1 Introduction

A convolutional neural network (CNN) is a machine learning algorithm that takes in an image and produces predictions on the classification of the image. A CNN consists of a series of connected layers. Each layer takes as input a set of feature maps (fmaps), performs some computation on them, and produces a new set of fmaps to be fed into the next layer. The input fmaps of the first layer come from the input images. Layers may require configuration values known as parameters, which must first be determined by training the CNN offline on pre-classified data. Once the parameters are finalized, the CNN can be deployed for inference — the classification of new data points. For most practical machine learning applications, the first-order concerns are the accuracy and execution time of online classification. Figure 1 shows a typical structure of a CNN. In this lab, we focus on the inference process. Especially, we want to perform hardware acceleration on the convolutional layers.

Networks with parameters and/or feature maps quantized to $+1/−1$ are called binarized neural networks (BNNs). Such networks have demonstrated low on-chip memory overhead, high compute throughput, and low power consumption on FPGAs while maintaining accuracy comparable to full precision networks in certain cases. Moreover, we can reduce the number of multipliers by replacing multiplications with bit operations. In this lab, we encode $+1$ as 1 in memory. In addition, $−1$ is encoded as 0. Hence, we only need one bit to store a binarized value. Figure 2 shows how we replace a multiplication with an XNOR operation after encoding. We use $\hat{x}$ to denote the encoded value of $x$.

---

1Part of this section is adapted from R. Zhao, et al. [1]
Following we show how to perform the dot product between two vectors with encoded values.

\[
\mathbf{A} \cdot \mathbf{B} = \sum_{i=0}^{L} A_i \times B_i
\]

(1.1)

\[
= 2 \sum_{i=0}^{L} (\hat{A}_i \odot \hat{B}_i) - L,
\]

(1.2)

where \( \mathbf{A} \) and \( \mathbf{B} \) are two vectors with the same length \( L \) (i.e., \( |\mathbf{A}| = |\mathbf{B}| = L \)), \( A_i \) and \( B_i \) are binarized values that are either +1 or −1, and \( \hat{A}_i \) and \( \hat{B}_i \) are encoded values for \( A_i \) and \( B_i \) according to Figure 2. The summation (\( \sum \)) in Equation (1.2) refers to integral addition.

Below we show an example.

\[
[+1, -1, -1, +1] \cdot [-1, -1, +1, +1] = 1 \times -1 + -1 \times -1 + -1 \times 1 + 1 \times 1 = 0 \quad \text{Eq. (1.1)}
\]

\[
= 2 \times (1 \odot 0 + 0 \odot 0 + 0 \odot 1 + 1 \odot 1) - 4 = 0 \quad \text{Eq. (1.2)}
\]

We can see that now we only need logic operations (i.e., XNOR) and additions to perform the dot product. The multiplication of two can be replaced with a shift operation. Below we describe two layer types which are found in most BNNs.

A **convolutional (conv)** layer takes in \( M \) input fmaps of size \( I \times I \) pixels, convolves them with filters of size \( K \times K \) pixels, and produces \( N \) output fmaps of size \( O \times O \) pixels. The convolution operation can be demonstrated using an example, which is shown in Figure 3. In this example, we have three input fmaps of size \( 5 \times 5 \) (\( i_0, i_1, \) and \( i_2 \)) and two output fmaps of size \( 3 \times 3 \) (\( o_0 \) and \( o_1 \)). To begin with, each input fmap is convolved\(^2\) with a \( 3 \times 3 \) filter, which generates a partial sum for the corresponding output pixel. In Figure 3, input fmap \( i_0 \) convolves with filter \( w_{0,0} \) and generates \( p_{0,0} \). The rest partial sums \( p_{0,1}, \ldots, p_{2,1} \) are produced in a similar manner. These partial sums are accumulated to produce the pixels of the output fmaps. In Figure 3, we sum up \( p_{0,0}, p_{1,0}, \) and \( p_{2,0} \) to produce \( o_0 \). Similarly, we can produce \( o_1 \) by accumulating \( p_{0,1}, p_{1,1}, \) and \( p_{2,1} \).

