Hardware for Deep Learning

Bill Dally
Stanford and NVIDIA

Stanford Platform Lab Retreat
June 3, 2016
HARDWARE AND DATA ENABLE DNNS

Courtesy – AlexNet, NIPS 2012
THE NEED FOR SPEED

Larger data sets and models lead to better accuracy but also increase computation time. Therefore progress in deep neural networks is limited by how fast the networks can be computed.

Likewise the application of convnets to low latency inference problems, such as pedestrian detection in self-driving car video imagery, is limited by how fast a small set of images, possibly a single image, can be classified.

More data ➔ Bigger Models ➔ More Need for Compute
But Moore’s law is no longer providing more compute…

Lavin & Gray, Fast Algorithms for Convolutional Neural Networks, 2015
Deep Neural Network

What is frames/J and frames/s/mm$^2$ for training & inference?

4 Distinct Sub-problems

- **Training**
  - Convolutional: Train Conv
  - Fully-Conn.: Train FC

- **Inference**
  - Convolutional: Inference Conv
  - Fully-Conn.: Inference FC

**Weight Reuse**
- B x S Weight Reuse Act Dominated
- B Weight Reuse Weight Dominated

**32b FP - large batches**
- Minimize Training Time
- Enables larger networks

**8b Int - small (unit) batches**
- Meet real-time constraint
Inference
Precision

Use the "smallest" representation that doesn't sacrifice accuracy
(32FP -> 4-6 bits quantized)
Number Representation

<table>
<thead>
<tr>
<th>Format</th>
<th>S</th>
<th>E</th>
<th>M</th>
<th>Range</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP32</td>
<td>1</td>
<td>8</td>
<td>23</td>
<td>$10^{-38} - 10^{38}$</td>
<td>.000006%</td>
</tr>
<tr>
<td>FP16</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>$6\times10^{-8} - 6\times10^{4}$</td>
<td>.05%*</td>
</tr>
<tr>
<td>Int32</td>
<td>1</td>
<td></td>
<td>31</td>
<td>$0 - 2\times10^{9}$</td>
<td>$\frac{1}{2}$</td>
</tr>
<tr>
<td>Int16</td>
<td>1</td>
<td></td>
<td>15</td>
<td>$0 - 6\times10^{4}$</td>
<td>$\frac{1}{2}$</td>
</tr>
<tr>
<td>Int8</td>
<td>1</td>
<td></td>
<td>7</td>
<td>$0 - 127$</td>
<td>$\frac{1}{2}$</td>
</tr>
<tr>
<td>Binary</td>
<td>1</td>
<td></td>
<td></td>
<td>$0 - 1$</td>
<td>$\frac{1}{2}$</td>
</tr>
</tbody>
</table>
## Cost of Operations

<table>
<thead>
<tr>
<th>Operation:</th>
<th>Energy (pJ)</th>
<th>Area (µm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8b Add</td>
<td>0.03</td>
<td>36</td>
</tr>
<tr>
<td>16b Add</td>
<td>0.05</td>
<td>67</td>
</tr>
<tr>
<td>32b Add</td>
<td>0.1</td>
<td>137</td>
</tr>
<tr>
<td>16b FP Add</td>
<td>0.4</td>
<td>1360</td>
</tr>
<tr>
<td>32b FP Add</td>
<td>0.9</td>
<td>4184</td>
</tr>
<tr>
<td>8b Mult</td>
<td>0.2</td>
<td>282</td>
</tr>
<tr>
<td>32b Mult</td>
<td>3.1</td>
<td>3495</td>
</tr>
<tr>
<td>16b FP Mult</td>
<td>1.1</td>
<td>1640</td>
</tr>
<tr>
<td>32b FP Mult</td>
<td>3.7</td>
<td>7700</td>
</tr>
<tr>
<td>32b SRAM Read (8KB)</td>
<td>5</td>
<td>N/A</td>
</tr>
<tr>
<td>32b DRAM Read</td>
<td>640</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Relative Energy Cost**

**Relative Area Cost**

Energy numbers are from Mark Horowitz “Computing’s Energy Problem (and what we can do about it)”, ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.
The Importance of Staying Local

LPDDR DRAM
GB

On-Chip SRAM
MB

640pJ/word

50pJ/word

Local SRAM
KB

5pJ/word
Mixed Precision

Store weights as 4b using Trained quantization, decode to 16b

Store activations as 16b

16b x 16b multiply round result to 16b

accumulate 24b or 32b to avoid saturation

Batch normalization important to ‘center’ dynamic range
Trained Quantization

Han et al. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding, ICLR 2016 (Best Paper)
2-6 Bits/Weight Sufficient

