情報工学コース卒業研究報告

Efficient Hardware Design of Quantized DNN Inference Using HLS and DNN Framework

2021年 2月

KIM Hyunjae

Abstract

DNN (Deep Neural Network) implementation on an embedded system with FPGAs has been actively conducted. Although several deployment tools for FPGA have been developed, such as PYNQ by Xilinx, DNN frameworks that generate trained networks are rare that can accelerate on FPGA. Therefore, this paper aims at the FPGA implementation of networks generated in a DNN framework, which is not targeting FPGA.

In order to implement DNN networks on FPGA, the use of floating-point numbers should be avoided due to the increase of the computation cost and circuit size. Consequently, we can think of quantizing the data to an integer. However, in order for the inference of the quantized implemented network to be equivalent to the inference in the original framework, it is necessary to run quantization aware training at the framework. Therefore we choose Neural Network Console (NNC) as a target of implementation. NNC can quantize parameters and activations from the stage of training. It is expected that the inference result of NNC and the inference result of the network implemented in the FPGA can match.

In this paper, we implement networks generated by the NNC on FPGAs. Moreover, we also explore high-efficiency hardware design using High-Level Synthesis. In a case study, we implement three networks where all parameters and activations are quantized to 8-bit integers.

We compare two hardware designs to explore high-efficiency hardware design in the case study. The first design uses a pipeline execution of a loop statement operating matrix multiplication, while the second one utilizes pipelined layers of DNN. In DNN, the main consumption of inference time occurs in the multiplication and addition of matrices. As a result of the performance gains obtained in pipelined matrix multiplications, it is observed that there is a 35 percent of performance improvement by pipelining one matrix multiplication. At the same time, there is no significant difference in circuit size. The LUT usage rate increased by only 0.1 percent. As for the pipelined DNN layers, the result is a 3.5 percent of performance improvement and a 0.45 percent increase in LUT usage. Thus, the efficiency is not as great as the pipelined multiplication.

Implementing without using any floating-point arithmetic, we confirm that inferences' accuracy is comparable to the original network. Of the three networks implemented, two confirm that the inference results are equivalent to the original, and one matches 99.9 percent. In the model that does not match completely, six inferences out of 10,000 are different from the original. Finding out the cause of this error should be done in the future.

We expect this research to be a first step that makes NNC able to support more hardware platforms. This implies we will be able to run a DNN model on various hardware.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Recently, there is a demand for Deep Neural Network (DNN) applications in embedded systems. For DNN development's convenience, several DNN frameworks such as Tensorflow [1] and PyTorch [2] have been developed. Some frameworks can accelerate computation by utilizing an external computing device rather than a CPU. A CPU has a disadvantage when using it for DNN calculation. A CPU is designed for generic serial operations, which is not suitable for matrix calculation, often required by DNN. Furthermore, a CPU's memory bandwidth is relatively slower than others, leading to a bottleneck issue. For example, a modern x86 CPU bandwidth using DDR4 SDRAM can have a maximum bandwidth of 35.2GB/s. However, the bandwidth of modern GPU using GDDR6X SDRAM can have a maximum bandwidth of 1TB/s (Reference [3,4]). For these reasons, a popular framework such as Tensorflow accelerates a calculation by utilizing GPU's high bandwidth memory and parallel computing power. However, frameworks that utilize non-GPU hardware such as FPGA are relatively rare. There are frameworks developed for implementing a DNN inference application to embedded systems. For example, TensorFlow Lite for Microcontrollers [5] allows running machine learning models with quantized parameters on devices with a few kilobytes of memory. Nevertheless, it does not support FPGA.

1.2 Purpose and contribution of the paper

Since many frameworks do not support implementing a network to embedded systems with FPGA, we review implementing networks created and trained by existing DNN frameworks in this paper. Furthermore, we explore efficient hardware design of implementation. We choose Neural Network Console (NNC) as its implementation target. The main reason for choosing NNC is that NNC provides to quantize its parameters and data path between layers. NNC supports quantization at the stage of training. This feature is essential because calculating with floating-point arithmetic requires more memory and computation cost than an integer calculation. Furthermore, we can make a FPGA circuit size relatively small by utilizing lower bit quantization.

Moreover, NNC provides a user-friendly experience by utilizing its GUI. If we can automate a implementation of NNC's network, we expect NNC becomes a more productive tool. Therefore, by reviewing implementing networks generated by NNC, we believe this could be the first step for creating an automated tool.

Implementation of FPGA using register-transfer level (RTL) language can make further optimization difficult and might be time-consuming. Therefore, we use a system-level design toolkit named SystemBuilder with high-level synthesis (HLS) for development [6,7]. We will present more detail about this process in section 3.1.

The contribution of this paper are as follows:

- Review of the FPGA implementation of a network generated by Neural Network Console.
- A case study shows that inferences from implementation were comparable as the original network from the framework.
- Exploring an efficient hardware design using HLS.

We believe that contribution of this paper will help in the further development of a tool that automatically implements the network from NNC. We present this tool in **Figure 6.1**.

Chapter 2

Related Work

2.1 DNNExplorer

There are various studies on customized DNN accelerators using FPGAs. Zhang and Ye study methods to explorer optimal FPGA design in [8]. They develop a framework named DNNExplorer for modeling and exploring a design. They claim that existing FPGA-based DNN accelerators typically fall into two design paradigms. The one is a paradigm that adopts a generic reusable architecture to support various DNN networks. A generic reusable architecture means this design sacrifices further performance and efficiency for generic reuse. The other design paradigm is a paradigm that adopts a layer-wise tailor-made architecture to optimize a layer-specific demand. Each layer in this design is combined into pipeline implementation. This design loses scalability as it has a limited number of DNN layers that can be supported. Moreover, deeper DNN means fewer resources for each layer, which might lead to performance degradation.

The key idea of DNNExplorer is to use both paradigms simultaneously. DNNExplorer utilizes the second's pipeline design for the first up to the split-point (SP) layer and uses the first's generic accelerator for the rest. DNNExplorer explores design architecture to find out optimal split-point, and it maps resources for each design architecture.

We believe this approach is an excellent reference for further design exploration in our model.

2.2 Other implementations of a DNN Framework

FINN is one of the DNN framework for an FPGA by Xilinx, Inc [9,10]. This framework targets quantized neural network, and the framework provides a training environment to set bit width. Moreover, PYNQ (Python productivity for ZYNQ) [11] is a hardware platform to develop DNN inference applications in Python. Developers can develop DNN inference more quickly because they can develop it on a browser with a Jupyter notebook. However, our research differs in that we aim to implement a machine learning model generated in a DNN framework that does not target FPGAs.

Nakahara et al. developed GUINNESS (GUI-based Neural Network Environment Synthe-Sizer) for DNN inference on an FPGA [12,13]. GUINNESS also provides train front-end with Chainer and supports from training to C++ code for HLS. The C++ code is inputted to SD-SoC [14], then HDL codes are generated for target FPGA. The hardware receives an activation from Linux on CPU; however, it co-work with small-scale and low overhead RTOS. Therefore, our framework has advantages for real-time systems and automotive systems.

Chapter 3

Preparation

This chapter discusses two tools we used in this paper, SystemBuilder and Neural Network Console.

3.1 SystemBuilder

For the convenience of development for the FPGA, we used a system-level design toolkit named SystemBuilder. SystemBuilder is a toolkit that abstracts the interface between hardware and software, such as a device interface and bus interface. Due to such abstraction, a designer can have better productivity by eliminating the need to do such low-level designs themselves.

In SystemBuilder, we name processes that run either hardware or software as functional units. As a result of the interface's abstraction, SystemBuilder provides four communication channels between functional units. We name these channels as a communication primitive. Since a functional unit could be either hardware or software, these primitive can also be used between hardware and software or between hardware and hardware, or between software and software.

A Designer should develop functional units and define a connection between units using the provided communication primitive. A unit will be implemented either a hardware or software process based on a system specification that a designer defined. We will discuss more details of a design description in section 3.1.1.