\(^2\)Please refer to Lecture 13 slides, p.36.
Figure 3: An example of convolution operation, where $M = 3$, $N = 2$, $I = 5$, $O = 3$, and $K = 3$.

From the above example, we can observe that for each output fmap, we need $M$ filters. Thus, to produce $N$ output feature maps, we need $M \times N$ filters. The above procedure can be formalized in Equation (1.3).

$$o_n(x, y) = \sum_{m=0}^{M-1} \sum_{r=0}^{K-1} \sum_{c=0}^{K-1} i_m(x+c, y+r) \times w_{m,n}(c,r),$$  \hspace{1cm} (1.3)

where $o_n(x, y)$ is the value of pixel $(x, y)$ of the $n^{th}$ output feature map, $i_m$ is the $m^{th}$ input feature map, and $w_{m,n}$ is the filter that convolves with input $i_m$ and produces a partial sum of output $o_n$. Note that we can apply Equation (1.2) to transform the multiplications into XNOR operations. For example, if we want to calculate the pixel $(2,1)$ of the first output fmap in Figure 3, we can have the following equation.
The number of multiplication-accumulation operations (MACs) needed during the above process is \( M \times N \times O \times O \times K \times K \). After we derive the output fmaps, we binarize the outputs by comparing each value with a pre-trained threshold \( t \), which is shown in Equation (1.4).

\[
\text{binarize}(x) = \begin{cases} 
+1, & x \geq t \\
-1, & x < t 
\end{cases} 
\]

(1.4)

The parameters of a conv layer are \( M \times N \times K \times K \) bits. Finally, we perform a 2D maximum pooling to halve the size of an output fmap, where we pick the maximum value in a \( 2 \times 2 \) window with a stride of two. An example is shown in Figure 4.

Figure 4: An example of performing 2D maximum pooling on a \( 8 \times 8 \) fmap, which results in a fmap of size \( 4 \times 4 \).

A dense or fully-connected (dense) layer takes in \( M \) input feature maps of size \( 1 \times 1 \) (i.e., a pixel) and produces \( N \) output feature maps of size \( 1 \times 1 \). The output feature maps are derived from the product between \( M \) input fmaps and \( M \times N \) weight matrix. Equation (1.5) shows the operation of a dense layer with \( M \) input pixels \( i_0, \ldots, i_{M-1} \), \( N \) output pixels \( o_0, \ldots, o_{N-1} \), and \( M \times N \) weights \( w_{0,0}, \ldots, w_{M-1,N-1} \).

\[
o_n = \sum_{m=0}^{M-1} i_m \times w_{m,n}
\]

(1.5)

The number of MAC operations is \( M \times N \). Similar to conv layers, we can apply Equation (1.2) to replace all multiplications. After we have the output fmaps, we quantize the output values.
according to their signs. The parameters of a dense layer are $M \times N$ bits, which are produced by the training process.

## 2 Objective

In this lab, you will be building a BNN inference accelerator on ZedBoard. Specifically, you are given a pre-trained BNN that performs digit recognition. From Table 1, we can observe from Columns 7-8 that conv layers are more compute intensive while dense layers are more memory intensive. In practice, a larger and deeper BNN has more conv layers, where the difference is more pronounced. Thus, although we offload all layers to FPGA in this lab, you should first focus on optimizing the conv layers.

![Diagram of software-hardware partition](image)

Figure 5: The software-hardware partition.

The software-hardware system is shown in Figure 5. The whole flow works as follows. First, the test images stored in CPU are sent to the hardware accelerator `bnn_xcel` one at a time. The test image will become the input fmaps of the first layer in `bnn_xcel`. The fmaps will be processed by the two conv layers inside `bnn_xcel` sequentially. The output fmaps of the last conv layer will become the input fmaps of the first dense layer in `bnn_xcel`. Finally, the prediction result is computed by the second dense layer and sent back to CPU. Your task is to optimize the performance of the accelerator with limited hardware resources.

