Compilers and Beyond: Research towards enhancing the design productivity for FPGA ML applications

Daniel Holanda Noronha and Steve Wilton
University of British Columbia, Vancouver, Canada
danielhn@ece.ubc.ca, stevew@ece.ubc.ca
A compiler is not enough: Engineers expect a complete ecosystem to design complex machine learning circuits.

- Ongoing research project towards such an ecosystem
Overview of today's talk

• Introduction - Using FPGAs for ML
• LeFlow - Going from Tensorflow to Verilog
  ○ General flow of our tool-kit
  ○ Tuning performance
  ○ Examples, Limitations and Opportunities
• On-chip debug of ML circuits
  ○ Existing debug flows
  ○ Creating specialized instruments for debugging ML circuits
Introduction - Using FPGAs for Machine Learning
Deep learning has emerged as an important application area for FPGAs
- Often faster than software and less power than GPU
- Cloud computing -> More designers to take advantage of FPGAs

Microsoft Catapult
- Started with 1,600 FPGA-enabled servers (2014)
- Today: Hundreds of thousands of FPGAs (15 countries, 5 continents)
- Project Brainwave
  - Offers real-time AI serving in the cloud
  - Pre-trained DNN models with high efficiencies at low batch sizes
Problem with FPGAs:
- Designing such applications is challenging
- Not many people can do it
Introduction - Using FPGAs for Machine Learning

Design flow for an FPGA machine learning accelerator

- Step 1 - Software model implemented using high-level framework
  - Abstraction of implementation details
  - Understand the required network size, convergence rate, etc.

- Step 2 - Map the network to a hardware implementation.
  - Often done manually, by writing C or RTL code
  - Time consuming and requires hardware design expertise

Our solution (research prototype):

LeFlow: FPGA High-Level Synthesis of Tensorflow Deep Neural Networks
LeFlow - Going from Tensorflow to Verilog
LeFlow

- Uses Google’s XLA compiler which emits LLVM code directly from Tensorflow
- LLVM code transferred to HLS tool to automatically generate hardware
- Allows rapid prototype of machine learning algorithms on FPGAs
- Not as efficient as hand-optimized hardware design
  - Compelling for a large number of design scenarios
  - May open the door for hardware acceleration to many domain experts
- Demonstrated using LegUp, but may be suitable for many other HLS tools
LeFlow

- Completely open-source
- Available on GitHub

LeFlow is an open-source tool-flow that maps numerical computation models written in Tensorflow to synthesizable hardware. Our flow bridges Google’s XLA compiler LegUp high-level synthesis tool to automatically generate verilog from a Tensorflow specification.

LeFlow Tool-kit - Overall Flow

- Open source library for numerical computation
- Nodes represent mathematical operations, edges represent tensors
- Extensive support for Deep Learning algorithms
- XLA: a domain-specific compiler for linear algebra

- Open source HLS tool developed at the University of Toronto
- LegUp can synthesize most of the C language to hardware
- Uses LLVM compiler infrastructure
LeFlow Tool-kit - Overall Flow

• The user creates a design in Python using Tensorflow

• Use XLA compiler to generate an LLVM intermediate representation (IR)

• LeFlow performs several transformations to the IR (will be described soon)

• LLVM IR can then be read as an input to a HLS tool, which generates a hardware description in Verilog.
• Handling inputs and outputs

  ○ XLA generates IR that is written in a software-like way

  ○ LeFlow remaps this IR to make it more suitable for generating hardware

  ○ Inputs to the network are stored in on-chip memory

  ○ LeFlow also takes special care to avoid those memories from being optimized away
LeFlow Tool-kit - Handling Unsupported Kernels

- Each particular Tensorflow operation can be transformed to IR in many different ways.
- The best way to generate IR depends on the several factors.
- XLA handles this design problem by implementing multiple kernels for a single operation and selecting them according to the problem at compile time.
LeFlow Tool-kit - Handling Unsupported Kernels

• Problem: Not all kernels implemented in Tensorflow can be directly mapped to our version of LegUp

• LeFlow avoids unsupported XLA kernels through the use of flags added to Tensorflow
  ○ Decision abstracted away from the user
LeFlow Tool-kit - Other transformations

• Optimization passes
  ○ XLA can emit both optimized (with O3) or unoptimized IR
  ○ Some optimizations might
    ■ Drastically change the way in which variables are addressed, making it hard to identify inputs and outputs
    ■ Generate IR instructions not supported by the HLS tool
  ○ LeFlow uses unoptimized IR and has its own tailored optimization recipe to avoid those problems

• LLVM Version Issues
  ○ LegUp uses LLVM 3.5.0, while Tensorflow uses LLVM 7.0
  ○ LeFlow performs transformations to address these differences
Tuning Performance - Compiler Optimizations

- Compiler optimizations can have a significant impact on the final hardware design.

