Efficiently Training CNN with FPGA

Presented By

Yu Wang
Associate Professor, Dept. of E.E., Tsinghua University
Co-founder of DeePhi Tech.
An Era of Deep Learning

Everything & Everywhere
Two Stages in Deep Learning: Training and Inference

Rocket: Deep learning
Fuel: Big data
Engine: Computing platform

According to Prof. Andrew Ng

Training
- How accurate the model can be

Inference
- How many applications can use DL

Top-5 Accuracy
- 84.70%
- 88.30%
- 93.30%
- 96.40%
- 97.00%

Server
→
Client
→
Client
→
Efficient inference has been highly focused on performance.

FPGA can achieve similar energy efficiency to ASIC and GPU.

A latest version can be found at: https://nicsefc.ee.tsinghua.edu.cn/projects/neural-network-accelerator/
What about training?

> Training is also heavy work!

Training a VGG model on ImageNet dataset:

\[
\text{Workload} = \left\{ \begin{array}{c}
\text{Samples} \\
~10^6
\end{array} \right\} \times \left\{ \begin{array}{c}
\text{(Inference+Update)} \\
~10^{10} \text{ FLOPs}
\end{array} \right\} \times \left\{ \begin{array}{c}
\text{Iterations} \\
~100
\end{array} \right\}
\]

NVIDIA Titan Xp (TDP 250W) \(\times 4 \times 60\text{hr}\)
We also need energy efficient training

> **For cloud services:**
  > The building cost of a data center is about $10000-20000/kW^1$

• **For end applications:**
  • changing environment -> changing model
  • Platform power limitation
  • Car: 10-100w
  • Satellite: hard to dissipate heat

---

What does training do?

> Stochastic Gradient Descent (or other variants)

\[ y = W \cdot x \]

\[ \delta y \]

\[ \delta x = \delta y \cdot W' \]

\[ \delta W = \delta y \cdot x' \]

\[ W = W + \lambda \cdot \delta W \]

- Similar to inference, the main workload of training is still **sum of product**
- Maybe we can do the same things as for inference
What we have done with inference

DL Framework

Software-Hardware Co-design

DPU Platform

ASIC

XILINX

© Copyright 2018 Xilinx
Techniques for inference: Software-Hardware Co-design

> Quantization

Negligible accuracy loss with 8-bit fixed-point weight & activation

Save FPGA resources

<table>
<thead>
<tr>
<th></th>
<th>Xilinx Logic multiplier</th>
<th>Xilinx DSP multiply &amp; add</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GoogLeNet</td>
<td>708</td>
<td>800</td>
</tr>
<tr>
<td>VGG-16</td>
<td>221</td>
<td>451</td>
</tr>
<tr>
<td>SqueezeNet</td>
<td>1112</td>
<td>111</td>
</tr>
<tr>
<td>VGG-CNN-F</td>
<td>289</td>
<td>17</td>
</tr>
</tbody>
</table>

Techniques for inference: Software-Hardware Co-design

**Sparsification**

Negligible WER increase with 10% weights

6x acceleration with customized hardware

---

Q: Can we take everything from inference to training?

A: Things are different!
Challenges

Quantization for training is more difficult than for inference

<table>
<thead>
<tr>
<th>Method</th>
<th>$k_W$</th>
<th>$k_A$</th>
<th>$k_G$</th>
<th>$k_E$</th>
<th>Opt</th>
<th>BN</th>
<th>MNIST</th>
<th>SVHN</th>
<th>CIFAR10</th>
<th>ImageNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>1</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>Adam</td>
<td>✓</td>
<td>1.29</td>
<td>2.30</td>
<td>9.90</td>
<td>-</td>
</tr>
<tr>
<td>BNN</td>
<td>1</td>
<td>1</td>
<td>32</td>
<td>32</td>
<td>Adam</td>
<td>✓</td>
<td>0.96</td>
<td>2.53</td>
<td>10.15</td>
<td>-</td>
</tr>
<tr>
<td>BWN$^1$</td>
<td>1</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>withM</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>43.2/20.6</td>
</tr>
<tr>
<td>XNOR</td>
<td>1</td>
<td>1</td>
<td>32</td>
<td>32</td>
<td>Adam</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>55.8/30.8</td>
</tr>
<tr>
<td>TWN</td>
<td>2</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>withM</td>
<td>✓</td>
<td>0.65</td>
<td>-</td>
<td>7.44</td>
<td>34.7/13.8</td>
</tr>
<tr>
<td>TTQ</td>
<td>2</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>Adam</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>6.44</td>
<td>42.5/20.3</td>
</tr>
<tr>
<td>DoReFa$^2$</td>
<td>8</td>
<td>8</td>
<td>32</td>
<td>8</td>
<td>Adam</td>
<td>✓</td>
<td>-</td>
<td>2.30</td>
<td>-</td>
<td>47.0/</td>
</tr>
<tr>
<td>TernGrad$^3$</td>
<td>32</td>
<td>32</td>
<td>2</td>
<td>32</td>
<td>Adam</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>14.36</td>
<td>42.4/19.5</td>
</tr>
<tr>
<td>WAGE</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>SGD</td>
<td>✓</td>
<td>0.40</td>
<td>1.92</td>
<td>6.78</td>
<td>51.6/27.8</td>
</tr>
</tbody>
</table>

