Compute-Efficient Neural-Network Acceleration

Ephrem Wu, Xiaoqian Zhang, David Berman, Inkeun Cho, John Thendean
FPGA 2019
2/24/2019
Goal

Build a compute-efficient and reconfigurable neural-net inference accelerator

- cat
- traffic light
- table
- airplane
We Don’t Have to Leave DSP Performance Behind

Achieved Operating Clock Rate (MHz)
Max. Datasheet DSP Clock Rate (MHz)
Realized Performance (%)

Clock Rate (MHz)

Achieved Clock Rate

References from FPL17

Our prior work on MxV
Key Design Principles

1. Make compute fast.
2. Keep compute busy.
3. Simplify implementation.

© Copyright 2019 Xilinx
Compute-Efficient Neural Processing

> **Case Study: GoogLeNet v1 Inference**
  > 3 parallel GoogLeNets with their own weights
  > Aggregate 3046 images/sec, 3.3 ms latency

> **Platform: VCU1525 board with VU9P-2 FPGA**

> **Compute**
  > DSP supertile arrays running at 720 MHz
  > Consumes only 56% DSP48 tiles
  > DSP cycles 95% utilized
  > Per-tensor block floating-point, 8-/16-bit significands

> **Memory**
  > No external DRAM on accelerator card used
  > All tensors stored in UltraRAM & BRAM
  > 1/2 DSP clock rate
System and Accelerator Block Diagrams

System

Host DRAM

CPU

FPGA DRAM (Unused)

FPGA Neural Network Accelerator

VCU1525 Accelerator Card

PCIe
System and Accelerator Block Diagrams

System

- Host DRAM
- CPU
- FPGA DRAM (Unused)
- VCU1525 Accelerator Card

AcCELERATOR

- Neural-Net Processor Chain
- Input images
- Weights, biases, and instructions
- Logits

From PCIe

To PCIe

3x
Apples-to-Apples Accelerator Comparison

Throughput per device doesn’t tell the whole story

Differences: FPGAs, total compute, data movement, numerical formats, and clock rates

Normalize results

- Achieved Clock Rate: Actual DSP clock compared to datasheet $F_{\text{max}}$
- Compute Efficiency: % available DSP cycles that do useful work

Overall Efficiency = % DSP $F_{\text{max}} \times$ Compute Efficiency

Make compute fast
Keep compute busy
## GoogLeNet Inference Comparison

<table>
<thead>
<tr>
<th>FPGA Implementation</th>
<th>Realized Clock Rate (% of Peak)</th>
<th>Compute Efficiency</th>
<th>Overall Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ngo, 2016</td>
<td>15.4%</td>
<td>15.0%</td>
<td>2.3%</td>
</tr>
<tr>
<td>Venieris, 2017</td>
<td>19.2%</td>
<td>66.6%</td>
<td>12.8%</td>
</tr>
<tr>
<td>Shen, 2017</td>
<td>26.1%</td>
<td>93.8%</td>
<td>24.5%</td>
</tr>
<tr>
<td>Huang, 2018</td>
<td>30.8%</td>
<td>81.4%</td>
<td>25.1%</td>
</tr>
<tr>
<td>Gokhale, 2017</td>
<td>38.4%</td>
<td>91.0%</td>
<td>34.9%</td>
</tr>
<tr>
<td>This work</td>
<td>92.9%</td>
<td>95.5%</td>
<td>88.7%</td>
</tr>
</tbody>
</table>

The higher the clock rate, the harder it is to reach high compute efficiency.

Bang for the buck from each DSP resource.
Convolution: From Tensors View to Matrix View
2D Convolution: Tensor View

Input Tensor $\mathbf{X} \in \mathbb{R}^{H_1 \times W_1 \times C_{in}}$
2D Convolution: Tensor View

Input Tensor $X \in \mathbb{R}^{H_1 \times W_1 \times C_{in}}$

Output Tensor $Y \in \mathbb{R}^{H_2 \times W_2 \times C_{out}}$

Output channel 0 Output channel 1 Output channel 2 Output channel 3

© Copyright 2019 Xilinx
2D Convolution: Tensor View

Although input and output tensors have three axes, convolution is 2D, not 3D.

There's a 2D convolution filter mask for every input-output channel pair (3 \times 4 = 12 in this example).
2D Convolution: Tensor View

Input Tensor $\mathcal{X} \in \mathbb{R}^{H_1 \times W_1 \times C_{in}}$

Output Tensor $\mathcal{Y} \in \mathbb{R}^{H_2 \times W_2 \times C_{out}}$

Filter Weight Tensor $\mathcal{W} \in \mathbb{R}^{F_y \times F_x \times C_{in} \times C_{out}}$
2D Convolution: From Tensor to Flat View

Input Image

Input Channels

Parallel convolution sliding windows, one per input channel

© Copyright 2019 Xilinx
2D Convolution: Flat View

Input Image

Parallel convolution sliding windows, one per input channel
2D Convolution: Flat View

Input Image

Input Channels

Parallel convolution sliding windows, one per input channel

© Copyright 2019 Xilinx
2D Convolution: Flat Silicon View

- Input channel broadcast
- Per-channel element-wise multiplication with filter weights