- Unrolling and Inlining offer tradeoff between area and latency:
  
  \[
  \begin{array}{c|c|c}
  \uparrow \text{Unrolling} & \uparrow \text{area} & \downarrow \text{cycles} \\
  \uparrow \text{Inlining} & \uparrow \text{area} & \downarrow \text{cycles} \\
  \end{array}
  \]

- LeFlow enables the user to optionally tune both unrolling and inlining thresholds at the Python level.

  ```python
  options.setUnrollThreshold(150)
  options.setInliningThreshold(500)
  ```
A common performance bottleneck in any parallel implementations is the memory
- Dual-port RAMs -> Only two reads/writes per cycle

FPGAs contain a vast number of independently accessible memories
- It is good to split big arrays into multiple memories

Memory partitioning is not part of LegUp 4.0, so LeFlow implements its own version of this transformation pass.

This pass is performed at the LLVM IR level, but configured at the user’s python code.
Examples - MLP and MNIST digit recognition

• In this example
  ○ MLP followed by a softmax is trained offline in Tensorflow using XLA
  ○ LeFlow-generated hardware is deployed in an FPGA for inference

• The example including the training phase with XLA is part of the LeFlow distribution.
Examples - MLP and MNIST digit recognition

```
1 import tensorflow as tf
2 import numpy as np
3 input = tensorflow.placeholder(tensorflow.float32, shape=[None, 784])
4 weights = tensorflow.placeholder(tensorflow.float32, shape=[784, 10])
5 bias = tensorflow.placeholder(tensorflow.float32, shape=[10])
6 with tf.Session() as sess:
7   session.run(tensorflow.global_variables_initializer())
8   with tf.device("device:XLA_CPU:0"):
9       y = tensorflow.nn.softmax(tensorflow.add(tensorflow.matmul(input, weights)[0], bias))
10      session.run(y, {input: MNIST_digit_to_classify, weights: desired_weights, bias: desired_bias})
```
Examples - Convolutional Network

- In this example a CNN with 1 input and 5 outputs is compiled to hardware using LeFlow.
- Image shows the result of the CNN when specific 3x3 filters are used as the weights of the network.
Examples - Convolutional Network

• It is unreasonable to fit an entire image and weights in the internal memory of an FPGA
  ○ Common practice: split the image in tiles and process it over multiple batches
• In this example, each input and output has 32x32 pixels

```python
1 import tensorflow as tf
2 import numpy as np
3 inputs = tf.placeholder(tf.float32, [1, 1, 1, 1])
4 weights = tf.placeholder(tf.float32, [3, 3, 1, 5])
5 with tf.Session() as sess:
6   sess.run(tf.global_variables_initializer())
7   with tf.device("device:XLA_CPU:0"):
8     y = tf.nn.conv2d(inputs, weights, strides=[1, 1, 1, 1], padding='SAME')
9     result = sess.run(y, {inputs: original_image, weights: desired_filters})
```

<table>
<thead>
<tr>
<th>Circuit</th>
<th>LEs</th>
<th>MemB</th>
<th>FMax</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>2,291</td>
<td>198,048</td>
<td>149.59 MHz</td>
<td>1,449,734</td>
</tr>
<tr>
<td>Unrolled</td>
<td>2,682</td>
<td>198,048</td>
<td>186.36 MHz</td>
<td>1,275,700</td>
</tr>
</tbody>
</table>
Examples

- More interesting examples available on our GitHub repository
Benchmarking Individual Layers