SystemBuilder then generates RTL hardware, RTOS-specific software, and the interface of these. SystemBuilder executes an external tool for synthesizing. Since SystemBuilder does partitioning of hardware and software, a designer can now efficiently explore hardware/software partitioning.

We will discuss the detail of the synthesis used in SystemBuilder in section 3.1.5. In section 3.1.10, we will explain a co-simulation feature that SystemBuilder provides. To summarize, we will discuss the entire workflow in section 3.1.12.

3.1.1 Design description

A designer should create the following files to use SystemBuilder.

- Functional units written in C language.
- System DeFinition (SDF) file. This is a description of a design target. **Figure 3.1** is an example of the SDF file we designed in the case study. Due to space limitations, this figure contains only part of the file.

  An SDF file includes the following information:

  – A partitioning scheme of functional units.

    A partitioning scheme corresponds to line 3–7 in **Figure 3.1**. The figure declares ‘topmod’ as a software process and the rest of the units as hardware processes.

  – A number and names of communication primitives.

    It corresponds to line 11–21 in **Figure 3.1**. It declares seven ‘BlockingChannel’ and one ‘MemoryChannel’.

  – Connection relations between functional units and communication primitives.

    It corresponds to line 23–39 in **Figure 3.1**. Under each functional unit (process), it declares the relations with communication primitive. For example, in line 34–37, primitive ‘conv1_mpool1’ is declared as an input, and primitive ‘mpool1_conv2’ is declared as an output of process ‘mpool1’.

SystemBuilder reads these files and follows the steps presented in section 3.1.12.

3.1.2 Partitioning

A designer using SystemBuilder should define a partition scheme in SDF File. This means a designer can decide whether to implement functional units in hardware or software. For example, in **Figure 3.2**, unit 1 is partitioned as a software process, units 2 and 3 as hardware processes. To test another partitioning scheme, a designer has only to edit an SDF file and rerun SystemBuilder. In this way, a designer can efficiently explore a partition scheme of hardware/software. To test an implementation works as expected, a designer can run a co-simulation. A Designer can test an implementation even faster by building all functional units as software. Since emulating software is much faster than simulating hardware, it is expected to be more efficient than running co-simulation. Co-simulation/simulation is discussed in section 3.1.10.

```
SysName: sp_hw

SW:
 - {core: Nios, id: 1, process: [topmod]}

HW:
 - {core: FPGA, process: [conv1, conv2, mpool1, mpool2, aff1, aff2]}

#HW_FREQ_MHZ = 50

BlockingChannel:
 - {name: start,  size: 8}
 - {name: result,  size: 8}
 - {name: conv1_mpool1,   size: 8, depth:  64}
 - {name: mpool1_conv2,   size: 8, depth:  64}
 - {name: conv2_mpool2,   size: 8, depth:  64}
 - {name: mpool2_aff1,    size: 8, depth:  64}
 - {name: aff1_aff2,      size: 8, depth:  64}

MemoryChannel:
 - {name: in_img_mem, size: 8, depth: 784}

StandardProcess:
 - name:    topmod
   file:    [topmod.c]
   ch(in):  [result]
   ch(out): [start, in_img_mem]

 - name:    conv1
   file:    [conv1.c]
   ch(in):  [start, in_img_mem]
   ch(out): [conv1_mpool1]

 - name:    mpool1
   file:    [mpool1.c]
   ch(in):  [conv1_mpool1]
   ch(out): [mpool1_conv2]

 - name:    conv2
```

Figure 3.1: Example of System DeFinition file

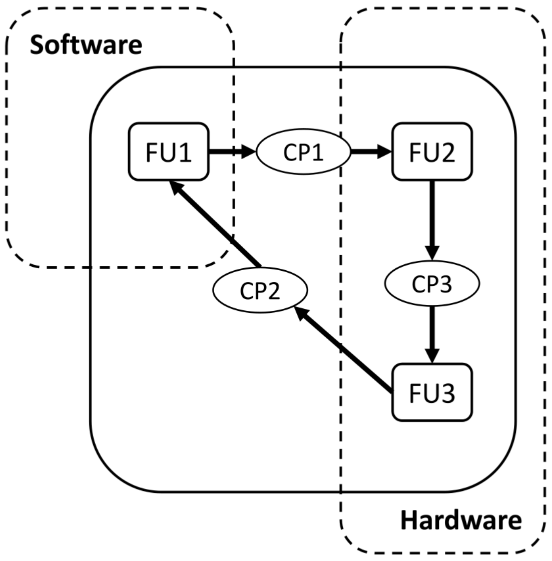


Figure 3.2: Example of partitioning Functional Units

3.1.3 Functional Units (FU)

As discussed in section 3.1.2, a designer can decide whether to implement a functional unit in hardware or software. However, there is a limitation. SystemBuilder uses an external high-level synthesis tool. It means a functional unit to be implemented as a hardware process must be subjected to the external HLS tool's limitation. On the contrary, a unit to be implemented as a software process does not have those limitations. It means that all units can be built as a software process, and it allows to test a design at a functional level.

3.1.4 Communication Primitives

SystemBuilder supports four communication primitive as follows.

Non-Blocking Communication Primitives (NBC)
NBC corresponds to a shared variable in software and a register in hardware.

Blocking Communication Primitives (BC)
BC corresponds to OS's communication feature in software and a FIFO in hardware. Access is executed by blocking in one direction.

Memory Primitives (MEM)
MEM corresponds to a global array in software and a dual-port memory in hardware. Access is executed by non-blocking. It is possible to implement a memory primitive as

shared memory. In order to implement as shared memory, shared memory must be declared in an SDF file.

PreFetch BC (PFBC)

PFBC is a memory designed between external memory and hardware process. The purpose of PFBC is to hide latency while reading external memory. PFBC reads the bulk of data from external memory and provides fast internal memory to hardware processes connected. PFBC uses shared memory only available for the software process side; therefore, shared variables must also be declared separately.

These primitives can be accessed as a C language function when designing a unit.

3.1.5 Synthesis

SystemBuilder takes SDF and C programs as an input and generates files for software, hardware, and interface.

3.1.6 Software Synthesis

Software synthesis takes place by a compiler. It will be compiled as software running on either ITRON [15] or AUTOSAR-CP [16]. Communication primitives between a software process are also implemented as software.

3.1.7 Hardware Synthesis

A hardware process and communication primitives between them will be synthesis as hardware. SystemBuilder uses a commercial tool to synthesis hardware processes. SystemBuilder supports the following HLS tool.

- eXCite
- CyberWorkBech [17]
- Vivado HLS [18]

These HLS tools support various optimization techniques such as:

- Pipelining the loop to implement the operations in a loop in a concurrent manner.
- Unrolling the loop to exploit parallelism between loop iterations.
- Expanding the array signal into the individual variable.
- Unrolling multi-dimensional array by a dimension designer configured.

- Inlining the function.

In order to take advantage of these features, a designer is required to develop processes with this in mind. We had a brief compare test of pipelining the network's loop statement represented in **Table 5.3** and **Table 5.4**.

3.1.8 Interface Synthesis

SystemBuilder automatically generates a hardware/software interface based on the SDF file. For the software, the device driver is generated. The device driver includes access functions, interrupt handlers, and semaphores for synchronization between hardware and software. For the hardware, the interface circuit described in HDL is generated. This interface includes glue logic that allows accessing the bus and the circuit that corresponds to communication primitives such as buffer or memory. The glue logic has an interface to a generic bus named VBUS. Therefore, SystemBuilder generates a circuit that converts protocol to connect VBUS.

3.1.9 Command

This subsection discusses the actual commands for synthesizing.

SystemBuilder provides '`SysGen`' command to synthesis the system. '`SysGen`' has the following options [19].

- Specify the SDF file.

  `-sdfy <SDF file>` Specify the YAML format of the SDF file.

  `-sdf  <SDF file>` Specify the sdf format of the SDF file.

- Specify the synthesis method.

  `-nobs` Specify to not to perform high-level synthesis.

  `-dobs` Specify to perform high-level synthesis.