Accuracy vs Number of bits per effective weight in all
FC layers

Accuracy vs Number of bits per effective weight in all
Conv layers
Sparsity

Don’t do operations that don’t matter (10x - 30x)
Sparsity

before pruning

after pruning

pruning synapses

pruning neurons

Han et al. Learning both Weights and Connections for Efficient Neural Networks, NIPS 2015
Retrain To Recover Accuracy

- Train Connectivity
- Prune Connections
- Train Weights

![Accuracy Loss vs Pruned Percentage Graph](graph.png)

- L2 regularization w/o retrain
- L1 regularization w/o retrain
- L1 regularization w/ retrain
- L2 regularization w/ retrain
- L2 regularization w/ iterative prune and retrain

Han et al. Learning both Weights and Connections for Efficient Neural Networks, NIPS 2015
Pruning + Trained Quantization

The graph shows the accuracy loss as a function of the model size ratio after compression. Different lines represent different methods:

- Red circles: Pruning + Quantization
- Purple triangles: Pruning Only
- Yellow squares: Quantization Only
- Green diamonds: SVD

The x-axis represents the model size ratio after compression, ranging from 2% to 20%. The y-axis represents the accuracy loss, ranging from 0.5% to 4.5%.
Pruning Neural Talk And LSTM

- **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**

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

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

- **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
Fixed-Function Hardware
- Diannao improved CNN computation efficiency by using dedicated functional units and memory buffers optimized for the CNN workload.
- Multiplier + adder tree + shifter + non-linear lookup orchestrated by instructions
- Weights in off-chip DRAM
- 452 GOP/s, 3.02 mm^2 and 485 mW

**Table 6.** Characteristics of accelerator and breakdown by component type (first 5 lines), and functional block (last 7 lines).
Efficient Inference Engine

<table>
<thead>
<tr>
<th>Power (mW)</th>
<th>Area (μm²)</th>
<th>[%]</th>
<th>[%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>9.157</td>
<td>638.024</td>
<td>60.42%</td>
</tr>
<tr>
<td>memory</td>
<td>5.416</td>
<td>594.786</td>
<td>59.15%</td>
</tr>
<tr>
<td>clock network</td>
<td>1.874</td>
<td>866</td>
<td>20.46%</td>
</tr>
<tr>
<td>register</td>
<td>1.026</td>
<td>9.465</td>
<td>11.20%</td>
</tr>
<tr>
<td>combinational</td>
<td>0.841</td>
<td>8.946</td>
<td>9.18%</td>
</tr>
<tr>
<td>filler cell</td>
<td></td>
<td>23,961</td>
<td></td>
</tr>
<tr>
<td>Act_queue</td>
<td>0.112</td>
<td>758</td>
<td>1.23%</td>
</tr>
<tr>
<td>PtrRead</td>
<td>1.807</td>
<td>121,849</td>
<td>19.73%</td>
</tr>
<tr>
<td>SpmatRead</td>
<td>4.955</td>
<td>469,412</td>
<td>54.11%</td>
</tr>
<tr>
<td>ArithmUnit</td>
<td>1.162</td>
<td>3,110</td>
<td>12.68%</td>
</tr>
<tr>
<td>ActRW</td>
<td>1.122</td>
<td>18,934</td>
<td>12.25%</td>
</tr>
<tr>
<td>filler cell</td>
<td></td>
<td>23,961</td>
<td></td>
</tr>
</tbody>
</table>

Note: The table above shows the power and area breakdown of different components in the Efficient Inference Engine (EIE) architecture. The components include memory, clock network, register, combinational, filler cell, Act_queue, PtrlRead, SpmatRead, ArithmUnit, ActRW, and filler cell. Each component's power and area are given in milliwatts (mW) and micrometers squared (μm²), respectively, along with the percentage contribution to the total power and area.
Conv layers of VGG – 16b arithmetic

<table>
<thead>
<tr>
<th></th>
<th>Activations</th>
<th>Weights</th>
<th>Math</th>
</tr>
</thead>
<tbody>
<tr>
<td>256K SRAM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparse Math</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4M SRAM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero Compress</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full Compress</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>On-Chip</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Inference Summary

