Highly Performant, Deep Neural Networks with sub-microsecond latency on FPGAs for Trigger Applications

Christian Schmitt (Mainz)
Motivation

• Deep neural networks are widely used for reconstruction and analyses but only few examples exist yet within low-level hardware triggers
  • Tight constraints on data rate and latency
  • E.g. ATLAS L1 Trigger (FPGA based):
    • 40 MHz incoming data rate,
    • <2.5μs overall latency, i.e. \( O(100\text{ns}) \) for inference of DNN
• Our approach:
  • Hardware centric, bottom-up approach for implementation of general neural networks on FPGAs
  • Focus on LHC like conditions: 40MHz data rate and latency of \( O(10)-O(100) \text{ ns} \)
FPGAs ("Field Programmable Gate Array")

- Programmable look-up tables (LUT, \(1.2M\))
- Combinational logic
-Registers (FF, \(2.4M\))
- Bit storage
- Programmable routing
- LUT/register wiring
- Specialized units
  - DSPs (6840 ‘simple ALUs’, MULT w/ subsequent ADD)
- Block memory (~10MB)
- …
- Lots of IO, computation; **predictable, ns-scale latencies**

Image: https://medium.com/@ckyrkou/what-are-fpgas-c9121ac2a7ae

Xilinx US+ XCVU9P-2
Development aims and arithmetics/performance

- **Focus on efficient resource usage**
- No in-depth understanding of implementation required by user (similar to hls4ml, see next talk); easy translation from trained model to VHDL
- **Arithmetics implementation**
  - Fixed point with configurable precision (layer-wise)
  - <16 bits sufficient for DNNs, easier to implement
- **Inference performance limit (theoretical)**
  - DSP for multiply-accumulate (MAC) operations
    - 1 MAC/cycle per DSP
  - Xilinx US+ XCVU9P-2 \( \Rightarrow \) ~5 TMAC/s
    - **LHC data frequency (40 MHz): ~100k - ~150k MAC/event**
- **Support at least the following DNN layers**
  - 2D convolution (image recognition), fully connected, maxpooling
Fully-connected layer design

- Exploit: every neuron needs every input
- Implement neuron processing in **DSP pipelines**
  - Inputs completely reusable
  - Only weight loading/fetching/multiplexing
- **Simple design with easy parallelisation**

Spatial axis (downwards the DSP pipeline)
Fully-connected layer design

- Exploit: every neuron needs every input
- Implement neuron processing in **DSP pipelines**
  - Inputs completely reusable
  - Only weight loading/fetching/multiplexing
  - Simple design with easy parallelisation

![Diagram of neuron processing in DSP pipelines](attachment:image.png)

**Spatial axis** (downwards the DSP pipeline)

**Temporal axis**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>DSP</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>![i0 w0,0]</td>
<td>![i1 w0,1]</td>
<td>![i2 w0,2]</td>
</tr>
<tr>
<td>2</td>
<td>![i0 w1,0]</td>
<td>![i1 w1,1]</td>
<td>![i2 w1,2]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>![i0 w2,0]</td>
<td>![i1 w2,1]</td>
<td>![i2 w2,2]</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>![i0 w3,0]</td>
<td>![i1 w3,1]</td>
<td>![i2, w3,2]</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>![i0 w0,0]</td>
<td>![i1 w0,1]</td>
<td>![i2, w0,2]</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>![i0 w1,0]</td>
<td>![i1 w1,1]</td>
<td>![i2, w1,2]</td>
<td></td>
</tr>
</tbody>
</table>

**Input**

- ![i0]
- ![i1]
- ![i2]

**Dense layer**

- ![o0]
- ![o1]
- ![o2]
- ![o3]
Fully-connected layer design

- Exploit: every neuron needs every input
- Implement neuron processing in **DSP pipelines**
- Inputs completely reusable
- Only weight loading/fetching/multiplexing
- Simple design with easy parallelisation

![Diagram of a fully-connected layer design](image)
**Fully-connected layer design**

- Exploit: every neuron needs every input
- Implement neuron processing in **DSP pipelines**
  - Inputs completely reusable
  - Only weight loading/fetching/multiplexing
  - Simple design with easy parallelisation

