

Lab 4: Binarized Convolutional Neural Networks

Team work: Groups of 2

Due Monday, November 04, 2024, 11:59pm

Late submission: 4% penalty per day; cannot be late by more than 6 days

1 Introduction¹

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)** (fmaps for short), 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.

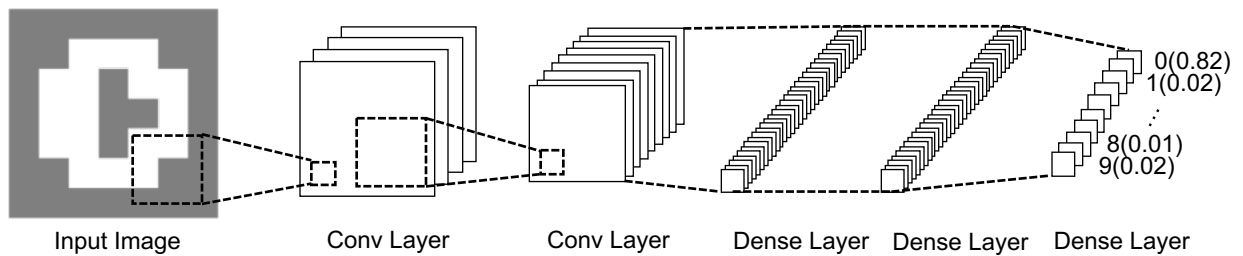


Figure 1: The typical structure of a CNN.

Networks with parameters and/or feature maps quantized to $+1/-1$ using polar encoding are called binarized neural networks (BNNs). Such networks have demonstrated high compute throughput and low on-chip memory footprint 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 represent $+1$

¹Part of this section is adapted from R. Zhao, et al. [1]

as 1 in memory. In addition, -1 is represented 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 .

| x | y | $x \times y$ |
|------|------|--------------|
| -1 | -1 | $+1$ |
| -1 | $+1$ | -1 |
| $+1$ | -1 | -1 |
| $+1$ | $+1$ | $+1$ |

(a) Normal multiplication between binarized variables x and y

| \hat{x} | \hat{y} | $\hat{x} \odot \hat{y}$ |
|-----------|-----------|-------------------------|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(b) Multiplication using XNOR with encoded variables \hat{x} and \hat{y}

Figure 2: The encoding and multiplication for binarized variables.

In the following, we show how to perform a dot product between two binarized vectors.

$$\mathbf{A} \cdot \mathbf{B} = \sum_{i=0}^L A_i \times B_i \quad (1.1)$$

$$= 2 \times \sum_{i=0}^L (\hat{A}_i \odot \hat{B}_i) - L \quad (1.2)$$

Here \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 the binarized elements that are either $+1$ or -1 in polar encoding; \hat{A}_i and \hat{B}_i are the actual binary values stored as 0/1 for A_i and B_i according to Figure 2. Below is a concrete 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 simple shift operation.

There are two common layer types in most BNNs — convolutional and fully connected layers. 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 5×5 (i_0 , i_1 , and i_2) and two output fmaps of size 3×3 (o_0 and o_1). To begin with, each input fmap is convolved with a 3×3 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 $p_{0,1}, \dots, p_{2,1}$ 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 $p_{2,1}$.