- Prune 70-95% of static weights (3-20x reduction in size and ops) [3-20x]
- Exploit dynamic sparsity of activations (3x reduction in ops) [10-60x]
- Reduce precision to ~4 bits (8x size reduction) [80-480x]
- 25-150x smaller fits in smaller RAM (100x less power) [8,000-48,000x]
- Dedicated hardware to eliminate overhead
  - Facilitates compression and sparsity
  - Exploit locality
Faster Training through Parallelism
Data Parallel - Run Multiple Inputs In Parallel

- Doesn’t affect latency for one input
- Requires P-fold larger batch size
- For training requires coordinated weight update
Parameter Update

Parameter Server \[ p' = p + \Delta p \]

Model Workers

Data Shards

Large Scale Distributed Deep Networks, Jeff Dean et al., 2013
Model-Parallel Convolution – by output region \((x,y)\)

Kernels
Multiple 3D\(K_{uvkj}\)

6D Loop
For all region \(XY\)
For each output map \(j\)
For each input map \(k\)
For each pixel \(x,y\) in \(XY\)
For each kernel element \(u,v\)
\[ B_{xyj} += A_{(x-u)(y-v)k} \times K_{uvkj} \]
Parallel GPUs on Deep Speech 2

We find synchronous SGD useful because it is reproducible and deterministic. We have found that the appearance of non-determinism in our system often signals a serious bug, and so having reproducibility as a goal has greatly facilitated debugging. In contrast, asynchronous methods such as asynchronous SGD with parameter servers as found in Dean et al. [17] typically do not provide reproducibility and are therefore more difficult to debug.

Synchronous SGD is simple to understand and implement. It scales well as we add multiple nodes to the training process.

Figure 4 shows that time taken to train one epoch halves as we double the number of GPUs that we train on, thus achieving near-linear weak scaling. We keep the minibatch per GPU constant at 64 during this experiment, effectively doubling the minibatch as we double the number of GPUs.

Although we have the ability to scale to large minibatches, we typically use either 8 or 16 GPUs during training with a minibatch of 512 or 1024, in order to converge to the best result.

Since all-reduce is critical to the scalability of our training, we wrote our own implementation of the ring algorithm [46, 63] for higher performance and better stability. Our implementation avoids extraneous copies between CPU and GPU, and is fundamental to our scalability.

We built our implementation using MPI send and receive, along with CUDA kernels for the element-wise operations.

Table 7 compares the performance of our all-reduce implementation with that provided by OpenMPI version 1.8.5. We report the time spent in all-reduce for a full training run that ran for one epoch on our English dataset using a 5 layer, 3 recurrent layer architecture with 2560 hidden units for all layers. In this table, we use a minibatch of 64 per GPU, expanding the algorithmic minibatch as we scale to more GPUs. We see that our implementation is considerably faster than OpenMPI's when the communication is within a node (8 GPUs or less). As we increase the number of GPUs and increase the amount of inter-node communication, the gap shrinks, although our implementation is still 2-4X faster.

All of our training runs use either 8 or 16 GPUs, and in this regime, our all-reduce implementation results in 2.5× faster training for the full training run, compared to using OpenMPI directly. Optimizing all-reduce has thus resulted in important productivity benefits for our experiments, and has made our simple synchronous SGD approach scalable.
### 4 Distinct Sub-problems

<table>
<thead>
<tr>
<th>Training</th>
<th>Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Convolutional</strong></td>
<td><strong>Low-Precision</strong></td>
</tr>
</tbody>
</table>
| 32b FP  
Batch  
Activation Storage  
GPUs ideal  
Comm for Parallelism | Compressed  
Latency-Sensitive  
Fixed-Function HW  
Arithmetic Dominated |
| **Fully-Conn.** |  |
| 32b FP  
Batch  
Weight Storage  
GPUs ideal  
Comm. for Parallelism | Low-Precision  
Compressed  
Latency-Sensitive  
No weight reuse  
Fixed-Function HW  
Storage dominated |
| 32b FP - large batches  
Minimize Training Time  
Enables larger networks | 8b Int - small (unit) batches  
Meet real-time constraint |
Summary

• Fixed-function hardware will dominate inference (100-10,000x gain)
  – Sparse, low-precision, compressed (25-150x smaller)
  – 3x dynamic sparsity
  – All weights and activations from local memory (10-100x less energy)
  – Flexible enough to track evolving algorithms

• GPUs will dominate training
  – Only dynamic sparsity (3x activations, 2x dropout)
  – Medium precision (FP16 – for weights), stochastic rounding
  – Large memory footprint (batch x retained activations)
  – Communication BW scales with parallelism