  `-csim` Specify to print C language level simulation.

- Specify high-level synthesis tool.

  `-bst <HLS tool name>`

- Specify the target board.

  `-s <board name>`

- Specify the RTOS for running software processes.

```
    -os <itron or atk>
```

'SysGen' commands synthesis the interface and hardware processes. It also generates 'Makefile' for software compiles.

3.1.10 Co-simulation

To test a design, SystemBuilder supports hardware and software co-simulation. It runs RTOS emulator and RTL simulators and then coordinating them to perform co-simulation. It connects the RTOS emulator and RTL simulator by a program named device manager. The device manager runs on the host computer then connects to the RTOS emulator using an inter-process communication function from the host pc. While the RTOS emulator tries to communicate with the RTL simulator, the device manager sends a signal to the RTL simulator. The device manager uses Foreign Language Interface (FLI) to connect VBUS to the RTL simulator.

SystemBuilder supports running software processes either on QEMU or directly from the host computer. Functional Units that run on RTOS are compiled and linked with the RTOS model. Next, SystemBuilder generates an object code directly executable on the host computer [20]. Due to native support, co-simulation speed is much faster than other simulators that simulate at the instruction-set level. In this paper, we used QEMU for the RTOS emulator and Questa Sim for the RTL simulator.

The benefit of running co-simulation is that designer can debug hardware processes. For example, a designer can observe a waveform of signals. **Figure 3.3** is an example of a waveform. In this figure, we can observe signals and variables.

3.1.11 Report from a HLS tool

As a result of a high-level synthesis of hardware processes, several HDL files are created. To implement this result to the FPGA, logic synthesizing is required. At this time, the SystemBuilder requests the HLS tool to print several design reports. Design reports contain various information such as BRAM usage and LUT usage. **Table 3.1** is an example of a memory usage report from Vivado. This table shows the BRAM usage of the design out of the RAM on the board. **Table 3.2** is an example of LUT usage report. We can know the logic circuit size by LUT as logic usage. LUT usage can be an indicator of design efficiency.

3.1.12 Workflow

In summary, we represent the workflow inside SystemBuilder in this section. **Figure 3.4** and the following steps are an overview of workflow inside SystemBuilder.

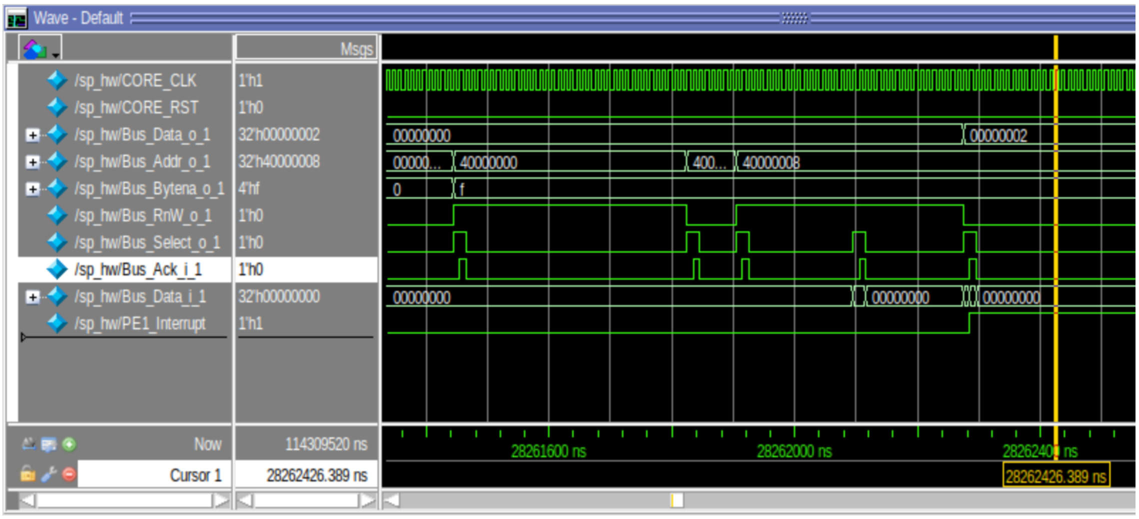


Figure 3.3: Waveform observed in Questa Sim

Table 3.1: Example of memory usage report

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	44.5	0	140	31.79
RAMB36/FIFO*	41	0	140	29.29
RAMB36E1 only	41			
RAMB18	7	0	280	2.50
RAMB18E1 only	7			

Table 3.2: Example of LUT usage report

Site Type	Used	Fixed	Available	Util%
Slice	3858	0	13300	29.01
SLICEL	2587	0		
SLICEM	1271	0		
LUT as Logic	9675	0	53200	18.19
using O5 output only	3			
using O6 output only	7520			
using O5 and O6	2152			

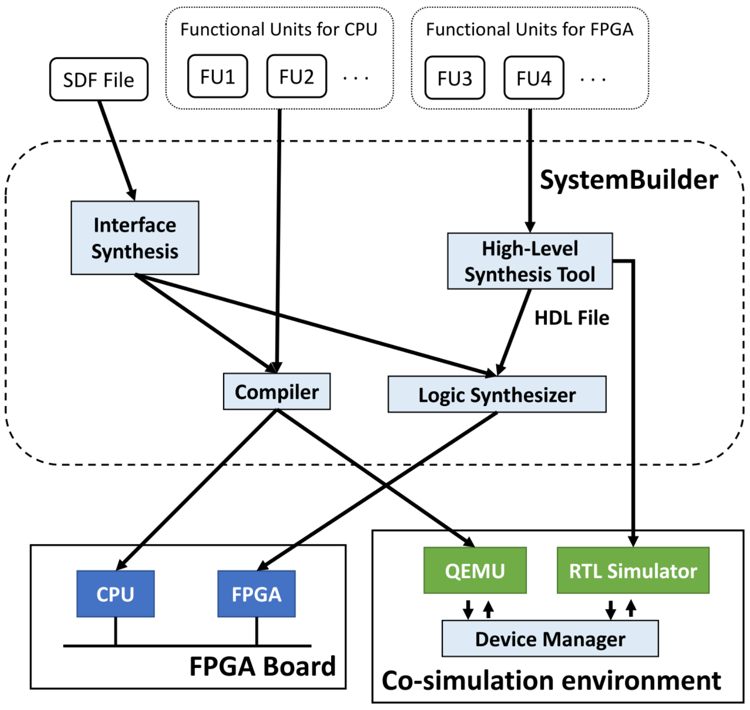


Figure 3.4: Workflow inside SystemBuilder.

1. SystemBuilder read SDF Files and determines a partition scheme of functional units.
2. SystemBuilder synthesis the interface between hardware and software.
3. For software, a compiler compiles functional units with the device driver generated in step 2.
4. For hardware, HLS Tool synthesis the code to HDL files.
5. Users now can choose whether to run co-simulation or implement the results to FPGA.

   Co-simulation This process is described in section 3.1.10.

   FPGA Logic synthesizer synthesis the HDL file created and device interface generated in step 2. This process generates design reports discussed in section 3.1.11.

3.2 Neural Network Console

A Designer can develop a network via GUI in NNC. **Figure 3.5** is an example of a network designed using NNC. As shown in this figure, NNC visualizes which layers are used in what order, from input to output.