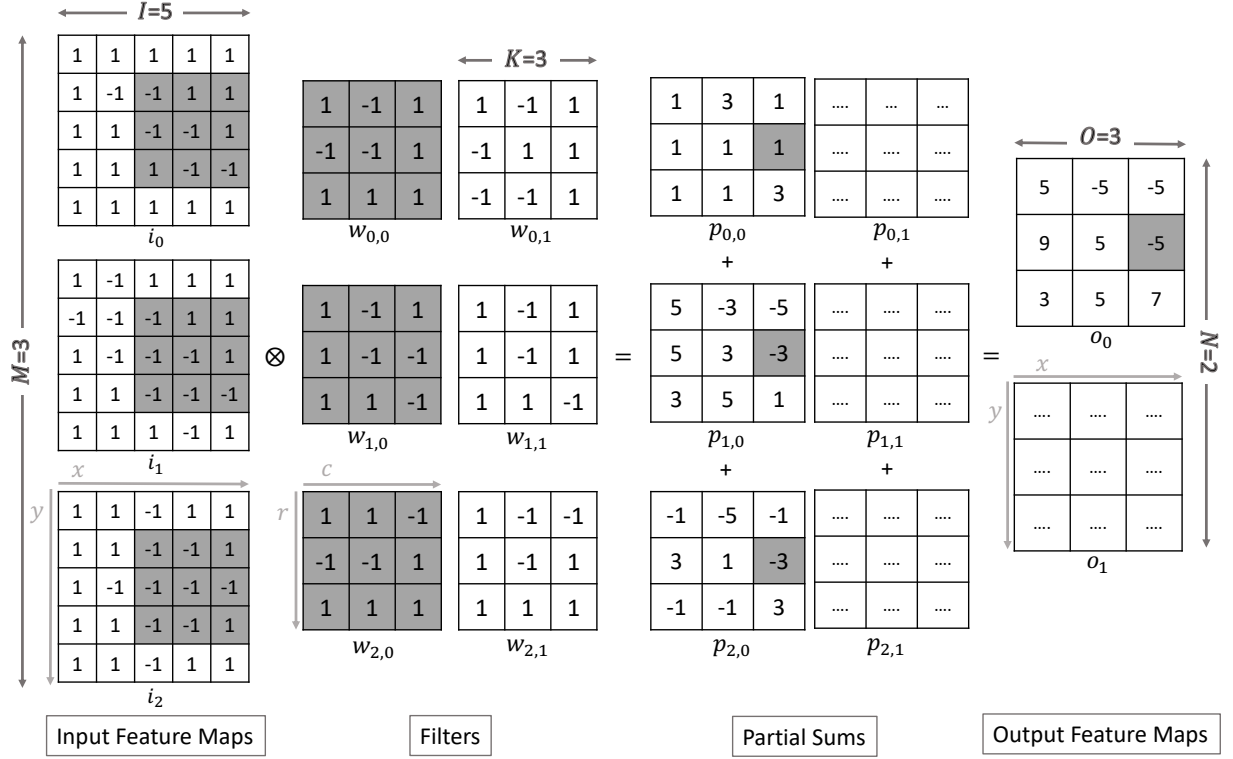


Figure 3: An example of convolution operation, where $M = 3$, $N = 2$, $I = 5$, $O = 3$, and $K = 3$.

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).

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

Here $o_n(x, y)$ 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.

$$\begin{aligned}
o_0(2, 1) &= \sum_{m=0}^2 \sum_{r=0}^2 \sum_{c=0}^2 i_m(2+c, 1+r) \times w_{m,0}(c, r) \\
&= 2 \times \sum_{m=0}^2 \sum_{r=0}^2 \sum_{c=0}^2 (\hat{i}_m(2+c, 1+r) \odot \hat{w}_{m,0}(c, r)) - 3 \times 3 \times 3 \\
&= 2 \times \{(0 \odot 1 + 1 \odot 0 + 1 \odot 1 + \dots + 0 \odot 1) \\
&\quad + (0 \odot 1 + 1 \odot 0 + 1 \odot 1 + \dots + 0 \odot 0) \\
&\quad + (0 \odot 1 + 0 \odot 1 + 1 \odot 0 + \dots + 1 \odot 1)\} - 27 = -5
\end{aligned}$$

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).

$$\text{binarize}(x) = \begin{cases} +1, & x \geq t \\ -1, & x < t \end{cases} \quad (1.4)$$

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 2×2 window with a stride of two. An example is shown in Figure 4.

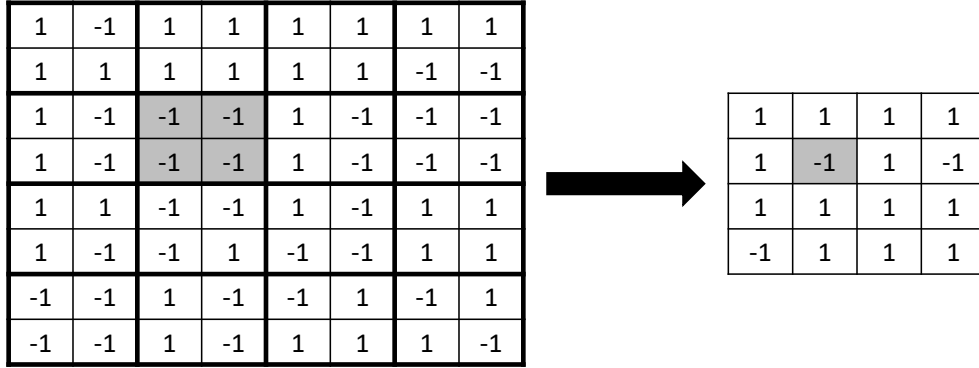


Figure 4: An example of performing 2D maximum pooling on a 8×8 fmap, which results in a fmap of size 4×4 .