![Diagram showing fully-connected layer design and DSP pipelines](image-url)
2D Convolution Layer

- 2D convolution way more difficult to implement
  - Naive implementation would need large amount of resources for multiplexing of inputs/weights
- Optimised approach
  - Use “slices” (channel x width) and “rows” (fixed height and channel) as basic quantities
  - “Row units” yield good compromise of computational efficiency and input/weight reuse
Implementation results: resource usage

Xilinx US+ XCVU9P-2
(6840 DSPs, 2.4M FF, 1.2M LUT)

• Main limitation is number of DSPs
• Fully-connected:
  \[ N_{DSP} \approx N_I \cdot N_N \cdot \frac{f_{Data}}{f_{FPGA}} \]

• 2D-Convolution:
  \[ N_{DSP} \approx V_I \cdot V_K \cdot \frac{f_{Data}}{f_{FPGA}} \]
Implementation results: operating frequency

- Maximum layer frequency depends on resource usage (signal propagation, routing complexity, …)
  - Fully-connected and pooling layers are less complex -> higher frequency
  - Can run at $\geq 400$ MHz even for layers with 10k operations
Network creation toolkit

- Python based toolkit for **automated network creation**
- **Starting point:** trained Keras network
  - Supported layers: Fully-connected, 2D-Conv, Maxpool
  - Activation: relu (best for FPGA)
- Additional design parameters can be specified:
  - Precision (integer and fractional bits)
  - Pipelining and routing behaviour
- **Output:**
  - VHDL code of the corresponding network
# assume all modules already imported
model = load_model(keras_model)

# define extra parameters for the layers
lrExtraData = []
for l in model.layers:
    lrExtraData.append((cycles, parallelization, precBitsV, precBitsW, precBitsV, truncMode_Dense, kwargs))

# Creating the network object
network = Network(name_net, model, name_din, name_dout, name_pkg, lrExtraData, 
                  input_scheme, name_sim, verb = False)

# Show network delay information
print("latencies:", network.computeNetDelay(verb = False))

## Creating the network top VHDL code
code_net_top = network.createNetTopCode()
writeFile(code_net_top, file_net_top)

## Creating the network package VHDL code
code_net_pkg = network.createNetPkgCode()
writeFile(code_net_pkg, file_net_pkg)

## Creating the network sim VHDL code
code_net_sim = network.createNetSimCode(iniFiles, 
                                          file_stim, file_res)
writeFile(code_net_sim, file_net_sim)

# Creating the init files
# (control and weight data for Conv and Dense layers)
network.createSimFiles(iniFiles)
• Successful network implementations up to 15k multiplications for a data frequency of 40 MHz (e.g. LHC)
Results: overall latency

- Latency depends on achievable frequency
- Full network output can be available in ~100ns

\[ C = \frac{f_{FPGA}}{f_{Data}} \]
Summary & Outlook

- Full networks consisting of 2D-Conv, Maxpooling and Fully-connected layers implemented on FPGAs
  - Can cope with LHC data frequencies of 40 MHz, full network latencies of O(100ns)
  - Easy to use python based toolkit for automatic creation of VHDL code from trained Keras model
- Publication: 2019 JINST14 P09014
- Next steps:
  - Implement first physics example network using this toolkit to fit within current LHC Run-3 hardware
  - Extend toolkit to support more layer types and further optimisations on layer implementations
Backup
### Example network architectures

#### Input: 14x14

#### Naming convention:

- 2D-Conv:
  - \((H_K \times W_K \times N_K)\)
- Maxpool:
  - \((H_P \times W_P)\)
- Dense
- \(N_{\text{Neuron}}\)