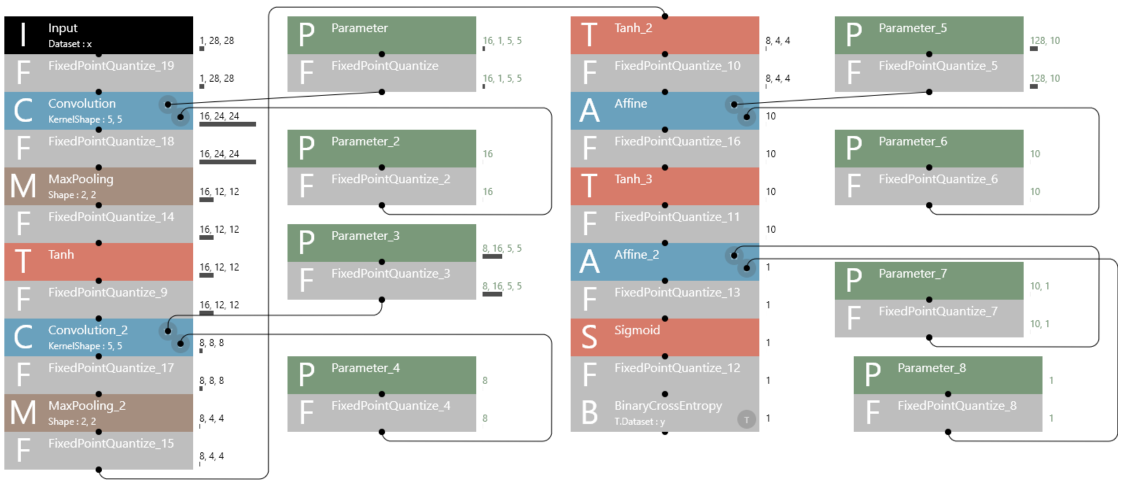


Figure 3.5: Example of designing a neural network in NNC

This section will cover the behavior of FixedPointQuantize, convolution, affine layers of NNC. We will also discuss the behavior at the source code level to understand layers' behavior better. The source code referenced in this section is Neural Network Libraries' code, which NNC is built on.

3.2.1 FixedPointQuantize

FixedPointQuantize layer performs linear quantization. This layer requires the following properties [21].

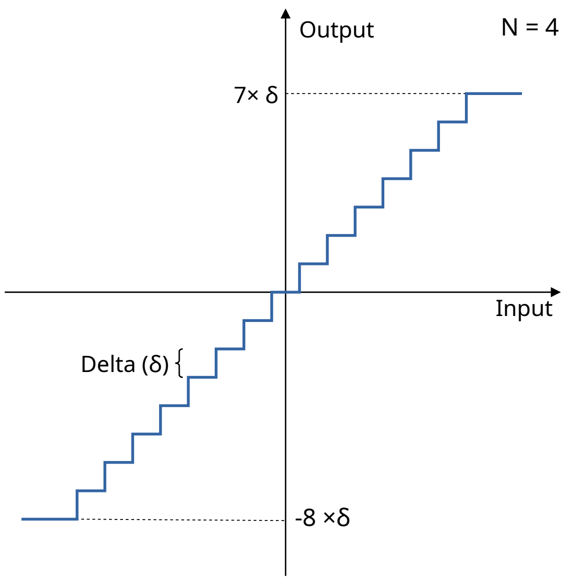
Sign Specify whether to include signs.

N Specify the number of quantization bits.

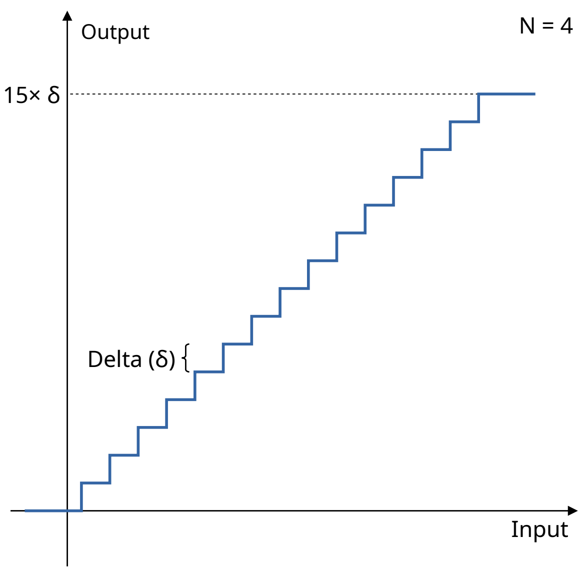
Delta (δ) Specify the quantization step size.

The FixedPointQuantize layer quantizes input and prints out data using the above properties. Printed data of the layer is not an **N**-bits integer but a float. **Figure 3.6** is a graph that expresses the FixedPointQuantize layer's behavior. The layer's output will be the value passed through the step function. **N** is specified as four in this figure. Therefore, the number of printable data is limited to $2^{\mathbf{4}}$. As a result, if **Sign** is true, the maximum and minimum output will be 7δ and -8δ. Furthermore, if **Sign** is false, the maximum and minimum output will be 15δ and 0.

In order to explain the behavior more specifically, the corresponding source code is described in **Figure 3.7**. This figure is directly imported from a file. Line 54 to 57 are codes that check overflow and underflow. Line 59 to 62 are codes that check the sign of the number (`sign_x`) and modify the value using the quantization step (`delta_`).



(a) Sign = True



(b) Sign = False

Figure 3.6: Graph of FixedPointQuantize layer

```
T y_tmp;
for (int s = 0; s < inputs[0]->size(); s++) {
  if (x[s] > max_) {
    y_tmp = max_;
  } else if (x[s] < min_) {
    y_tmp = min_;
  } else {
    bool sign_x = (x[s] < 0.0);
    T abs_x = std::fabs(x[s]);
    y_tmp = T(int((abs_x / delta_) + 0.5)) * delta_;
    y_tmp = sign_x ? -y_tmp : y_tmp;
  }
  y[s] = y_tmp;
}
```

Figure 3.7: `src/nbla/function/generic/fixed_point_quantize.cpp`

```
178   for (int n = 0; n < outer_size_; ++n) {
179     // Im2col
180     unfold_to_patches<T>(x + n * inner_size_i_, col, channels_i_,
181                          spatial_shape_i_, kernel_, pad_, stride_, dilation_);
182     // Convolution by matrix multiplication
183     T *y_n = y + n * inner_size_o_;
184     for (int g = 0; g < group_; ++g) {
185       MatrixMap<T> mcol(col + g * row_col_ * col_col_, row_col_, col_col_);
186       ConstMatrixMap<T> mk(w + g * row_w_ * col_w_, row_w_, col_w_);
187       MatrixMap<T> my(y_n + g * row_y_ * col_y_, row_y_, col_y_);
188       my = mk * mcol;
189     }
190     // Adding bias
191     if (inputs.size() == 3) {
192       MatrixMap<T> my(y_n, channels_o_, col_y_);
193       my.colwise() += ConstColVectorMap<T>(b, channels_o_);
194     }
195   }
```

Figure 3.8: `src/nbla/function/generic/convolution.cpp`

By this method, it is possible to keep the FixedPointQuantize layer's output as a float. As a result, the other layers do not need to consider whether the input is quantized or not.

3.2.2 Convolution

The convolution layer convolves the input as the following equation.

$$\mathbf{O}_{x,y,m} = \sum_{i,j,n} \mathbf{W}_{i,j,n,m}\, \mathbf{I}_{x+i,y+j,n} + \mathbf{b}_m \tag{3.1}$$

Where **O** is the output, **W** is the kernel weight, **I** is the input, **b** is the bias, i, j is the kernel size, x, y, n is the input index, m is the output map. As the equation, it includes multiplications and additions of matrices.

Figure 3.8 is the part of the convolution layer's source code. The for loop at line 184 is where $\sum$ in the equation 3.1 occurs. After specifying the matrix to be calculated on line 185 – 187, the multiplication occurs on line 188. Next, the addition of bias occurs in line 191–194.

3.2.3 Affine

The affine layer is a fully-connected layer that connects all inputs to all output neurons. The following equation executes it.

```
82  ConstMatrixMap<T> mx(x, i_row_, i_col_);
83  ConstMatrixMap<T> mw(w, w_row_, w_col_);
84  MatrixMap<T> my(y, o_row_, o_col_);
85  my = mx * mw;
86  if (inputs.size() == 3) {
87    // With bias
88    const T *b = inputs[2]->get_data_pointer<T>(this->ctx_);
89    my.rowwise() += ConstRowVectorMap<T>(b, o_col_);
90  }
```

Figure 3.9: `src/nbla/function/generic/affine.cpp`

$$o = Wi + b$$

Where o is the output, W is the weight, i is the input, and b is the bias.