Baseline : 36.7/15.3

Challenges

Pruning is applied when the network is well trained

- Cannot be accelerated utilizing sparsity
- Can be accelerated utilizing sparsity

Using sparsity only brings more workload to training

Opportunity

> Previous hardware only utilizes operand sparsity

>> \( C = \Sigma A B \) (A and B can be sparse)

>> Apply to inference and back-propagate phases

\[
\begin{align*}
\text{vector} &\quad a_0 & a_2 & a_3 \\
\text{weight} &\quad \begin{array}{c}
0 & 0 & w_{0,2} & w_{0,3} & 0 & w_{0,5} \\
0 & 0 & 0 & 0 & 0 & w_{1,5} \\
0 & 0 & 0 & 0 & 0 & w_{2,5} \\
0 & 0 & 0 & 0 & 0 & w_{3,5} \\
0 & 0 & 0 & 0 & 0 & w_{4,5} \\
0 & 0 & 0 & 0 & 0 & w_{5,5} \\
0 & 0 & 0 & 0 & 0 & w_{6,5} \\
0 & 0 & 0 & 0 & 0 & w_{7,5} \\
\end{array}
\end{align*}
\]

ESE\(^1\)

• We should also utilize **result sparsity**

• \( C = \Sigma A B \) (C is sparse)

• Apply to update phase: \( \delta W = \delta y \cdot x' \)

Our Solution

1. A training process with fixed-point gradient, activation, weight
   - 32-bit float multiplication -> 8-bit fixed-point multiplication
   - 8x energy saving

2. Pruning before the training process converge
   - Reduce floating point training process to 1/3

3. Customized FPGA accelerator design to accelerate the training of the sparse model.
   - Potentially more than 3x speed up for the largest layers
Training with fixed-point data

> Replace all the floating point data with fixed-point data

> Short bit-width for MAC: reduce computation cost

> Long bit-width for weight storage: able to accumulate small gradients
Pruning before convergence

> 2D-kernel-wise pruning for CONV layers
> Normal pruning for FC layers
> For the simplicity of hardware and sparse coding style

M×N fully connected layer

M×N channel convolution layer with 3×3 kernels
Pruning before convergence

- Dataset: CIFAR-10; Network: VGG-11
- Prune standard: <1% accuracy loss

More than half of the weights can be pruned
Pruning and Quantization

- Dataset: CIFAR-10; Network: VGG-11
- Weight: 32/24/16 bit; Activation: 8 bit
- 24bit weight and pruning as early as 60 epochs is enough for a good training result
Hardware design

**Diagram:**
- **CPU**
- **DRAM**
- **FPGA**
  - **Interconnect**
  - **Pre Stream**
  - **Post Stream**
  - **PE 1**
  - **PE 2**
  - **PE N**

**Controller**
- FP: ReLU/Pooling
- BP: ReLU/Pooling
- BP: Sum
- FP&BP: CONV/FC

**PE**
- **AGU**
  - **X buf**
  - **Y buf**
  - **CONV/FC cnt**
- **MUX**
  - **weight Buf**
  - **data buf**
  - **MAC array**
- **Controller**
  - **res buf**
- **Pre Stream**
- **Post Stream**

**Annotations:**
- **Pre Stream res buf**
- **Post Stream res buf**
- **Pre Stream Y buf**
- **Post Stream X buf**
- **Pre Stream CONV/FC cnt**
- **Post Stream CONV/FC cnt**

**Text Notes:**
- FP: Forward Processing
- BP: Backward Processing
- FP&BP: Forward and Backward Processing
- ReLU: Rectified Linear Unit
- Pooling: Pooling Operation
- Sum: Summation Operation
- CONV/FC: Convolution/Fully Connected Layer
Parallelism

> Batch is welcomed for training
  ✓ Parallel batch process

> Input / Output channel
  ! Affected by sparsity
  ✓ Duplicate input channel and parallelize output channel

