MICRO’16 Tutorial on Rapid Exploration of Accelerator-rich Architectures

Rapid Hardware Specialization with HLS: Glass Half Full?

Zhiru Zhang

School of ECE, Cornell University
October 2016
Best of Times to Work on Hardware Accelerators

- **Pressing demand** to efficiently accelerate a growing array of datacenter & embedded workloads
  - Multicore performance scaling significantly slowed
  - Pervasive hardware specialization is inevitable?

---

**Google Tensor Processing Unit**

**Microsoft Project Catapult**
Best of Times to Work on Hardware Accelerators

- **Substantial leverage** readily available for building realistic specialized computing systems
  - Open-source hardware movement (e.g., RISC-V)
  - Open-source compiler infrastructure and architecture simulator (e.g., LLVM, gem5)
  - FPGA SoCs have come of age

![Diagram of Xilinx Zynq SoC and RISC-V + Accelerator](image-url)
“Worst of Times” Also?

- Target of specialization is constantly moving, and moving fast!
  - In particular, deep learning algorithms are changing every few days
  - Tooling and software frameworks are also rapidly evolving

Is RTL design method still viable?

Introduction

Deep Neural Networks (DNN) have substantially pushed the state-of-the-art in a wide range of tasks, especially in speech recognition [1, 2] and computer vision, notably object recognition from images (Krizhevsky et al., 2012; Szegedy et al., 2014), speech recognition (Hinton et al., 2012; Sainath et al., 2013), and natural language processing (Devlin et al., 2014; Sutskever et al., 2014). In this paper, we address another important problem in the realm of DNNs: the large-scale image recognition setting.

We trained a large, deep convolutional neural network to classify the 1.2 million images in the ImageNet classification track of the ILSVRC 2012 challenge. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.2%, respectively, compared to 26.2% achieved by the second-best entry.

There are many reasons for the success of DNNs. First, their layered structure is particularly well-suited to the hierarchical organization of visual data. Second, they can automatically learn relevant features from raw data, which may not be obvious to a human expert. Finally, they are trained using backpropagation, an algorithm that efficiently computes gradients in deep networks.

However, the immense complexity of the object recognition task means that this problem requires a large amount of training data. For example, the ImageNet dataset consists of hundreds of thousands of fully-segmented images, and it is only recently become possible to collect such large amounts of data.

But objects in realistic settings exhibit considerable variability, so to learn to recognize them it is necessary to use much larger training sets. And indeed, the shortcomings of small image datasets have been widely recognized (e.g., Pinto et al., 2014), but it has only recently become possible to collect such large amounts of data.

To prove their performance, we can collect larger datasets, learn more powerful models, and use better algorithms and applications. For instance, the best-performing algorithm to date is a deep convolutional neural network trained on 13 million images with a training time of 15 days.

In this work we investigate the effect of the convolutional filter size on the performance of the network. To reduce overfitting in the fully-connected layers, we used non-saturating neurons and a very efficient GPU implementation.

We trained the network for 160 epochs, and achieved top-1 and top-5 error rates of 39.8% and 19.7%, respectively, compared to 26.2% achieved by the second-best entry.

Finally, we hope that this work will inspire further research in this area.

Abstract

We introduce a method to train Binarized Neural Networks (BNNs) - neural networks with binary weights and activations at run-time and when computing the parameters' gradients at train-time. The network is then presented in Section 3, and the learning process is described in Section 4.

The code for training and running our BNNs is available.

Today, DNNs are almost exclusively trained on one or many very fast and power-hungry Graphic Processing Units (GPUs) (Coates et al., 2013). As a result, it is often a challenge to run DNNs on target low-power devices, and much research work is done to speed-up DNNs at run-time on both general-purpose (Vanhoucke et al., 2011; Gong et al., 2014; Romero et al., 2014; Han et al., 2015) and specialized computer hardware (Fubert et al., 2014a,b; Pham et al., 2012; Chen et al., 2014a,b; Esser et al., 2015).