Similar to the convolution layer, the affine layer uses multiplication and addition of matrices. **Figure 3.9** shows the source code of the affine layer. We can observe the similarity with the convolution layer. After specifying the matrix to be calculated on line 82 – 84, the multiplication occurs on line 85. Next, the addition of bias occurs on line 86–90.

3.2.4 Max-pooling

The max-pooling layer outputs the maximum value from the region specified by the kernel shape. This layer requires to specify the size of the kernel, stride, and padding. Stride is the number that specifies the pixels shifts over the input matrix. Padding is the number that specifies the size of zero paddings added to the ends of the input matrix before the pooling process.

Figure 3.10 shows the source code of the max-pooling layer. The variable 'y' represents the output matrix, and 'x' represents the input matrix. The code in lines 91 through 96 computes the range of region to found out the maximum value. Since the input is three dimensions in the code, it finds the range of regions for each dimension. The code in lines 97–110 finds the maximum value in the region obtained earlier. We can see that NNC access array using multiple statements.

```
for (y_idx[0] = 0; y_idx[0] < y_shape[0]; ++y_idx[0]) {
  for (y_idx[1] = 0; y_idx[1] < y_shape[1]; ++y_idx[1]) {
    for (y_idx[2] = 0; y_idx[2] < y_shape[2]; ++y_idx[2]) {
      for (int a = 0; a < 3; a++) {
        pool_start[a] = y_idx[a] * stride[a] - pad[a];
        pool_end[a] = min(pool_start[a] + kernel[a], x_shape[a] + pad[a]);
        pool_start[a] = max(pool_start[a], 0);
        pool_end[a] = min(pool_end[a], x_shape[a]);
      }
      auto max_idx = (pool_start[0] * x_stride[0] +
                      pool_start[1] * x_stride[1] + pool_start[2]);
      auto max_val = x[max_idx];
      for (int i0 = pool_start[0]; i0 < pool_end[0]; ++i0) {
        for (int i1 = pool_start[1]; i1 < pool_end[1]; ++i1) {
          for (int i2 = pool_start[2]; i2 < pool_end[2]; ++i2) {
            auto idx = i0 * x_stride[0] + i1 * x_stride[1] + i2;
            if (max_val < x[idx]) {
              max_val = x[idx];
              max_idx = idx;
            }
          }
        }
      }
      *m++ = max_idx;
      *y++ = max_val;
    }
  }
}
```

Figure 3.10: src/nbla/function/generic/max_pooling.cpp

Chapter 4

Model Design for SystemBuilder

After generating and training a network in NNC, we convert the model for SystemBuilder to implement on FPGA. This chapter will discuss this model design for SystemBuilder using network structure and trained parameters from NNC.

4.1 Overall Design

To use SystemBuilder, we write the System DeFinition (SDF) file that describes the overall model design. **Figure 4.1** is the example showing the SDF file we designed. Communication primitives are represented as an ellipse, and functional units are represented as a rounded rectangle. The arrow leaving the unit represents the write function toward the communication primitive, and the incoming arrow represents the read function of the communication primitive. We use memory primitive to send input data to a hardware process and blocking channel primitive to send start signal and receive the inference result. If multiple hardware

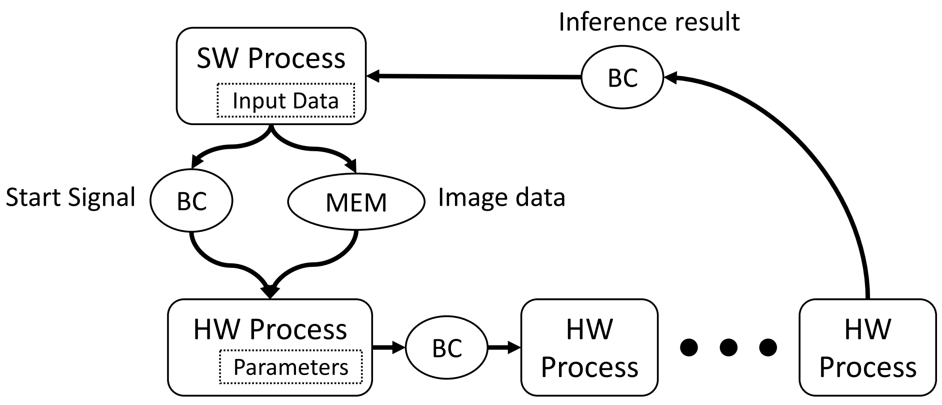
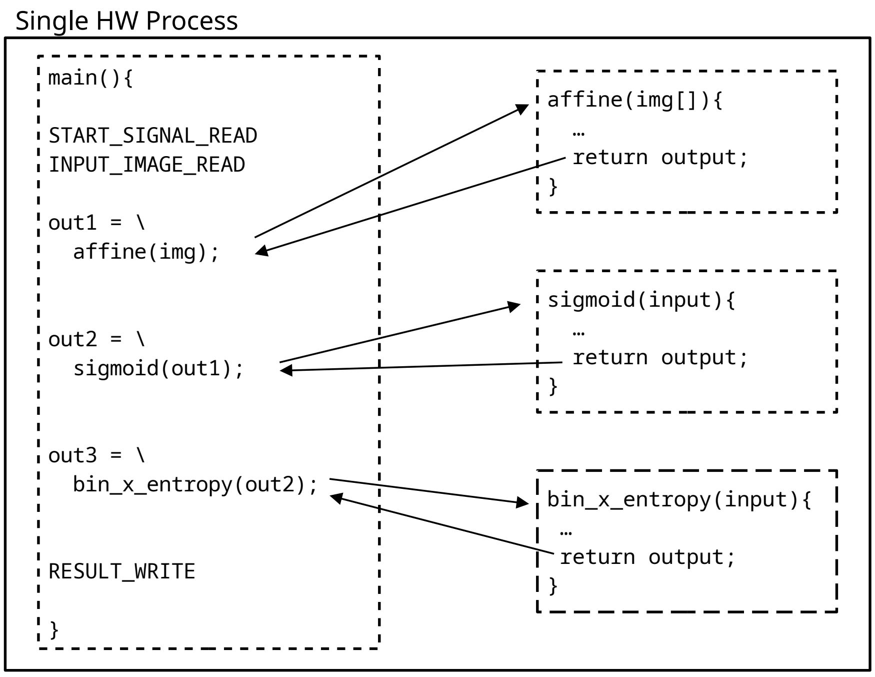


Figure 4.1: Overall design of the implemented model



```
main(){

START_SIGNAL_READ
INPUT_IMAGE_READ

out1 = \
  affine(img);


out2 = \
  sigmoid(out1);


out3 = \
  bin_x_entropy(out2);


RESULT_WRITE

}
```

```
affine(img[]){
  …
  return output;
}
```

```
sigmoid(input){
  …
  return output;
}
```

```
bin_x_entropy(input){
 …
 return output;
}
```

Figure 4.2: Single hardware process

processes are declared as **Figure 4.1**, we use blocking channel primitive to communicate between hardware processes. **Figure 3.1** is a part of the SDF file we designed.

The input data is prepared as text and read as a global array variable in the software process, and the parameters are also prepared as text and used as global variables in each layer code.

As shown in **Figure 4.1**, the hardware returns inferred result to SW process from the received data. For hardware process design, we take two approaches.

Design 1 The first approach uses a single hardware process, as shown in **Figure 4.2**. In this single process, several functions that play each layer's role are declared. It operates by calling each function inside the hardware process's primary function.

Design 2 The second approach uses several hardware processes, as shown in **Figure 4.3**. Each process has a single function that plays the role of each layer. However, not all layer is created as a process. We create a process that corresponds to the convolution, affine, and max-pooling layers. These layers are layers that have different sizes of input data and output data. The other layers, such as an activation layer, will be built-in in the previous layer's process. For example, if one matrix element is calculated in the convolution operation, this element value passes through the activation function and is sent to the next layer.