<table>
<thead>
<tr>
<th></th>
<th>LEs</th>
<th>MemB</th>
<th>FMax</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>vecmul_a</td>
<td>661</td>
<td>768</td>
<td>301.93</td>
<td>123</td>
</tr>
<tr>
<td>vecmul_b</td>
<td>664</td>
<td>6,144</td>
<td>289.69</td>
<td>963</td>
</tr>
<tr>
<td>vecmul_b_u</td>
<td>2,346</td>
<td>6,144</td>
<td>228.78</td>
<td>98</td>
</tr>
<tr>
<td>dense_a</td>
<td>1,743</td>
<td>1,056</td>
<td>267.45</td>
<td>380</td>
</tr>
<tr>
<td>dense_b</td>
<td>1,749</td>
<td>8,224</td>
<td>291.21</td>
<td>3,012</td>
</tr>
<tr>
<td>softmax_a</td>
<td>7,209</td>
<td>960</td>
<td>203.54</td>
<td>902</td>
</tr>
<tr>
<td>softmax_b</td>
<td>7,206</td>
<td>6,336</td>
<td>206.31</td>
<td>7,174</td>
</tr>
<tr>
<td>softmax_b_u</td>
<td>21,688</td>
<td>6,336</td>
<td>135.72</td>
<td>4,708</td>
</tr>
<tr>
<td>conv2d_a</td>
<td>2,286</td>
<td>6,720</td>
<td>165.23</td>
<td>32,187</td>
</tr>
<tr>
<td>conv2d_a_u</td>
<td>63,430</td>
<td>6,720</td>
<td>47.70</td>
<td>1,784</td>
</tr>
<tr>
<td>conv2d_b</td>
<td>2,289</td>
<td>393,792</td>
<td>152.32</td>
<td>2,370k</td>
</tr>
<tr>
<td>maxp_a</td>
<td>981</td>
<td>2,176</td>
<td>221.43</td>
<td>229</td>
</tr>
<tr>
<td>maxp_b</td>
<td>979</td>
<td>35,968</td>
<td>219.25</td>
<td>5,533</td>
</tr>
<tr>
<td>maxp_b_u</td>
<td>59,346</td>
<td>35,968</td>
<td>160.93</td>
<td>502</td>
</tr>
<tr>
<td>thxprls_g</td>
<td>18,520</td>
<td>704</td>
<td>185.22</td>
<td>4675</td>
</tr>
</tbody>
</table>

- LeFlow comes with an automated test script to run multiple small components.
- These components represent building blocks needed to create a deep neural network.
- Simulation times of different blocks vary from seconds to hours.
- Especially useful for those in the community who wish to build upon and expand this tool.
Quality of results

- Not as efficient as hand-optimize RTL designs but flexible
  - It is easy to add support for new operations
  - LeFlow should be expanded to support blocks optimized for single workload acceleration (e.g. CNN overlays of MlSuite)

- Ongoing work towards evaluating the quality of the results of circuits generated by LeFlow
Limitations and Opportunities

1. LeFlow currently uses kernels meant to be used by CPUs
   ○ Compiler optimizations and scheduling are able to retrieve a substantial amount of parallelism
   ○ LeFlow would heavily benefit from an XLA back-end with kernels for FPGAs

2. Automatic memory partitioning for ML.
   ○ The high dimensionality of inputs/weights and the amount of parallel accesses in ML applications is a challenge
   ○ LeFlow would specially benefit from a machine learning specific automatic memory partitioning algorithm
Limitations and Opportunities

3. Using a customizable fixed-point bit width
   ○ Adding fixed-point support
   ○ Automatically profile the application choose the appropriate representation

4. Debug Infrastructure
   ○ It is straightforward to use Tensorflow to debug the functionality of an implementation
   ○ Difficult for software developers to debug the generated hardware in terms of the original Python code
   ○ A performance debugging infrastructure suitable for software developers is another interesting venue for research.
On-chip debug of Machine Learning Circuits
On-chip debug

- Records the behaviour of the design as it runs at speed for later interrogation

- Challenge:
  - Record enough information on-chip to understand the problem
**Why focus on on-chip debug?**

### Kernel-level bugs
- Self-contained
- Debug in isolation
- Easy to reproduce

Debug code on workstation (gdb, pdb, tensorboard).

### RTL-level bugs
- Framework/RTL mismatch
- Framework tool errors or usage errors

Run co-simulation on workstation.

### System-Level Bugs
- Bugs in interfaces
- Dependent on:
  - I/O data patterns
  - Interaction timing
- Hard to reproduce, or require long run times

Debug on FPGA (Requires observing internals of FPGA)

**These are the difficult bugs**
Key observation

To find certain bugs we must debug in hardware
Why not use common on-chip hardware debug tools?