> Feature map pixel / Convolution kernel
  ! Loop size varies greatly
  ✓ Configurable parallelism

Inference:
\[ F'_j = \sum_{i=0}^{M-1} \text{conv2d}(F^{l-1}_i, K_{ij}) + b_j \quad j = 0, 1, ..., N - 1 \]

Update:
\[ dK_{ij} = \text{conv2d}(F^{l-1}_i, dF^l_j) \quad j = 0, ..., N - 1 \]
\[ i = 0, ..., M - 1 \]
Configurable parallelism

> Each PE locally do multiply and accumulate

> Inference: 4 PEs process 4 output pixels for 2D convolution in parallel
  >> Data is shared within a group of PE to utilize the data locality
Configurable parallelism

- Each PE locally do multiply and accumulate
- Inference: 4 PEs process 4 output pixels for 2D convolution in parallel
  - Data is shared within a group of PE to utilize the data locality
- Update: same with inference
  - Waste the computation units!
Configurable parallelism

- Each PE locally do multiply and accumulate
- Inference: 4 PEs process 4 output pixels for 2D convolution in parallel
  - Data is shared within a group of PE to utilize the data locality
- Update: 4 PEs process a part of the gradient in parallel
  - Partial gradients are summed in a successive module
PE structure to support result sparsity

- Random access buffer for both input and output data
- Switch and MUX for index and address generation unit
Use Compressed Sparse Block (CSB) format

- **FC layers**: split the weight matrix into blocks and encode each element in a block with a 2D coordinate.
- **CONV layers**: we split the kernels into channel blocks and encode each 2d kernel in a block with a 2D coordinate.
- **Transpose by switch both the 2D coordinate in hardware and the block order in software**
**Scheduling**

**Inference:**
Each PE calculates 4 output channels

**Back-propagate:**
Each PE calculates 4 input channels
Evaluation

- FPGA Platform: KCU1500 development board

- Hardware Parameter:
  - 250MHz clock
  - 32 PE, each with 32 MAC unit to process a batch of 32 images
  - 2 DDR4-2400 external memory

- Bounded by on-chip memory

- Accumulation buffer occupies most of the RAMs because:
  - Large bitwidth (32 compared with 8 for data and param)
  - Ping-pong strategy to cover data transfer time
  - Ultra-RAM may relief this problem
## Evaluation

> Performance breakdown (simulation)

<table>
<thead>
<tr>
<th>layer</th>
<th>Comp. (GOP)</th>
<th>Time (us)</th>
<th>Perf. (GOP/s)</th>
<th>bound type</th>
<th>Utilize rate</th>
<th>Time (us)</th>
<th>Perf. (GOP/s)</th>
<th>bound type</th>
<th>Utilize rate</th>
<th>Time (us)</th>
<th>Perf. (GOP/s)</th>
<th>bound type</th>
<th>Utilize rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>0.11</td>
<td>733</td>
<td>154.4</td>
<td>B</td>
<td>27%</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4158</td>
<td>-</td>
<td>27.2</td>
</tr>
<tr>
<td>conv2</td>
<td>1.21</td>
<td>1487</td>
<td>812.2</td>
<td>C</td>
<td>95%</td>
<td>1487</td>
<td>812.5</td>
<td>C</td>
<td>95%</td>
<td>2804</td>
<td>430.8</td>
<td>B</td>
<td>50%</td>
</tr>
<tr>
<td>conv3_1</td>
<td>1.21</td>
<td>1696</td>
<td>712.4</td>
<td>C</td>
<td>97%</td>
<td>1694</td>
<td>713.0</td>
<td>C</td>
<td>97%</td>
<td>2477</td>
<td>487.7</td>
<td>B</td>
<td>67%</td>
</tr>
<tr>
<td>conv3_2</td>
<td>2.42</td>
<td>2877</td>
<td>839.7</td>
<td>C</td>
<td>98%</td>
<td>2876</td>
<td>840.1</td>
<td>C</td>
<td>98%</td>
<td>4398</td>
<td>549.3</td>
<td>B</td>
<td>64%</td>
</tr>
<tr>
<td>conv4_1</td>
<td>1.21</td>
<td>1217</td>
<td>992.3</td>
<td>C</td>
<td>97%</td>
<td>1217</td>
<td>992.9</td>
<td>C</td>
<td>97%</td>
<td>2686</td>
<td>449.8</td>
<td>B</td>
<td>44%</td>
</tr>
<tr>
<td>conv4_2</td>
<td>2.42</td>
<td>1457</td>
<td>1658.4</td>
<td>C</td>
<td>97%</td>
<td>1456</td>
<td>1659.2</td>
<td>C</td>
<td>97%</td>
<td>3933</td>
<td>614.2</td>
<td>B</td>
<td>36%</td>
</tr>
<tr>
<td>conv5_1</td>
<td>0.60</td>
<td>366</td>
<td>1651.7</td>
<td>B</td>
<td>32%</td>
<td>358</td>
<td>1687.7</td>
<td>B</td>
<td>33%</td>
<td>915</td>
<td>659.8</td>
<td>B</td>
<td>13%</td>
</tr>
<tr>
<td>conv5_2</td>
<td>0.60</td>
<td>365</td>
<td>1652.7</td>
<td>B</td>
<td>32%</td>
<td>358</td>
<td>1688.7</td>
<td>B</td>
<td>33%</td>
<td>915</td>
<td>660.0</td>
<td>B</td>
<td>13%</td>
</tr>
<tr>
<td>dense6</td>
<td>0.02</td>
<td>329</td>
<td>51.0</td>
<td>B</td>
<td>1%</td>
<td>328</td>
<td>51.1</td>
<td>B</td>
<td>1%</td>
<td>717</td>
<td>23.4</td>
<td>B</td>
<td>0.5%</td>
</tr>
<tr>
<td>dense7</td>
<td>0.02</td>
<td>329</td>
<td>51.0</td>
<td>B</td>
<td>1%</td>
<td>328</td>
<td>51.1</td>
<td>B</td>
<td>1%</td>
<td>717</td>
<td>23.4</td>
<td>B</td>
<td>0.5%</td>
</tr>
<tr>
<td>dense8</td>
<td>0.003</td>
<td>75</td>
<td>4.4</td>
<td>B</td>
<td>0.1%</td>
<td>74</td>
<td>4.4</td>
<td>B</td>
<td>0.1%</td>
<td>272</td>
<td>1.2</td>
<td>B</td>
<td>0.02%</td>
</tr>
<tr>
<td>total</td>
<td>9.81</td>
<td>10931</td>
<td>897.5</td>
<td>-</td>
<td>-</td>
<td>10988</td>
<td>892.8</td>
<td>-</td>
<td>-</td>
<td>23993</td>
<td>408.9</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
### Evaluation