A **fully-connected (or dense)** layer takes in M input feature maps of size 1×1 (i.e., a pixel) and produces N output feature maps of size 1×1 . 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, \dots, i_{M-1} , N output pixels o_0, \dots, o_{N-1} , and $M \times N$ weights $w_{0,0}, \dots, w_{M-1,N-1}$.

$$o_n = \sum_{m=0}^{M-1} i_m \times w_{m,n} \quad (1.5)$$

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 the last two columns 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.

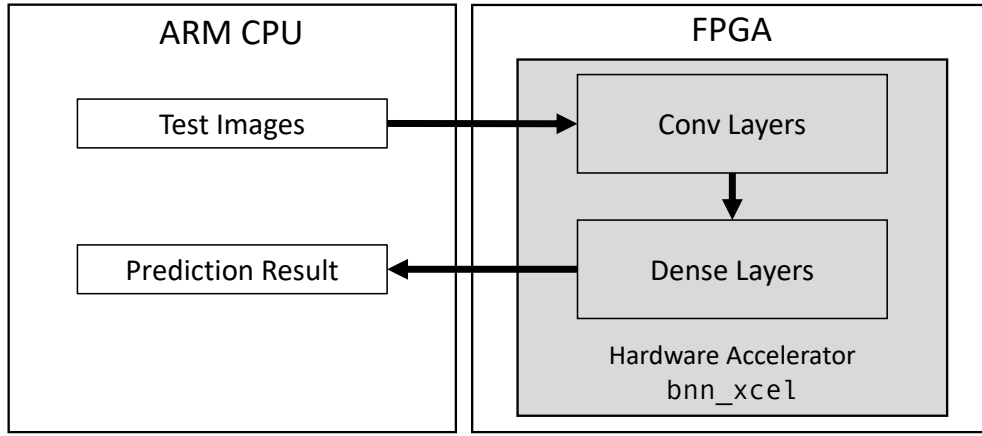


Figure 5: The software-hardware partition.

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/ece6775/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.

You may find the following files in *ecelinux* and *zedboard* folders:

- *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.h*: a header file that defines the templates for the layers in the BNN
- *bnn.test.cpp*: a test bench file that runs and tests the BNN model.
- *host.cpp*: a program that runs on the ARM core and invokes the BNN accelerator.
- *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 (csim and csynth).
- *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.

| Layers ² | #Input fmaps | #Output fmaps | Size of Input fmaps | Size of Output fmaps | Filter Size | #MACs | #Params |
|---------------------|--------------|---------------|---------------------|----------------------|-------------|--------|---------|
| conv1 | 1 | 16 | 16×16 | 8×8 | 3×3 | 36864 | 144 |
| conv2 | 16 | 32 | 8×8 | 4×4 | 3×3 | 294912 | 4608 |
| dense1 | 512 | 256 | 1×1 | 1×1 | - | 131072 | 131072 |
| dense2 | 256 | 10 | 1×1 | 1×1 | - | 2560 | 2560 |

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 $16 \times 16 \times 16 = 4096$ 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.

²Convolutional layers are followed by maxpooling layers, which downsample feature map sizes by 2.

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 around 0.90 (90%). 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 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 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. The host program will first test the accuracy over 100 images; this part is not timed. Then, it starts the timer and feeds the 100-image dataset to the accelerator for 20 repetitions (i.e., 2000 images in total) to amortize the CPU-FPGA communication latency. After running the code, the execution times you get from **make sw** and **make fpga** will be the CPU baseline and FPGA baseline, respectively. The FPGA baseline should take around **27600 ms** to process **2000** images.

5.2 Design Optimizations and Constraints

In the first three labs, you have explored various design optimization techniques related to customized data types (e.g. fixed point) and customized compute engines that exploit parallel processing and pipelining. In this lab, you will further learn to build customized on-chip memory architectures to maximize the efficiency of your accelerator. Besides using HLS pragmas/directives, such as **unroll**, **pipeline**, and **array_partition**, you will also need to modify the given source code to reorganize the on-chip data storage. Specifically, there are several optimizations you can try to implement.³