We believe that the contributions of our article are the following:

- We introduce a method to train Binarized Neural Networks (BNNs), which are neural networks with binary weights and activations at run-time and when computing the parameters' gradients at train-time (see Section 1).
- We show how to quantize the parameters and activations to binary values using a simple rounding procedure.
- We introduce BinaryConnect, a method which consists in training a DNN with binary weights and activations at run-time and when computing the parameters' gradient at train-time (see Section 1).
- We introduce a method to quantize the parameters and activations to binary values using a simple rounding procedure.
- We introduce BinaryConnect, a method which consists in training a DNN with binary weights and activations at run-time and when computing the parameters' gradient at train-time (see Section 1).

Acknowledgements

We thank the anonymous reviewers for their insightful comments and suggestions. We also thank the members of the Neural Networks group at the University of Toronto for their support and encouragement.
Motivation for High-Level Synthesis (HLS)

### RTL Verilog

```verilog
module dut(rst, clk, q);
    input rst;
    input clk;
    output q;
    reg [7:0] c;

    always @(posedge clk) begin
        if (rst == 1b'1) begin
            c <= 8'b00000000;
        end
        else begin
            c <= c + 1;
        end
    end

    assign q = c;
endmodule
```

### Untimed C code

```c
uint8 dut() {
    static uint8 c;
    c += 1;
}
```

### HLS

An 8-bit counter
# HLS: From Untimed to Timed

<table>
<thead>
<tr>
<th>Control-Data Flow Graph</th>
<th>Latency</th>
<th>Area</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Untimed Diagram]</td>
<td>![Combinational Diagram]</td>
<td>![Sequential Diagram]</td>
<td>![Pipelined Diagram]</td>
</tr>
</tbody>
</table>

- **Untimed**: Combinational
- **Sequential**: Sequential
- **Pipelined**: Pipelined
HLS Interface Synthesis

- A GCD example with handshaking interface
  - HLS greatly simplifies interface design

C code

```c
void GCD ( msg& req, 
        msg& resp ) {
    short a = req.msg_a;
    short b = req.msg_b;
    while ( a != b ) {
        if ( a > b )
            a = a - b;
        else
            b = b - a;
    }
    resp.msg = a;
}
```

Manual RTL (partial)

```verilog
module GcdUnitRTL
(
    input wire [ 0:0] clk,
    input wire [ 31:0] req_msg,
    output wire [ 0:0] req_rdy,
    input wire [ 0:0] req_val,
    input wire [ 0:0] reset,
    output wire [ 15:0] resp_msg,
    input wire [ 0:0] resp_rdy,
    output wire [ 0:0] resp_val
);
```

always @ (*) begin
    if ((curr_state__0 == STATE_IDLE))
        if (req_val)
            next_state__0 = STATE_CALC;
    if ((curr_state__0 == STATE_CALC))
        if (!is_a_lt_b && is_b_zero)
            next_state__0 = STATE_DONE;
    if ((curr_state__0 == STATE_DONE))
        if ((resp_val && resp_rdy))
            next_state__0 = STATE_IDLE;
end

State transition

always @ (*) begin
    if ((curr_state__0 == STATE_IDLE))
        if (req_val)
            req_rdy = 1;
            resp_val = 0;
            a_mux_sel = A_MUX_SEL_IN;
            b_mux_sel = B_MUX_SEL_IN;
            a_reg_en = 1;
            b_reg_en = 1;
    if ((curr_state__0 == STATE_CALC))
        if (!is_a_lt_b && is_b_zero)
            req_rdy = 0;
            resp_val = 0;
            a_mux_sel = do_swap ? A_MUX_SEL_B : A_MUX_SEL_SUB;
            a_reg_en = 1;
            b_reg_en = do_swap;
            b_mux_sel = B_MUX_SEL_A;
    else
        if ((curr_state__0 == STATE_DONE))
            req_rdy = 0;
            resp_val = 1;
            a_mux_sel = A_MUX_SEL_X;
            b_mux_sel = B_MUX_SEL_X;
            a_reg_en = 0;
            b_reg_en = 0;
end