## 3 Materials

You are given a zip file named `lab4.zip` on ecelinux under `/classes/ece5775/labs`, which contains the following directories:

- **ecelinux**: contains a C++ project for you to build the `bnn` HLS design and synthesize it to a hardware module. This code should be completed on ecelinux.
- **zedboard**: contains symbolic links to the files in the ecelinux directory required for software execution of `bnn` on CPU. This time the host program is given.

Both ecelinux and zedboard folders contain the following files:
- `bnn.cpp`: a source file which contains the program (dut) to be synthesized on FPGA.
- `bnn.h`: a header file that defines the prototype of functions in `bnn.cpp`.
- `layer.cpp`: a source file where conv layers, dense layers, and maximum pooling operations are defined.
- `layer.h`: a header file that defines the prototype of functions in `layer.cpp`. It also contains the network configuration.
- `bnn_test.cpp`: a test bench file that runs and tests the BNN model.
- `model.h`: a header file that contains the weights for all layers.
- `typedefs.h`: a header file that defines the data types used in the design.
- `data`: a folder that contains the weights and testing data.
- `run.tcl`: a Tcl script that helps you run the HLS flow.
- `Makefile`: a makefile similar to Lab 3 that helps you compile the design and generate the bitstream.

4 Design Overview

The configuration of our network is shown in Table 1.

<table>
<thead>
<tr>
<th>Layers</th>
<th>#Input fmaps</th>
<th>#Output fmaps</th>
<th>Size of Input fmaps</th>
<th>Size of Output fmaps</th>
<th>Filter Size</th>
<th>#MACs</th>
<th>#Params</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>1</td>
<td>16</td>
<td>16×16</td>
<td>8×8</td>
<td>3×3</td>
<td>36864</td>
<td>144</td>
</tr>
<tr>
<td>conv2</td>
<td>16</td>
<td>32</td>
<td>8×8</td>
<td>4×4</td>
<td>3×3</td>
<td>294912</td>
<td>4608</td>
</tr>
<tr>
<td>dense1</td>
<td>512</td>
<td>256</td>
<td>1×1</td>
<td>1×1</td>
<td>-</td>
<td>131072</td>
<td>131072</td>
</tr>
<tr>
<td>dense2</td>
<td>256</td>
<td>10</td>
<td>1×1</td>
<td>1×1</td>
<td>-</td>
<td>2560</td>
<td>2560</td>
</tr>
</tbody>
</table>

Table 1: The network configuration of our BNN.

Following we describe the details of each hardware component.

**CPU/FPGA Interface:** Similar to Lab 3, an input FIFO and an output FIFO are used to transfer data between software and hardware. Each pixel of a 16 × 16 input image is represented by a Bool variable, which stores the encoded value of the original pixel. The output is a number ranging from 0 to 9 that represents the classification result. The C/C++ code for this interface is provided in file `zedboard/host.cpp`.

**On-chip Memories:** The on-chip memories can be classified as two parts: feature maps and parameters. From Table 1, we can see that the parameters used by both conv and dense layers only require 138384 bits (17k bytes). Thus, we can store all parameters on-chip to minimize software-hardware communication. The stored weights and thresholds are in file `model.h`. All weights are already encoded for XNOR operations. Similarly, the maximum size of fmaps we need is 16 × 16 × 16 = 4096 bits. Therefore, the fmaps for conv layers are also stored on-chip. The stored fmaps are in file `bnn.cpp`. The value of each pixel in the fmaps is also encoded.

Both fmaps and parameters are stored as 1D arrays. The reason why we use 1D arrays is that we can reuse the same 1D arrays to represent matrices with different shapes. The main
issue of using 1D arrays is how we translate between the original matrix and the 1D array. Figures 6 and 7 show the mappings of 3D fmaps ($I \times I \times M$ or $O \times O \times N$) and 4D weights ($K \times K \times M \times N$) into 1D arrays, respectively.

![Figure 6: The indexing of pixels in fmaps for examples in Figure 3.](image)

The mappings are provided in file `layer.cpp` as well.