Embedded Logic Analyzer (Altera SignalTap II):

Your RTL Circuit → Debug Tool:
- Chooses signals to trace
- Debug circuitry added

Run

- RTL-level debug is not suitable for debugging applications designed at a high level of abstraction
  - Understanding the hardware is time consuming
  - RTL looks nothing like the original description due to compiler optimizations
  - Beyond the expertise of software developers
Debug levels of abstraction

- Hardware-oriented debug
- HLS-oriented debug
- ML-specific debug
On-chip debug for HLS

Capture system-level bugs → Need to run at-speed, on-chip

Solution: Record and Replay

1. User selects variables, tool determines signals, inserts instrumentation

2. Compile

3. Execute and record

4. Stop and retrieve

5. Software-like debug using recorded data

```c
void qSort(int *arr) {
    int piv, beg[N], end[N];
    int i=0;
    int L, R, swap;
    ...
}
```
How we used to do it
HLS Debug - Efficiency

- Signals are recorded according to selected signals and the HLS schedule
- Recorded signals change each cycle
- Circuit-by-Circuit custom compression
- 50x-100x more memory efficient than traditional hardware-oriented debug
Debug levels of abstraction

- Hardware-oriented debug
- HLS-oriented debug
- ML-specific debug
On-chip debug of Machine Learning Circuits

A flow to accelerate the debug of machine learning applications on FPGAs

- Previous work is not ideal for debugging ML circuits
  - Even longer run-times; “Correctness” hard to determine; Commonly designed at a high level.

- This work uses domain-specific characteristics of ML circuits to create instruments that:
  - Maximize the utilization of trace buffer space
  - Provide information that is meaningful to an engineer
Overview of our instruments

- Many machine learning applications consist of large arrays (e.g., activations or weights)
- Instruments track large arrays over time

Distribution Instrument

- Creates a history of the distribution of the matrix we are observing over time (over multiple frames)
- In a CNN, a frame may represent all calculations corresponding to a single input image
**Debug Instruments**

**Spatial Sparsity Instrument**

- Stores an indication whether each element of the array is zero or non-zero.
- The same logic could also be used to track elements close to 1, another upper bound or NaN.

**Summary Statistics Instrument**

- Tracks only one kind of statistic (sparsity, mean, std. dev) per frame.
Main Differences

- Stepping through frames instead of stepping through clock cycles (hardware-oriented debug) or lines of C code (HLS-debug)
- No access to raw values, we can trace the circuit for a longer period
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Kernel</th>
<th>FMax (MHz)</th>
<th>LEs</th>
<th># Traced Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Previous work</strong> (HLS-oriented debug)</td>
<td>32x28x28</td>
<td>213.79</td>
<td>3391</td>
<td>0.124</td>
</tr>
<tr>
<td></td>
<td>8x28x28</td>
<td>260.05</td>
<td>3324</td>
<td>0.498</td>
</tr>
<tr>
<td></td>
<td>1x28x28</td>
<td>287.89</td>
<td>3167</td>
<td>3.985</td>
</tr>
<tr>
<td><strong>Distribution Instrument - 32 bins</strong></td>
<td>32x28x28</td>
<td>200.48</td>
<td>2867</td>
<td>195</td>
</tr>
<tr>
<td></td>
<td>8x28x28</td>
<td>227.65</td>
<td>2834</td>
<td>223</td>
</tr>
<tr>
<td></td>
<td>1x28x28</td>
<td>229.87</td>
<td>2676</td>
<td>284</td>
</tr>
<tr>
<td><strong>Distribution Instrument - 128 bins</strong></td>
<td>32x28x28</td>
<td>189.62</td>
<td>3670</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>8x28x28</td>
<td>225.17</td>
<td>3600</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>1x28x28</td>
<td>228.98</td>
<td>3488</td>
<td>71</td>
</tr>
<tr>
<td><strong>Spatial Sparsity Instrument</strong></td>
<td>32x28x28</td>
<td>200.46</td>
<td>2547</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>8x28x28</td>
<td>211.13</td>
<td>2531</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>1x28x28</td>
<td>214.70</td>
<td>2393</td>
<td>127</td>
</tr>
<tr>
<td><strong>Summary Statistics Instrument - Sparsity</strong></td>
<td>32x28x28</td>
<td>213.17</td>
<td>2557</td>
<td>6666</td>
</tr>
<tr>
<td></td>
<td>8x28x28</td>
<td>258.75</td>
<td>2531</td>
<td>7692</td>
</tr>
<tr>
<td></td>
<td>1x28x28</td>
<td>285.30</td>
<td>2390</td>
<td>10000</td>
</tr>
<tr>
<td><strong>Proposed instruments combined</strong></td>
<td>32x28x28</td>
<td>189.23</td>
<td>2930</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>8x28x28</td>
<td>206.69</td>
<td>2927</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>1x28x28</td>
<td>220.51</td>
<td>2786</td>
<td>87</td>
</tr>
</tbody>
</table>
## Results

