectrum of accuracy versus energy and inference speed for SqueezeNext, SqueezeNet (v1.0 and v1.1), Tiny DarkNet, and MobileNet. SqueezeNext shows superior performance (in both plots higher and to the left is better).

Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, Kurt Keutzer
SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size
Amir Gholami, Kiseok Kwon, Bichen Wu, Zizheng Tai, Xiangyu Yue, Peter Jin, Sicheng Zhao, Kurt Keutzer
SqueezeNext: Hardware-Aware Neural Network Design
Kiseok Kwon, Alon Amid, Amir Gholami, Bichen Wu, Krste Asanovic, Kurt Keutzer
Co-Design of Deep Neural Nets and Neural Net Accelerators for Embedded Vision Applications, DAC’18
ShiftNet [Wu, 2018]

Figure 1: Illustration of a shift operation followed by a 1x1 convolution. The shift operation adjusts data spatially and the 1x1 convolution mixes information across channels.

Challenge:
Zero FLOP for 3x3, but increased #param and #FLOP for 1x1
=> there’s no free lunch, but ShiftNet opened up a much larger design space
Part 4/4: HW Architecture/NN Architecture Co-design

DPU [DeePhi, 2018]

(a) DeePhi’s DPU Architecture for MobileNet

(b) Fused DP-convolution and PW-convolution

(c) Result on Xilinx ZU9 FPGA: increased the utilization from 28.6% to 45.8%
Summary: Evolution of NN Accelerators

- Computation Specialization
  - NeuFlow [2011]
  - TPU-v1 [2015]

- Memory-centric Specialization
  - Diannao [Chen, 2014]
  - Eyeriss [Chen, 2016]
  - TPU v2 [Google, 2017]
  - Volta GPU [NVIDIA, 2017]
  - Brainwave [Microsoft, 2017]

- SW/HW Co-design: Compression before Acceleration
  - EIE [Han, 2016]
  - ESE [Han, 2017]
  - SCNN [Parashar, 2017]
  - DLA [NVIDIA, 2017]

- Emerging NN/ Accelerator Architectures
  - Squeezelerator [Kwon, 2018]
  - DPU-V2 [DeePhi, 2018]
  - ShiftNet [Wu, 2018]
Outlook: Design Automation for Neural Networks

Customers

Engineers
Model Compression by Human:
Labor Consuming, Sub-optimal

Model Compression by AI:
Automated, Higher Compression Rate, Faster

Agent: DDPG

Reward = -Error * log(FLOP)

Environment: Channel Pruning

Layer t-1
Layer t
Layer t+1

Fig. 1. Overview of AutoML for Model Compression (AMC) engine. Left: AMC replaces human and makes model compression fully automated while performing better than human. Right: Form AMC as a reinforcement learning problem. We process a pretrained network (e.g., MobileNet-V1) in a layer-by-layer manner. Our reinforcement learning agent (DDPG) receives the embedding $s_t$ from a layer $t$, and outputs a sparsity ratio $a_t$. After the layer is compressed with $a_t$, the model moves to the next layer $L_{t+1}$. The accuracy of the pruned model with all layers compressed is evaluated. Finally, as a function of accuracy and FLOP, reward $R$ is returned to the reinforcement learning agent.

We observe that the accuracy of the compressed model is very sensitive to the sparsity of each layer, requiring a fine-grained action space. Therefore, instead of searching over a discrete space, we come up with a continuous compression ratio control strategy with a DDPG agent to learn through trials and errors: penalizing accuracy loss while encouraging model shrinking and speedup. The actor-critic structure also helps to reduce variance, facilitating stabler training. Specifically, our DDPG agent processes the network in a layer-wise manner. For each layer $L_t$, the agent receives a layer embedding $s_t$ which encodes useful characteristics of this layer, and then it outputs a precise compression ratio $a_t$. After layer $L_t$ is compressed with $a_t$, the agent moves to the next layer $L_{t+1}$. The validation accuracy of the pruned model with all layers compressed is evaluated without fine-tuning, which is an efficient delegate of the fine-tuned accuracy. This simple approximation can improve the search time not having to retrain the model, and provide high quality search result. After the policy search is done, the best-explored model is fine-tuned to achieve the best performance.
Outlook: Design Automation for Neural Networks

AMC: AutoML for Model Compression

We discussed three bottlenecks toward AI to automate the model compression process, reducing a day of deep learning computing. These bandwidth-efficient algorithms to reduce the required bandwidth: Deep Compression reduces compression on ResNet50. AMC: AutoML for Model Compression and Acceleration on Mobile Devices, ECCV’18

Figure 15: Our reinforcement learning agent (AMC) can prune the model to a lower density than achieved by human experts without loss of accuracy. (Human expert: 3.4× compression on ResNet50. AMC: 5× compression on ResNet50.)

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Jia Li, Song Han
AMC: AutoML for Model Compression and Acceleration on Mobile Devices, ECCV’18
<table>
<thead>
<tr>
<th>Model</th>
<th>MAC</th>
<th>Top-1</th>
<th>Top-5</th>
<th>Latency</th>
<th>Speed</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNet</td>
<td>569M</td>
<td>70.6%</td>
<td>89.5%</td>
<td>119.0ms</td>
<td>8.4 fps</td>
<td>20.1MB</td>
</tr>
<tr>
<td>AMC (50% MAC)</td>
<td>285M</td>
<td>70.5%</td>
<td>89.3%</td>
<td>64.4ms</td>
<td>15.5 fps</td>
<td>14.3MB</td>
</tr>
<tr>
<td>AMC (50% Time)</td>
<td>272M</td>
<td>70.2%</td>
<td>89.2%</td>
<td>59.7ms</td>
<td>16.8 fps</td>
<td>13.2MB</td>
</tr>
</tbody>
</table>

Latency measured with TF-Lite on Samsung Galaxy S7 Edge with Qualcomm Snapdragon SoC. Single core, Batch size = 1 (mobile, latency oriented)

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Jia Li, Song Han
AMC: AutoML for Model Compression and Acceleration on Mobile Devices, ECCV’18
Outlook: Design Automation for Neural Networks

Figure 12: Save engineer bandwidth through design automation: using AI to improve AI.
Thank you!

Hardware, AI and Neural-nets

PI: Song Han

Hanrui Wang
Yujun Lin
Ji Lin
Zhijian Liu
Hanrui Wang
Jiacheng Yang
Zhekai Zhang

songhan@mit.edu
Thank you!

Hardware, AI and Neural-nets