<table>
<thead>
<tr>
<th>Architecture (see text)</th>
<th>MACs (DSP eff.)</th>
<th>(T_P) (ns)</th>
<th>WNS (ns)</th>
<th>latency (cycles)</th>
<th>(N_{\text{LUT}})</th>
<th>(N_{\text{DSP}})</th>
<th>(N_{\text{BRAM}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{Arc}_{A1} (C = 16)) (input ((7 \times 7)))</td>
<td>334 (0.485)</td>
<td>1.562</td>
<td>-</td>
<td>56</td>
<td>1793</td>
<td>43</td>
<td>10.5</td>
</tr>
<tr>
<td>((2 \times 2 \times 1)-(2 \times 2)-10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Arc}_{A2} (C = 14))</td>
<td>1089 (0.630)</td>
<td>1.786</td>
<td>-</td>
<td>60</td>
<td>5060</td>
<td>108</td>
<td>17</td>
</tr>
<tr>
<td>((2 \times 2 \times 1)-(2 \times 2)-7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Arc}_{A3} (C = 14)) (input ((7 \times 7)))</td>
<td>1024 (0.620)</td>
<td>1.786</td>
<td>-</td>
<td>57</td>
<td>3051</td>
<td>118</td>
<td>19</td>
</tr>
<tr>
<td>((2 \times 2 \times 3)-(2 \times 2)-16)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Arc}_{A4} (C = 13))</td>
<td>3188 (0.774)</td>
<td>1.923</td>
<td>-</td>
<td>63</td>
<td>8689</td>
<td>317</td>
<td>54.5</td>
</tr>
<tr>
<td>((2 \times 2 \times 2)-(2 \times 2)-17)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Arc}_{A5} (C = 13))</td>
<td>7854 (0.967)</td>
<td>1.923</td>
<td>-</td>
<td>68</td>
<td>15567</td>
<td>625</td>
<td>93.5</td>
</tr>
<tr>
<td>((2 \times 2 \times 4)-(2 \times 2)-25)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Arc}_{A6} (C = 11))</td>
<td>12884 (0.894)</td>
<td>2.273</td>
<td>-</td>
<td>68</td>
<td>20962</td>
<td>1310</td>
<td>166</td>
</tr>
<tr>
<td>((3 \times 3 \times 4)-(2 \times 2)-50)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Arc}_{B1} (C = 12))</td>
<td>8858 (0.812)</td>
<td>2.083</td>
<td>-</td>
<td>76</td>
<td>18587</td>
<td>909</td>
<td>99.5</td>
</tr>
<tr>
<td>((2 \times 2 \times 4)-(2 \times 2)-(2 \times 2 \times 4)-25)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Arc}_{B1} (C = 16))</td>
<td>8858 (0.812)</td>
<td>2.083</td>
<td>-</td>
<td>87</td>
<td>17205</td>
<td>713</td>
<td>71.5</td>
</tr>
<tr>
<td>((2 \times 2 \times 4)-(2 \times 2)-(2 \times 2 \times 4)-25)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Arc}_{B3} (C = 11))</td>
<td>11362 (0.792)</td>
<td>2.273</td>
<td>-</td>
<td>79</td>
<td>28383</td>
<td>1305</td>
<td>102.5</td>
</tr>
<tr>
<td>((2 \times 2 \times 6)-(2 \times 2)-(2 \times 2 \times 4)-25)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Arc}_{B2} (C = 10))</td>
<td>15610 (0.855)</td>
<td>2.500</td>
<td>-0.134</td>
<td>84</td>
<td>40998</td>
<td>1825</td>
<td>68</td>
</tr>
<tr>
<td>((3 \times 3 \times 6)-(2 \times 2)-(3 \times 3 \times 6)-25)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Arc}_{B3} (C = 16))</td>
<td>11362 (0.825)</td>
<td>1.562</td>
<td>-0.014</td>
<td>93</td>
<td>26006</td>
<td>861</td>
<td>71.5</td>
</tr>
</tbody>
</table>
- Use **multiple but shorter pipelines** with additional adder in parallel ("neuron unit") **to reduce latency**
2D-Convolution: Firmware implementation

All cases:

Regular case:
Maxpooling layer

- All inputs are only needed once
  - no way of saving resources or input accesses
  - no need to use complicated row allocation patterns
- For simplicity reasons, the concept of output rows and row units was still maintained

ErUM-Data

Christian Schmitt
18/13
schmittc@uni-mainz.de
Activation function

- RELU activation:
  - Resource usage: B/2 LUTs or (B-1) FFs for B bit values
  - Any other activation could be implemented using value-derivative lookup tables
- Example for tanh and sigmoid with 16 sample points:

![Graph of linearly approximated activations]

- Characteristics like base value and derivative saturation could be further exploited for an even decreased LUT requirement.
Network MACs assuming LHC Data Rate of 40MHz