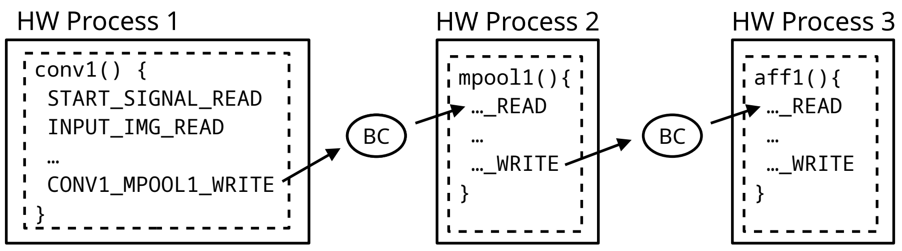


Figure 4.3: Multiple hardware processes

Each process transfers one element of the resulting matrix to the next process as it is computed. The next process receives input results and then operate. Data transfer between hardware processes uses blocking channel primitive.

4.2 Design of each layer

In the previous section, we discussed the overall model design. We also discussed how the hardware processes are configured. This section will discuss the actual implemented design of three-layer, convolution, affine, and max-pooling. These layers are separately designed to be hardware process in **Design 2** of section 4.1.

We will discuss the design using the pseudocode. However, addition and multiplication in the pseudocode used are far from real. Since we are handling quantized numbers in the implementation, we take different addition and multiplication method, which will be discussed in section 4.3.

4.2.1 Convolution

The following code is pseudocode for the convolution layer.

```
Receive results from the previous process

for (o_ch = 0; o_ch < output_channel; ++o_ch){
    for (o_h = 0; o_h < output_height; ++o_h){
        for (o_w = 0; o_w < output_width; ++o_w){

            result = 0;
            for (k_d = 0; k_d < kernel_dimension; ++k_d){
                for (k_h = 0; k_h < kernel_height; ++k_h){
                    for (k_w = 0; k_w < kernel_width; ++k_w){
                        result += input_element * kernel_element;
                    }
13                  }
14              }
15              // add bias
16              result += bias_element;
17
18              Modify result if needed (e.g. activation function)
19              Print result
20          }
21      }
22  }
```

The convolution layer is a layer that adds a bias to the sum of matrices multiplications. The triple for statement calculates each element of the output matrix sequentially (line 3–5). In the inner triple loop, we access each element of the input and the kernel (line 8–10). Then, we calculate the sum of the accessed elements' multiplications (line 11) and add a bias (line 16). Next, as we discussed in section 4.1, we built some layers in one of the two hardware designs, such as an activation layer in its previous layer. In this case, it is processed (line 18). Then finally, the code print the output element (line 19).

4.2.2 Affine

The following code is pseudocode for the affine layer.

```
 1  Receive results from the previous process
 2
 3  for (ch = 0; ch < output_channel; ++ch){
 4      result = 0;
 5      for (i = 0; i < input_size; ++i){
 6          result += input_element * parameter_element;
 7      }
 8      // add bias
 9      result += bias_element;
10
11      Modify the result if needed (e.g. activation function)
12      Print the result
13  }
```

An affine layer is written in a structure similar to the convolution layer. The outer for statement calculates each output matrix element sequentially (line 3). Inner for statement calculates the sum of the multiplications (line 5–7). Like the convolution layer, after adding the bias (line 9), we modify the result if needed and print the result (line 11–12).

4.2.3 Max-pooling

The following code is pseudocode for the max-pooling layer.

```
 1  Receive results from the previous process
 2
 3  for (o_ch = 0; o_ch < output_channel; ++o_ch){
 4      for (o_h = 0; o_h < output_height; ++o_h){
 5          for (o_w = 0; o_w < output_width; ++o_w){
 6
 7              local_max = -128;
 8              for (k_h = 0; h < kernel_height; ++k_h){
 9                  for (k_w = 0; w < kernel_width; ++k_w){
10                      temp = input_element;
11                      if (temp > local_max) {
12                          local_max = temp;
13                      }
14                  }
15              }
16
17              Modify the local_max if needed (e.g. activation function)
18              Print the local_max
19          }
20      }
21  }
```

Like other designs described above, the code calculates each element of the output matrix sequentially using for statement (line 3–5). Next, it founds the largest element within the range specified by the kernel using double for statement (line 7–15). Like other designs, we modify the result and print the result (line 17–18).

4.3 Implementation of operations with quantized data

As discussed in section 3.2, the FixedPointQuantize layer's output is a float in NNC. Therefore the other layer can be a generic usable design without considering the input is quantized or not. However, we want to eliminate the use of floating-point arithmetic from our models. Therefore, we should design the layers to handle only an integer type.

Since we set the quantization step size (δ) as the power of two, data can be expressed as follows:

$$\text{quantized data in original scale} = \text{quantized data} \times \delta, \quad \delta = 2^{-n} \ \ (n = 1, 2, \ldots)$$

While ‘quantized data’ is an integer value and δ is the quantization step.

Our model only handles this ‘quantized data’. Therefore, rather than the concept of passing the float data through the FixedPointQuantize layer like NNC, we designed other layers to include the concept of FixedPointQuantize layer.

Each layer receives the quantized integer as input and outputs as an integer. Quantization steps of input, output, and parameters can be different. Consequently, we take this into account when designing. This subsection will cover the implemented methods of operations with quantized data.

Addition

Let a and b be defined as quantized numbers, and let A and B be values expressed in the original scale of a and b. The following equations express these relations.

$$A = a \times \delta_a$$
$$B = b \times \delta_b$$

Since we use delta as the power of two in our implementation, let us also define δ_a and δ_b as the following equations.

$$\delta_a = 2^{-\alpha}$$
$$\delta_b = 2^{-\beta}$$
$$\alpha,\ \beta \in \mathbf{Z}$$

$A + B$ will be expressed as the following equation.

$$A + B = (a \times 2^{\beta-\alpha} + b) \times 2^{-\beta}$$

Assume, without loss of generality, that δ_a is greater than or equal to δ_b. Then $a \times 2^{\beta-\alpha} + b$ will be an integer. This value will be the result of the addition between quantized numbers. This value has the following meanings: The result is the addition between a with adjusted quantization step and b. The result’s quantization step will be a lower value among the operand’s quantization step, which is $2^{-\beta}$ in the above equation. The following codes can express this process in the implementation.

```
a_alter = a;
if (a < 0) {
    a_alter = -((-a_alter) << ( β - α  ));
4 } else {
5     a_alter = a_alter << ( β - α  );
6 }
7 result = a_alter + b;
```

To decrease a's quantization step by divided by $2^{\beta-\alpha}$, we increased a multiplied by $2^{\beta-\alpha}$. To maximize the performance, we use a bitwise shift operation. The behavior of bitwise left shift operation with a negative number is undefined in C standard [22]. Therefore, we use the if statement as line 2 to determine the negative number to solve this problem.

Variable '`a_alter`' and '`result`' should be declared in wide-enough data type to avoid integer overflow in operation.

Multiplication

Let a and b use the same definition as in **Addition**. Multiplication of a and b can be expressed as the following equation.

$$AB = ab \times (\delta_a \times \delta_b)$$

This equation means the result of a multiplication is $a \times b$, and the quantization step size will be the multiplication of the operand's quantization step. The following codes can express the multiplication in the implementation.

```
result = ((int32_t) a) * b;
```

We typecast the variable to wider data types to avoid overflow. Moreover, the variable '`result`' should be declared in a wider data type.

Typecasting to narrower data types

We store the operation result in a wider data type to implement addition and multiplication. We use a 32-bit integer to store the operation value during the process. After performing the operation required by each layer, typecasting to narrower data types is required. Each layer's output should be quantized to an 8-bit integer, not 32-bit. Moreover, we should also adjust the number's quantization step to match the output's one.

To perform this process, we use the following steps. First, we change the quantization step to match layer output. Second, we check the overflow and underflow.