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Kernel</th>
<th>FMax (MHz)</th>
<th>LEs</th>
<th># Traced Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous work  (HLS-oriented debug)</td>
<td>32x28x28</td>
<td>213.79</td>
<td>3391</td>
<td>0.124</td>
</tr>
<tr>
<td></td>
<td>8x28x28</td>
<td>260.05</td>
<td>3324</td>
<td>0.498</td>
</tr>
<tr>
<td></td>
<td>1x28x28</td>
<td>287.89</td>
<td>3167</td>
<td>3.985</td>
</tr>
<tr>
<td>Distribution Instrument - 32 bins</td>
<td>32x28x28</td>
<td>200.48</td>
<td>2867</td>
<td>195</td>
</tr>
<tr>
<td></td>
<td>8x28x28</td>
<td>227.65</td>
<td>2834</td>
<td>223</td>
</tr>
<tr>
<td></td>
<td>1x28x28</td>
<td>229.87</td>
<td>2676</td>
<td>284</td>
</tr>
<tr>
<td>Distribution Instrument - 128 bins</td>
<td>32x28x28</td>
<td>189.62</td>
<td>3670</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>8x28x28</td>
<td>225.17</td>
<td>3600</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>1x28x28</td>
<td>228.98</td>
<td>3488</td>
<td>71</td>
</tr>
<tr>
<td>Spatial Sparsity Instrument</td>
<td>32x28x28</td>
<td>200.46</td>
<td>2547</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>8x28x28</td>
<td>211.13</td>
<td>2531</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>1x28x28</td>
<td>214.70</td>
<td>2393</td>
<td>127</td>
</tr>
<tr>
<td>Summary Statistics Instrument - Sparsity</td>
<td>32x28x28</td>
<td>213.17</td>
<td>2557</td>
<td>6666</td>
</tr>
<tr>
<td></td>
<td>8x28x28</td>
<td>258.75</td>
<td>2531</td>
<td>7692</td>
</tr>
<tr>
<td></td>
<td>1x28x28</td>
<td>285.30</td>
<td>2390</td>
<td>10000</td>
</tr>
<tr>
<td>Proposed instruments combined</td>
<td>32x28x28</td>
<td>189.23</td>
<td>2930</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>8x28x28</td>
<td>206.69</td>
<td>2927</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>1x28x28</td>
<td>220.51</td>
<td>2786</td>
<td>87</td>
</tr>
</tbody>
</table>

**Takeaway:**

Domain-specific instrumentation allow us to store more **useful information** on-chip.
Architecture Study

Distribution Instrument

• In this experiment, we vary the number of bins while number of histograms remains the same.

• **Frequency** drops as the number of bins increases, however, the impact is less than 5% when using 64 bins and 64 frames.

• **Area** and **memory bits** grow linearly with the number of bins
Architecture Study

Spatial Sparsity Instrument

• In this experiment, we vary the number of frames traced while keeping the size of each frame constant, for several kernels.

• **Frequency** has not changed for most cases.

• Approximately same initial **area** overhead all circuits that does not increase with the trace size.

• **Memory bits** grows linearly with the number of frames traced.
Future Work

• Making instrumentation configurable at debug time
  ○ FPGA synthesis is very slow -> Debug cycles are slow
  ○ Important for scenarios in which FPGA cannot be turned off

• Adapting this infrastructure debug multiple FPGAs are on a single task
  ○ Not practical to have one USB JTAG on each FPGA;
  ○ Project Brainwave

• Combining this domain-specific instrumentation with general-purpose debug tools
  ○ Domain-specific -> Coarse-grained view of circuit for long period
  ○ General-purpose -> Fine-grained view of circuit for short period
Final remarks
Final remarks

- **A compiler is not enough**: Engineers expect a complete ecosystem to design complex machine learning circuits.

- So far, we explored:
  - Compiler (LeFlow)
    - Allows software developers without hardware expertise to implement Deep Neural Networks in FPGAs using Tensorflow.
  - On-chip debug of ML applications
    - By specializing the debug instrumentation we store more useful information on the chip
Final remarks

What is next?

- More integration between components of our ecosystem
- Better use of scheduling information, not only for debug, but why not for power
- More specialized solutions for application-specific problems