> Performance comparison with state-of-the-art CNN inference/training accelerators and GPU

<table>
<thead>
<tr>
<th>Platform</th>
<th>TCAD 18</th>
<th>FPGA 17</th>
<th>ESE</th>
<th>FCNN</th>
<th>FPT17</th>
<th>FPDeep</th>
<th>Proposed</th>
<th>GPU Titan X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XC7Z020</td>
<td>GX1150</td>
<td>KU060</td>
<td>Maxeler MPC-X</td>
<td>ZU19EG</td>
<td>VC709</td>
<td>KCU1500</td>
<td>GM200</td>
</tr>
<tr>
<td>Function</td>
<td>Inference</td>
<td>Inference</td>
<td>Inference</td>
<td>Inference</td>
<td>Training</td>
<td>Training</td>
<td>Inference</td>
<td>Training</td>
</tr>
<tr>
<td>Quantization</td>
<td>fixed 8</td>
<td>fixed 8/16</td>
<td>fixed 12</td>
<td>float 32</td>
<td>float 32</td>
<td>float 32</td>
<td>fixed 16</td>
<td>fixed 8/24</td>
</tr>
<tr>
<td>Sparsity</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Performance (GOP/s)</td>
<td>84.3</td>
<td>645.25</td>
<td>2516</td>
<td>62.06</td>
<td>7.01</td>
<td>86.12</td>
<td>1022</td>
<td>897.5</td>
</tr>
<tr>
<td>Power (W)</td>
<td>3.5</td>
<td>21.2</td>
<td>41</td>
<td>N.A.</td>
<td>27.3</td>
<td>14.2</td>
<td>32</td>
<td>29</td>
</tr>
<tr>
<td>Energy Eff. (GOP/s/W)</td>
<td>24.1</td>
<td>30.43</td>
<td>61.4</td>
<td>N.A.</td>
<td>0.27</td>
<td>6.05</td>
<td>31.97</td>
<td>30.95</td>
</tr>
</tbody>
</table>
Future work

> More functions for training
  >> BN layers, quantization for other optimizers

> Scaling up the design?
  >> Reduce memory consumption by optimizing memory design
  >> Lower down bandwidth requirement by optimizing scheduling
  >> We also have HBM

> Improve energy efficiency?
  >> More advanced training methods

> Evaluation with real training tasks
  >> We never know if the method applies to a new application
Adaptable.

Intelligent.