Step 1 Changing the quantization step can be expressed as the following codes in the implementation.

```
1 if (result < 0) {
2         result -= (1 << (rescale_amount - 1));
3         result = -((-result) >> (rescale_amount));
4 } else {
5         result += (1 << (rescale_amount - 1));
6         result = (result) >> (rescale_amount);
7 }
```

In the code, it increases the quantization step by multiplied by $2^{\texttt{rescale\_amount}}$. We use the right shift operation to accomplish it. Since the right shift operation will truncate the `result`, we added a number to round it (line 2 and 5). For example, let us apply the code above to the following variable:

$$\texttt{result} = \texttt{0b 0100 0100}$$
$$\texttt{rescale\_amount} = 4$$

The variable '`result`' then will be "`0b 0000 0101`".

In the case of decreasing the quantization step, we can use the left shift operation method described in **Addition**. However, in the case study, the quantization step of the output always has been large or equal to the quantization step of an internal variable like '`result`'.

An equivalent code in NNC An equivalent task happens in NNC. **Figure 3.7** is the part of the source code of the FixedPointQuantize layer. The equivalent part is line 59 – 62 of this figure.

- The concept that determines the sign of '`result`' in our model (if statement in line 1) is equivalent to lines 59, 60, and 62 of **Figure 3.7**. We design to divide the case with an if statement, but developers of NNC designs to handle with absolute values first and then handle sign next.
- The concept avoids truncating while left shifting is implemented in lines 2 and 5 on our models. This is equivalent to line 61 of **Figure 3.7**. The developers of NNC designs to add 0.5 before the multiplication of quantization step (`delta_`).

Step 2 Checking the overflow and underflow can be expressed as the following codes in the implementation.

```
if (result < MIN) {
        output = MIN;
} else if (result > MAX) {
        output = MAX;
} else {
        output = result;
}
```

Where `MIN` is a minimal value, and `MAX` is a maximum value of `output`'s data type. For example, if `output` is declared in an unsigned 8-bit integer, it will be `0` and `255`. Furthermore, if `output` is declared in a signed 8-bit integer, it will be `-128` and `127`.

An equivalent code in NNC An equivalent part in NNC is line 54 – 57 in **Figure 3.7**. However, in NNC, checking the overflow and underflow can be performed before step 1 process because the input and output are on the same scale.

Chapter 5

Case Study

5.1 Environment

We used environment and programs as listed in **Table 5.1**.

5.2 DNN Models

For the case study, we use three following DNN models provided by NNC.

- Logistic regression
- Binary CNN
- LeNet

These are the network that classifies the handwriting images of Arabic numerals. Logistic regression and binary CNN classify handwriting images of 4 or 9, which binary stands for in the name of binary CNN. Unlike the previous two networks, LeNet classifies handwriting images of 0 to 9.

These three network receive an image with 28x28 pixels of one color channel as an input. We present the detail of each network in section 5.5. Moreover, all networks we implemented can be referred to at reference [23].

Table 5.1: Environment and programs

Purpose	Category	Name	Version
Running DNN framework	Host OS	Windows 10 Pro	19041.746
	DNN Framework	Neural Network Console	1.8.7502.9306
FPGA Implemention	Host OS	Ubuntu	18.04.05
	System Generator	SysGen	2.0
	HLS Tool	Vivado HLS	2019.1
	Compiler	GCC	6.3.1
	RTL Emulator	Questa Sim	2019.10
	RTOS Simulator	QEMU	2.11.1
	Terminal Multiplexer	GNU Screen	4.06.02
	FPGA Board	Zybo Z7-20	-
Other Work	Host OS	Arch Linux	-
	Image Modifier	ImageMagick	7.0.10.58
	CLI for nnabla	nnabla_cli	1.13.0

5.3 Training the network from Neural Network Console

5.3.1 Quantization

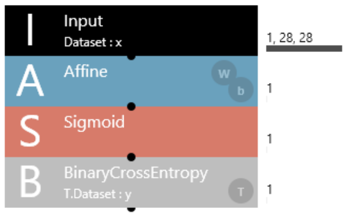
Using floating-point arithmetic on FPGA would have performance degradation compared to using an integer due to higher logic requirements and speed reduction. Therefore, we did not want to use any floating-point arithmetic in our models. Since Neural Network Console supports quantization from the stage of training, we could eliminate the use of floating-point arithmetic from our models. We modified the networks to quantize the parameters and the data transfer between layers before training the network.

Figure 5.1 shows the example of quantization in Neural Network Console. As the image shows, the FixedPointQuantize layer is inserted between other layers to quantize activations, and it is inserted in front of the parameter layer to quantize parameters. In this method, we were able to quantize all parameters and activations.

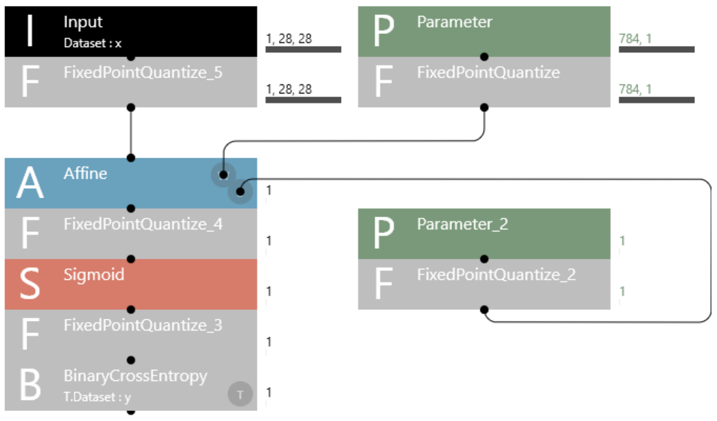
The following equation is the equation we covered in section 4.3. In the case study, we quantized all parameters and activations to 8-bit integers, and we set the quantization step size (δ) as the power of two.

$$\text{quantized data in original scale} = \text{quantized data} \times \delta, \quad \delta = 2^{-n} \ (n = 1, 2, \ldots)$$

'quantized data' is integer in the equation and δ stands for the size of 1 bit. Since we are using δ as the power of two, we can calculate between the quantized data with low cost



(a) Before



(b) After

Figure 5.1: Example of quantization in Neural Network Console

Table 5.2: The number of images used and accuracy of the network

Network	Training	Evaluation	Epoch	Accuracy
Logistic regression	1,500	500	100	95.2 %
Binary CNN	1,500	500	100	98.0 %
LeNet	60,000	10,000	10	99.2 %

while implementing to FPGA. It is because we can use the shift operator for adjusting δ. The detail with the operation with quantized numbers is discussed in section 4.3.

5.3.2 Training and Accuracy

We use MNIST dataset [24] to train and evaluate the networks. **Table 5.2** shows the number of images used for training and evaluation. The accuracy measured in the evaluation are also listed in the same table.

5.4 Implementation process flow

5.4.1 Preparing input data

As we have discussed in section 5.3.2, we use the MNIST dataset to train and evaluate. Neural Network Console downloads and converts the MNIST dataset in PNG format to use it. We use the same PNG format of the MNIST dataset to make it easier to compare the inference

Signature	
File Size	
Reserved	Reserved
File Offset to The Pixel Array	

Figure 5.2: Bitmap File Header

results. Our main purpose is not on reading PNG images but implementing a network in FPGA. Therefore, we convert the PNG image to text for development convenience.

To prepare the image as a text, we read each pixel of an image. For the convenience of reading pixels of an image, we convert the image to BMP format. We can have the same image data because PNG and BMP are both lossless formats. We use a tool named ImageMagick [25]. Following command was used.

```
convert -compress none <input>.png <output>.bmp
```

The '`-compress none`' option defines not to compress the image. It is required because by default ImageMagick version 7.0.10.58 compress BMP image with a run-length encoding algorithm, making it harder to read pixels.

Next, we built the program that reads each pixel of the BMP image to print it. Since the image data is not compressed, the program can open the image as binary and just read each pixel. **Figure 5.2** shows the file header structure of BMP file. The width of the figure is 4 bytes. Therefore, the BMP format stores the offset of the pixel array at 0x0A. After reading the offset in the header, we are able to access its data.