Output logic

```c
void GCD ( msg& req, 
        msg& resp ) {
    short a = req.msg_a;
    short b = req.msg_b;
    while ( a != b ) {
        if ( a > b )
            a = a - b;
        else
            b = b - a;
    }
    resp.msg = a;
}
```
Case Study: Convolutional Neural Networks (CNNs)

- Typical CNN architecture:
  - **Convolutional** (conv) layers in the front
  - **Fully connected** (dense) layers in the back
  - **Pooling** layers reduce the size of intermediate images (feature maps)

- CNNs have enormous computational and memory requirements
  - **VGG-19 Network** (ImageNet ILSVRC2014): 140 million floating-point (FP) parameters and 15 billion FP operations to classify one image [1]
  - 90+% of the computation is in the conv layers, 90+% of the model size is in the conv and dense weights (~528 MB)

Binarized Neural Networks (BNNs)

Key Differences
1. Input fmaps to conv/dense layers are binarized (-1 or +1)
2. Conv/dense weights are binarized (-1 or +1)
3. Output feature maps are binarized (after some other steps)

Benefits for Hardware
- Binarized weights greatly reduce total model size
- Conv/Dense FP ops replaced with binary logic ops
- Complex non-linearities (tanh, sigmoid) replaced with Sign function

Cifar10 BNN Architecture

- Target dataset
  - CIFAR-10: 60000 32x32 color images
  - 10 classes consisting of animals and vehicles

- Architecture
  - 6 conv, 3 dense, 3 max pooling layers
  - First conv layer’s input is floating-point image

- Software model
  - Python code from [3] available on Github

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input Fmaps</th>
<th>Output Fmaps</th>
<th>Output Dim</th>
<th>Output Bits</th>
<th>Weight Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv1</td>
<td>3</td>
<td>128</td>
<td>32</td>
<td>131K</td>
<td>4480</td>
</tr>
<tr>
<td>Conv2</td>
<td>128</td>
<td>128</td>
<td>32</td>
<td>131K</td>
<td>148K</td>
</tr>
<tr>
<td>Pool</td>
<td>128</td>
<td>128</td>
<td>16</td>
<td>33K</td>
<td></td>
</tr>
<tr>
<td>Conv3</td>
<td>128</td>
<td>256</td>
<td>16</td>
<td>66K</td>
<td>297K</td>
</tr>
<tr>
<td>Conv4</td>
<td>256</td>
<td>256</td>
<td>16</td>
<td>66K</td>
<td>593K</td>
</tr>
<tr>
<td>Pool</td>
<td>256</td>
<td>256</td>
<td>8</td>
<td>16K</td>
<td></td>
</tr>
<tr>
<td>Conv5</td>
<td>256</td>
<td>512</td>
<td>8</td>
<td>33K</td>
<td>1.2M</td>
</tr>
<tr>
<td>Conv6</td>
<td>512</td>
<td>512</td>
<td>8</td>
<td>33K</td>
<td>2.4M</td>
</tr>
<tr>
<td>Pool</td>
<td>512</td>
<td>512</td>
<td>4</td>
<td>8192</td>
<td></td>
</tr>
<tr>
<td>FC1</td>
<td>8192</td>
<td>1024</td>
<td>1</td>
<td>1024</td>
<td>8.4M</td>
</tr>
<tr>
<td>FC2</td>
<td>1024</td>
<td>1024</td>
<td>1</td>
<td>1024</td>
<td>1.0M</td>
</tr>
<tr>
<td>FC3</td>
<td>1024</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>10K</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td>14.2M</td>
<td></td>
</tr>
<tr>
<td>Conv</td>
<td></td>
<td></td>
<td></td>
<td>4.59M</td>
<td></td>
</tr>
<tr>
<td>FC</td>
<td></td>
<td></td>
<td></td>
<td>9.46M</td>
<td></td>
</tr>
</tbody>
</table>

Hardware-Optimized BNN Model

**Changes**

1. Removed biases (they had little effect on accuracy)
2. Simplified batch norm calculation
3. Quantized input image and batch norm parameters