- Reorganize the arrays that store the binary parameters and fmaps by packing multiple 1-bit values into integers with a higher bitwidth (say 32 or 64 bits). This would allow each access to the array to read or write many bits in one slot and naturally enable many (bitwise) XNOR operations to be performed in parallel.
 - This *can* be achieved by repacking the weights into wider words ahead of time and rewriting the code. However, the **array_reshape** directive exists specifically to perform such an optimization without major code rewrites. You can find the full documentation on **array_reshape** in UG902 [3]. In a nutshell, the pragma uses similar syntax to **array_partition**, but instead the directive reorganizes the data layout such that the number of required memory operations is reduced.
- Exploit data reuse in 2D convolution by introducing reuse buffers, which is discussed in Lecture 2. To understand the key concepts, you can refer to Chapter 9 “Video Systems” of this book [4]. Then, you should be able to restructure your code with **unroll** and **array_partition** pragmas to implement such a reuse scheme yourself. Alternatively, there are also helpful tutorials on YouTube [5] that give detailed explanation

³You don’t have to implement all of them to achieve a good performance.

and guidelines of using the `LineBuffer` and `Window` provided by Vivado HLS.

- If you choose to use the HLS video library, make sure to only include necessary headers to avoid compilation issues on the Zedboard. For example, if you want to use `hls::LineBuffer`:

```
// BAD! Don't do this.
#include "hls_video.h"
// Do this.
typedef ap_uint<32> HLS_SIZE_T;
#include "hls/hls_video_mem.h"
```

You are allowed and encouraged to change any parts of the program to improve performance; the only two restrictions are:

- The utilization ratio for any type of resources (i.e., BRAMs, LUTs, FFs, and DSPs) **should not exceed 85%**.⁴ You can verify this by checking the HLS synthesis report.
- The test error should be **no greater than 10%**.

If HLS takes a long time to synthesize your design (it usually finishes in several minutes), or Vivado takes hours to generate the bitstream, please check related warnings in the HLS log and adjust your optimizations accordingly.

5.3 Grading Scheme

This lab has 10 points and a 0.5pt bonus. Up to 6.5 points are given according to the speedup achieved by the optimized design. Below is the grading scheme:

- **+2.5pt**: Estimated worst-case latency is at least $10\times$ lower than the estimated worst-case latency of the baseline from the HLS report.
- **+1.5pt**: Measured execution time on Zedboard is at least $30\times$ lower than the FPGA baseline.
- **+1pt**: Measured execution time on Zedboard is at least $60\times$ lower than the FPGA baseline.
- **+1pt**: Measured execution time on Zedboard is at least $120\times$ lower than the FPGA baseline.
- **+0.5pt (bonus)**: Measured execution time on Zedboard is at least $200\times$ lower than the FPGA baseline.

Points are accumulative; for example, if a design achieves $100\times$ speed up from the HLS report and $65\times$ speedup running on the Zedboard, it earns $2.5 + 1.5 + 1 = 5$ points.

The report is worth 4 points.

5.4 Report

- Please write your report in a **single-column and single-space format with a 10pt font size. Page limit is 2**. Please include the names and NetIDs of your team members on the report.

⁴Using more resources may cause routing congestion that increases the compile time to generate bitstream.

- 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.
- 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 aspects (e.g., performance, resource utilization, accuracy). This section should contain a table which reports the measured execution time, LUT, FF, DSP, and BRAM resource utilizations under different optimization methods, including the CPU baseline and the FPGA baseline.
- There should be one paragraph at the end that concisely describe the division of work between the team members.
- 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.

It is preferred that you zip your solution with `make bnn.zip` in the lab root directory in order to prevent any unneeded files or directories from being included.

7 Acknowledgement

The baseline FPGA+Linux setup used in the lab is based on the Xilinx distribution provided by Xillybus (<https://xillybus.com/xilinx>).

References

- [1] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, *Accelerating Binarized Convolutional Neural Networks with Software-Programmable FP-*

- GAs*, International Symposium on Field-Programmable Gate Arrays (FPGA), Feb. 2017.
- [2] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, *Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks*, International Symposium on Field-Programmable Gate Arrays (FPGA), Feb. 2015.
- [3] Xilinx Inc., *Vivado Design Suite User Guide: High-Level Synthesis UG902 (v2019.2)*, Available at <https://docs.amd.com/v/u/2019.2-English/ug902-vivado-high-level-synthesis>
- [4] R. Kastner, J. Matai, and S. Neuendorffer, *Parallel Programming for FPGAs*, 2018. Available at <https://arxiv.org/pdf/1805.03648.pdf>
- [5] YouTube Tutorials, *Vivado HLS 2D Convolution on Hardware*. [Part 1], [Part 2], [Part 3]