Input Image

Input Channels

© Copyright 2019 Xilinx
2D Convolution: Flat Silicon View

Input Image

Input Channels

Accumulates $3 \times 3 \times 3 = 27$ products

Reduction per output channel

© Copyright 2019 Xilink
Interpreting Weights (Sort of)

Looking for edges

Looking for green patches

© Copyright 2019 Xilinx
Computing Convolution:
Tensor Matricization and MxV
Computing Convolution

> For each element in some output channel
  >> Input vector: Flattened input patches
  >> Weight vector: Flattened filter weights for this output channel
  >> Compute dot product between input vector and weight vector

> Input vector broadcast to multiple output channels
→ Hardware matrix-vector multiplier (MxV)
Reshaping Tensors to Matrices

3-Mode Output Tensor
\( \mathcal{Y} \in \mathbb{R}^{H_2 \times W_2 \times C_{out}} \)

Output Matrix
\( Y \in \mathbb{R}^{C_{out} \times H_2 W_2} \)

Elements per Output Channel
\( N = 28 \times 28 = 784 \)

Output channels
\( M = 192 \)
Reshaping Tensors to Matrices

3-Mode Output Tensor

\[ Y \in \mathbb{R}^{H_2 \times W_2 \times C_{out}} \]

\[ N = 28 \times 28 = 784 \]

Output Matrix

\[ Y \in \mathbb{R}^{C_{out} \times H_2 \times W_2} \]

\[ M = 192 \]

Output channels

Elements per Output Channel

\[ C_{out} = 192 \]

Input Channels × Weights per Filter

\[ K = 3 \times 3 \times 128 = 1152 \]

4-Mode Weight Tensor

\[ W \in \mathbb{R}^{F_y \times F_x \times C_{in} \times C_{out}} \]

\[ C_{in} = 128 \]

Weight Matrix

\[ W \in \mathbb{R}^{C_{out} \times F_y \times F_x \times C_{in}} \]

\[ M = 192 \]

© Copyright 2019 Xilinx
Reshaping Tensors to Matrices

3-Mode Output Tensor
\( \mathcal{Y} \in \mathbb{R}^{H_2 \times W_2 \times \text{out}} \)

Elements per Output Channel
\( N = 28 \times 28 = 784 \)

Output Matrix
\( \mathcal{Y} \in \mathbb{R}^{\text{out} \times H_2 \times W_2} \)

Output channels
\( M = 192 \)

\( \text{out} = 192 \)

\( 3 \times 3 \)

\( 28 \times 28 \)

4-Mode Weight Tensor
\( \mathcal{W} \in \mathbb{R}^{F_y \times F_x \times \text{in} \times \text{out}} \)

Input Channels × Weights per Filter
\( K = 3 \times 3 \times 128 = 1152 \)

Weight Matrix
\( \mathcal{W} \in \mathbb{R}^{\text{out} \times F_y \times F_x \times \text{in}} \)

\( \text{out} = 192 \)

\( F_y \times F_x \times \text{in} \times \text{out} \)

\( 3 \times 3 \)

\( 3 \times 3 \)

\( 28 \times 28 \)

3-Mode Input Tensor
\( \mathcal{X} \in \mathbb{R}^{H_1 \times W_1 \times \text{in}} \)

Input Matrix
\( \mathcal{X} \in \mathbb{R}^{F_y \times F_x \times \text{in} \times H_2 \times W_2} \)

\( H_1 \times W_1 \times \text{in} \)

\( F_y \times F_x \times \text{in} \times H_2 \times W_2 \)

© Copyright 2019 Xilinx
Example Weight Tensor Matricization Using Input Channel Interleaving

Input Channels × Weights per Filter

Input channel  R  Input channel  G  Input channel  B

1  4  7
2  5  8
3  6  9

Weight column, weight row, input channel

Ordering:

Output channel $j$}

```
RGB RGB RGB RGB RGB RGB RGB RGB RGB
1  2  3  4  5  6  7  8  9
```
Distributing Compute Over Hardware MxV Array

1. Divide weight matrix into blocks.
2. Copy weight block matrices into MxV array.
3. Divide input matrix into column vectors.
4. Run matrix-vector multiply-accumulate.
Maximizing Array Compute Efficiency

Assign input channels to MxV input lanes.

Assign output channels to MxV output lanes.
Maximize Clock Rate: Use DSP Supertiles [FPL '17]

DSP supertile column: 4 physically adjacent columns good for timing closure

Use DSP summing cascades instead of summing trees

Map Dot Product to Summing Cascade

Two DSP Supertiles in One PBlock
Maximize Clock Rate: Use DSP Supertiles

DSP Supertile Array = Matrix-Vector Multiplier: $z \leftarrow Pq + r$
Maximize Clock Rate: Use DSP Supertiles

Double-buffered operand cache in distributed RAM stores filter weights

Distributed RAM can run as fast as DSP tiles
Example: Time Interleaving Four Output Channels

3 × 3 Conv2D
Stride 1

Input feature reused for 4 cycles.