**Accuracy Impact**

<table>
<thead>
<tr>
<th>BNN Model</th>
<th>Test Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Claimed in paper [2]</td>
<td>11.40%</td>
</tr>
<tr>
<td>Python out-of-the-box [2]</td>
<td>11.58%</td>
</tr>
<tr>
<td>C++ optimized model</td>
<td>11.34%</td>
</tr>
<tr>
<td>Accelerator</td>
<td>11.34%</td>
</tr>
</tbody>
</table>

Challenges of BNN Acceleration

1. How to exploit many sources of available parallelism
   (1) across input maps (2) across output maps
   (3) within a map (4) within a filter

2. Resource-efficient design uses a single module for all layers, and must handle different-sized input feature maps

3. Binarized data requires parallelism at the sub-word level (unique to BNN)
BNN Accelerator Architecture

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Our Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many diverse sources of parallelism</td>
<td>Flexible architecture with parameterized numbers of input and output streams ($f_{in}, f_{out}$)</td>
</tr>
<tr>
<td>Design must handle different-sized input maps</td>
<td>Novel hardware structures <strong>Bitsel</strong> and <strong>variable-width line buffer</strong> ensure pipeline is fully utilized regardless of input map width</td>
</tr>
<tr>
<td>Design must exploit sub-word level parallelization</td>
<td>Primary convolution pipeline processes data word-by-word, not pixel-by-pixel</td>
</tr>
</tbody>
</table>
HLS Code in C++

- **User writes and tests in a productive high-level language**
  - Architectural exploration through adding tool directives
  - CPU-FPGA interface automatically synthesized
  - Significant reduction in verification time
    - Full BNN would take days to simulate at RTL level
    - C++ execution finishes in seconds

- **Design effort**
  - Entire software-hardware implementation and testing performed by one student in 10 weeks

```c++
1 VariableLineBuffer linebuf;
2 ConvWeights wts;
3 IntegerBuffer outbuf;
4
5 for (i = 0; i < n_input_words; i++) {
6   #pragma HLS pipeline
7
8   // read input word, update linebuffer
9   WordType word = input_data[i];
10  BitSel(linebuf, word, input_width);
11
12  // update the weights each time we begin to process a new fmap
13  if (i % words_per_fmap == 0)
14     wts = weights[i / words_per_fmap];
15
16  // perform conv across linebuffer
17  for (c = 0; c < LINE_BUF_COLS; c++) {
18    #pragma HLS unroll
19    outbuf[i % words_per_fmap][c] +=
20    conv(c, linebuf, wts);
21  }
22 }
23 ```
FPGA Implementation Results

1. Target platforms
   - **CPU**: Intel Xeon E5-2640 multicore processor
   - **GPU**: NVIDIA Tesla K40 GPU
   - **mGPU**: Jetson TK1 embedded GPU board
   - **FPGA**: ZedBoard with Xilinx Zynq-7000

2. Performance Comparison
   - FPGA is 14x faster than embedded GPU
   - ~6x higher performance/watt over server-class GPU
   - Projected ASIC performance is 10x higher than FPGA

<table>
<thead>
<tr>
<th>Execution time per image (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
</tr>
<tr>
<td>Conv1</td>
</tr>
<tr>
<td>Conv2-5</td>
</tr>
<tr>
<td>FC1-3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
</tr>
<tr>
<td><strong>Speedup</strong></td>
</tr>
<tr>
<td>Power (Watt)</td>
</tr>
<tr>
<td>imgs/sec/Watt</td>
</tr>
</tbody>
</table>

A value we could not measure is indicated with –
Numbers with * are sourced from datasheets
ASIC Implementation with SystemC

**C++ Design**
- C++ Test Bench
- Core Algorithm in C++
- C++ Test Memory

**SystemC Design**
- SystemC TB Wrapper
- C++ Test Bench
- SystemC Design Wrapper
- Core Algorithm in C++
- SystemC Test Memory

**Frequency**
- C++: 141MHz
- SystemC: 1GHz

Supported by DARPA award HR0011-16-C-0037 under the CRAFT program
Teaching HLS for FPGAs

- Cornell ECE 5775: High-level digital design automation
  - 30-40 graduate and senior students
  - Finished multiple HLS assignments
  - Implemented a digit recognition accelerator on FPGAs
Case Study: Digit Recognition

- Use a simple machine learning algorithm to recognize handwritten digits
  - 2000 training instances per digit
  - Each training/test instance is a 7x7 bitmap after downsampling

MNIST dataset: http://yann.lecun.com/exdb/mnist/
K-Nearest-Neighbor (KNN) Implementation