### 5 Guidelines and Hints

#### 5.1 Coding and Debugging

Your first task is to run the code on `ecelinux` by using `make` to make sure you have a functional design. The accuracy should be 0.81 (81%). In this lab, you only need to optimize the FPGA implementation on a ZedBoard. The process of generating the bitstream, logging onto a ZedBoard, and programming the ZedBoard using the bitstream is identical to Lab 3. Be aware that it may take 15-20 minutes to generate the bitstream. Since the bitstream generation may be slower with a more complex design after optimization, we strongly recommend you to use the information from the Vivado HLS synthesis report to estimate the performance of your hardware design before generating the bitstream. Similar
Figure 7: The indexing of pixels in filter weights for examples in Figure 3.

to Lab 3, use make sw to run the entire design on ARM CPU and use make fpga to run the program with the generated bitstream. Timers identical to those from Lab 3 will be used. After you complete the code, the runtimes you get from make sw and make fpga will be the CPU baseline and FPGA baseline, respectively. The FPGA baseline should be around 3500 msec for 100 test images.

5.2 Hardware Design Optimization

For hardware optimization, you are allowed to change any part of the program to reduce runtime. However, the only restriction is that the utilization ratio for each hardware resource (i.e., BRAMs, LUTs, FFs, and DSPs) should be at most 75%. You can verify this by checking your HLS synthesis report. Also, your test error should be no greater than 20%. There are a number of optimizations you can try to achieve better performance.

- Adding HLS optimization pragmas such as unroll, pipeline, and array_partition.
- Packing binarized parameters and fmaps to wide integers so that the bit operations can be performed in parallel.
- Exploring different loop orders.
- Exploiting data reuse by introducing reuse buffers.

You are also encouraged to review the deep learning tutorial\textsuperscript{3} and read the paper by Zhang,

\textsuperscript{3}Lecture 14.
et al [2]. Needless to mention, please do not change the timer.

5.3 Grading Scheme

We will do a class competition to see who could achieve the best performance (i.e., lowest execution time measured on ZedBoard), while still remaining within a fixed resource limitation. The report accounts for 3pt (out of 6) of the overall grade. Below is the grading scheme for the additional points.

- +3.0pt : Top 5 performers
- +2.5pt : Top 6-10
- +2.0pt : Top 11-15
- +1.5pt : Top 16-25
- +1.0pt : Those who speed up the design by at least 4X

5.4 Report

- Please write your report in a single-column and single-space format with a 10pt font size. Page limit is 3. Please include your names and NetIDs on the report.

- The report should start with an overview of the document. This should inform the reader what the report is about, and highlight the major results. In other words, this is similar to an abstract in a technical document. Likewise, there should be a summary, describing the results, and highlight the important points.

- There should be a section describing how you optimize the design. If you use more than one optimization methods, please compare their impacts regarding different aspect (e.g., performance, resource usage, accuracy, etc.). This section should contain a table which reports the measured runtime and resource usage under different optimization methods, including the CPU baseline and the FPGA baseline.

- All of the figures and tables should have captions. These captions should do their best to explain the figure (explain axis, units, etc.). Ideally you can understand the report just by looking at the figures and captions. But please avoid just putting some results and never saying anything about them.

- The report should only show screenshots from the tool when they demonstrate some significant idea. If you do use screenshots, make sure they are readable (e.g., not blurry). In general, you are expected to create your own figures. While more time consuming, it allows you to show the exact results, figures, and ideas you wish to present.

6 Deliverables

Please submit your lab on CMS. You are expected to submit your report and your code and scripts (and only these files, not the project files generated by the tool) in a zipped file named bnn.zip that contains the following contents:

- report.pdf: the project report in pdf format.
• The folders `ecelinux` and `zedboard`. These should contain the completed source files for the software-only and optimized FPGA implementations of the bnn design. Make sure the design can be built using the Makefile and scripts in the folders. Please run `make clean` to remove all the generated output files.

7 Acknowledgement

The baseline FPGA+Linux setup used in the lab is based on the Xillinux distribution provided by Xillybus Ltd. (http://xillybus.com/xillinux)

References