Input channel address (green box) moves ¼ as fast as weight address (red box).
Mapping GoogLeNet to a Processor Chain
GoogLeNet

Image in

Blue: 2D convolution or fully-connected plus activation (ReLU)
Red: Max-pooling layer
Green: Concatenation layer
Yellow: Softmax

1000 classes out


© Copyright 2019 Xilinx
There are 3 SLRs on each VU9P.
Each SLR contains a 4-processor chain.
Each chain is independent and has its own set of weights.
Each processor in the chain is event-driven (w/o central controller).

<table>
<thead>
<tr>
<th>Processor</th>
<th>DSP Supertile Array Input Lanes $N_1$</th>
<th>DSP Supertile Array Output Lanes $N_2$</th>
<th>Linear Layers in GoogLeNet</th>
<th>Cycles Per Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>21</td>
<td>8</td>
<td>1</td>
<td>691792</td>
</tr>
<tr>
<td>$P_2$</td>
<td>32</td>
<td>16</td>
<td>2</td>
<td>686432</td>
</tr>
<tr>
<td>$P_3$</td>
<td>96</td>
<td>16</td>
<td>54</td>
<td>708671</td>
</tr>
<tr>
<td>$P_4$</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>128000</td>
</tr>
</tbody>
</table>
Fast and Big Memory Wanted

Thousands

Weight Tensor Size

Weight Upload Bandwidth (GB/s)
Keep All Tensors On-Die in UltraRAM and BRAM

MxV doesn’t have to wait for memory

- Distributed RAM (bits to kilobits)
- Block RAM (10s of megabits)
- UltraRAM (100s of megabits)
- External DDR DRAM (10s of gigabits)

New in UltraScale+™

© Copyright 2019 Xilinx
UltraRAM vs. DDR4 Bandwidth

Memory Bandwidth

<table>
<thead>
<tr>
<th>Off-Chip DRAM</th>
<th>On-Chip SRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDR4 4x64</td>
<td>0.0768 TB/s</td>
</tr>
<tr>
<td>VU3P 320 URAMs</td>
<td>3.072 TB/s</td>
</tr>
<tr>
<td>VU9P 960 URAMs</td>
<td>9.216 TB/s</td>
</tr>
<tr>
<td>VU13P 1280 URAMs</td>
<td>12.288 TB/s</td>
</tr>
</tbody>
</table>

Holds three copies of GoogLeNet weights and activations in our design.
**Two Components of Compute Efficiency**

Balance factor is a “hurry-up-and-wait” metric.

Distribution efficiency measures multiply-add cycle utilization.

**Compute Efficiency** = Balance Factor × Distribution Efficiency

<table>
<thead>
<tr>
<th>MxV Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Balance Factor</th>
<th>Distribution Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>
## GoogLeNet Efficiency Metrics (ImageNet 1K Dataset)

<table>
<thead>
<tr>
<th>Processor</th>
<th>MxV Shape</th>
<th>Cycles Per Image</th>
<th>Balance Factor</th>
<th>Distribution Efficiency</th>
<th>Compute Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$21 \times 8$</td>
<td>691792</td>
<td>97.6%</td>
<td>100.0%</td>
<td>97.6%</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$32 \times 16$</td>
<td>686432</td>
<td>96.9%</td>
<td>100.0%</td>
<td>96.9%</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$96 \times 16$</td>
<td>708671</td>
<td>100.0%</td>
<td>95.2%</td>
<td>95.2%</td>
</tr>
<tr>
<td>$P_4$</td>
<td>$8 \times 1$</td>
<td>128000</td>
<td>18.1%</td>
<td>100.0%</td>
<td>18.1%</td>
</tr>
<tr>
<td>Pipeline</td>
<td></td>
<td>708671</td>
<td>98.8%</td>
<td>96.7%</td>
<td>95.5%</td>
</tr>
<tr>
<td>Resource</td>
<td>Used</td>
<td>Available</td>
<td>Utilization (%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>--------</td>
<td>-----------</td>
<td>-----------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>URAM</td>
<td>960</td>
<td>960</td>
<td>100.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSP48E2</td>
<td>3817</td>
<td>6840</td>
<td>55.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAMB36E2</td>
<td>791.5</td>
<td>2160</td>
<td>40.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUT Memory</td>
<td>129019</td>
<td>591840</td>
<td>21.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUT as Logic</td>
<td>288463</td>
<td>1182240</td>
<td>24.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Serdes</td>
<td>4</td>
<td>76</td>
<td>5.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCIE40E4</td>
<td>1</td>
<td>6</td>
<td>16.7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Can close timing despite 100% URAM utilization due to simple SRAM-DSP interconnections
Summary

> Make compute fast: Use DSP supertile arrays

> Keep compute busy
  >> Store tensors in SRAM (UltraRAM and BRAM)
  >> Map convolution channels to MxV ports
  >> Configure MxV shapes to maximize pipeline balance factors

> Simplify implementation
  >> Use straight routes. (Recall the flat view.)
  >> Don’t build convoluted memory -compute networks
  >> Use temporal locality to reduce tensor memory read bandwidth
  >> Run the rest of the design at half the DSP clock rate