```c
bit4 digitrec( digit input )
{
    #include "training_data.h"
    // This array stores K minimum distances per training set
    bit6 knn_set[10][K_CONST];
    // Initialize the knn set
    for ( int i = 0; i < 10; ++i )
        for ( int k = 0; k < K_CONST; ++k )
            // Note that the max distance is 49
            knn_set[i][k] = 50;

    L2000: for ( int i = 0; i < TRAINING_SIZE; ++i )
    {
        L10: for ( int j = 0; j < 10; j++ )
        {
            // Read a new instance from the training set
            digit training_instance = training_data[j * TRAINING_SIZE + i];
            // Update the KNN set
            update_knn( input, training_instance, knn_set[j] );
        }
    }
}
```

Main compute loop
(10 cycles per innermost loop)

~200K cycles by default without optimizations
Assignment: 100x Speedup!

- Students are expected to insert pragmas or modify source code to achieve a 100x speedup over unoptimized baseline.
Optimization through Pragmas

- To achieve a 100x speedup over unoptimized baseline
  - Two pragmas would suffice: pipelining + array partitioning
    - Pipelining outer loop entails unrolling of the inner loop

- Only a small subset of students used the minimum set of pragmas
  - Many achieved the target speedup with pragma overuse and unnecessary code changes
  - “Wrong” combinations of pragmas led to counter-intuitive performance-area trade-offs
    - Completely partition the array hurts the performance
So Have We Solved HLS?

- Existing HLS methodologies are still not intuitive enough for non-expert users
  - Low guarantee on **out-of-the-box** quality of results (QoR)
  - **Leaky abstraction** of user directives

---

**List of tunable options of a commercial HLS tool:**

- `inline_calls` [-all] [object_id]
- `pipeline_function` [-latency {min [max]}] [-extra_timing_effort]
- `convert_to_lookup`
- `unroll_loop` [-num_pre_body integer] [-num_in_body integer] [-num_tries integer]
- `pipeline_loop` [-init_interval num_states] [-extra_timing_effort] [-min_lat_interval num_states] [-max_lat_interval num_states] [-allow_io_reordering]
- `flatten_array`
- `merge_arrays` [-data | -addr] [-min_write_width integer] [-name new_array_name]
- `split_array` -addr/data bit_index
- `restructure_array` addr_width_delta

---

**HLS Expert Users**

*Who can figure out what annotations to provide to get the desired hardware*
Our Ongoing Effort

- Co-design of programming language, compiler, and architecture
  - Domain-specific specifications
  - Cross-layer synthesis + parallel autotuning
  - Fine-grained, medium-grained, and coarse-grained co-processor architectures

// algorithm
padded(x, y, j, i) = select(x >= 0 && x < S && y >= 0 && y < S,
converted(x, y, j, i), 0);
RDom r (0, K, 0, K, 0, M);
res(x, y, j, i) += kernel(r.x, r.y, r.z, j) * padded(x+1-r.x, y+1-r.y, r.z, i);

// schedule
padded.compute_root();
res.vectorize(x).parallel()
Concluding Remarks

▸ HLS is here to stay as a key enabler for rapid hardware specialization
  – More turns per day than RTL-based flow

▸ Challenges remain to widespread use of HLS
  – Substantial room for improvement in out-of-the-box QoR and programming abstractions
  – Research opportunities abound for both architecture and DA communities!