5.4.2 Preparing parameters

After modifying the network in section 5.3.1, and training in section 5.3.2, we implement it to FPGA. NNC stores the network in a binary file called '`results.nnp`'. To extract the parameter from this file, we used `nnabla_cli` [26]. `nnbla_cli` is the command-line interface of Neural Network Libraries, which NNC is built on. We ran the following command to decode parameters.

```
nnabla_cli decode_param -p results.nnp -o output_dir
```

This command will print out the parameters as files to '`output_dir/`'. Data of each `Parameter` layer will be created under the directory. For example, if we decode parameters of **Figure 5.1b**, '`Parameter.txt`' and '`Parameter_2.txt`' will be created. **Figure 5.3** is an example of decoded parameter text files. It includes the first nine lines of the parameter

```
(16, 1, 7, 7)
0.07549327611923218
0.051595233380794525
0.023067651316523552
0.09380219876766205
0.10407224297523499
-0.027228567749261856
0.058655522763729095
-0.03321220353245735
```

Figure 5.3: Example of decoded parameter

file. The first line indicates the matrix size of the parameter, and from the second line, the contents are recorded.

As shown in this figure, NNC stores the parameter as a float. Therefore, we quantized it to integer and stores it for use in the implementation. The parameter will be implemented as a C array in the hardware processes.

5.4.3 Designing functional units and the SDF file

We design functional units and the SDF file as discussed in chapter 4. After designing the functional units and SDF, we simulated the design.

5.4.4 Functional level simulation

To test the design at a functional level, we simulate the design by synthesizing all units as software processes. It is useful for testing the output result of the design. If all units were synthesized as software, simulation time could be shortened since it does not go through a process such as high-level synthesis of hardware or emulating hardware using an RTL simulator.

We created a new SDF file that declares all functional units in software processes to run this simulation. We synthesize the design with the following command.

```
SysGen -sdfy <SDF file> -dobs -s zybo_z7_20 -bst vivado
```

Each option of the command is discussed in section 3.1.9. The RTOS is not specified in the command, but ITRON is used as a default for this board.

The command specifies to perform HLS. However, since there is no hardware process declared in the SDF file, HLS will not occur in this process. The command will create a new directory with the value declared as the system name (e.g. line 1 in **Figure 3.1**). For the functional level simulation, we use *sp* as the system name. The directory '`sp/`' now has a result of synthesis. Next we change our working directory to '`sp/fmp/`'. Then we simulate the design with the following command.

```
make runq
```

This command compiles the software processes and runs QEMU to emulate RTOS with software processes. Next, we compare the received results with inference from NNC to determine whether the inference results were the same.

5.4.5 Co-simulation

To debug the design, we run co-simulation. In the SDF file, the inference function must be partitioned as a hardware process in order to test the hardware behavior. To distinguish this design from the previous design in functional level simulation, we use *sp_hw* as the system's name, which indicates that the system has hardware processes.

Next, we synthesize the design with the same command discussed in functional level simulation. Unlike in the functional level simulation, we have declared the hardware processes. Therefore this command runs the HLS tool. We use Vivado for high-level synthesis.

```
SysGen -sdfy <SDF file> -dobs -s zybo_z7_20 -bst vivado
```

After changing the working directory to '`sp_hw/fmp/`', we executed the following commands.

```
make runsim &
make runq
```

The first command runs Questa Sim. Questa Sim compiles HDL files and simulates the hardware, and the second command compiles the software processes and runs QEMU to emulate the software processes. Next, we check the signal wave to debug the design. We run the following commands at Questa Sim's transcript.

```
add wave -position end sim:/sp_hw/*
```

The wave window will now print the signal wave on its window. Then, we ran the following commands at Questa Sim's transcript to start the RTL simulation.

```
run -all
```

Finally, after sending a start signal at QEMU, we observe the result of co-simulation.

5.4.6 Implementation on the real-machine

After co-simulation, we implement the design on the real-machine to test inference and performance. Like other processes discussed above, we synthesize the entire design and high-level synthesize the hardware with the following command.

```
SysGen -sdfy <SDF file> -dobs -s zybo_z7_20 -bst vivado
```

Next, we run logic synthesis. SystemBuilder provides Vivado's batch script for logic synthesis. To use this, copy the files in '`Zynq/ZYBO_Z7_20/`' of the SystemBuilder installation path to '`sp_hw/zynq/`' and run `run.sh`.

After logic synthesis, we connect the FPGA board to the computer and run the following command as the root account. This command implements the bitstream on the real-machine.

```
make runf
```

Next, we communicate with the serial port to obtain the results. To communicate with the serial port on Linux, we did the following:

Suppose FPGA board's serial port is connected to '`/dev/ttyUSB1`'. We run the following command as the root account to display '`/dev/ttyUSB1`'.

```
screen /dev/ttyUSB1 115200
```

This command connects '`/dev/ttyUSB1`' with baud rate at 115200. Then we are able to communicate with the board via GNU Screen.

5.5 Design decision of each case study

5.5.1 Logistic regression

Logistic regression is a simple network that is formed with two layers. This network can be referred to from **Figure 5.4**. For hardware design, we use **Design 1** discussed in section 4.1. This hardware design is to use a single hardware process with multiple functions inside. Logistic regression is the first case study with the top priority of implementing it in an FPGA. Single hardware process design is simple comparing with multiple hardware processes design. Thus it is relatively effortless to implement the model.

To implement the activation function, we use the Taylor series of activation function that can be referenced in [27]. However, we could not obtain the correct result by following the equation in [27]. We expect this is because the calculations have occurred with the low precision of 8-bit integers. We made exception handling for the input that did not match the correct result to solve this problem.

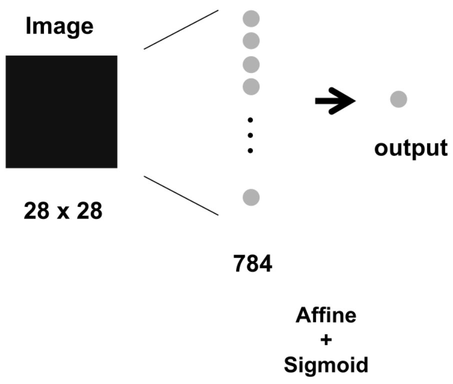


Figure 5.4: Logistic regression.

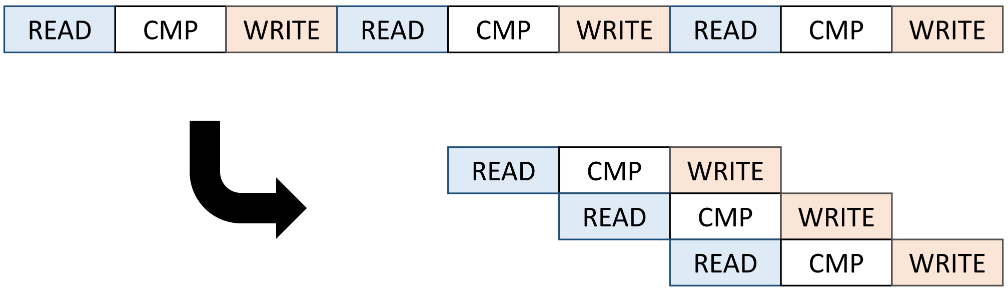


Figure 5.5: Image of pipelining the loop

Pipelining the loop

We use a loop statement to implement the matrix multiplication in an affine layer. We have tested pipelining this loop statement using HLS. **Figure 5.5** depicts the concurrent execution by pipelining operations, which will shorten the computation time. Its effectiveness will be discussed in the results (section 5.6).

5.5.2 Binary CNN

After successful implementation of logistic regression in section 5.5.1, we implement another network. We search for the DNN Model, which includes the convolution layer. It is because we think that the convolution layer has room for further optimization with a loop statement. This room could help to test the HLS tool's optimization as future work. We use binary CNN from example networks provided by NNC [23]. This network can be referred from **Figure 5.6**.

For hardware design, we use **Design 2** discussed in section 4.1. This hardware design is to use a multiple hardware process that plays each layer's role. As discussed in section 4.1,

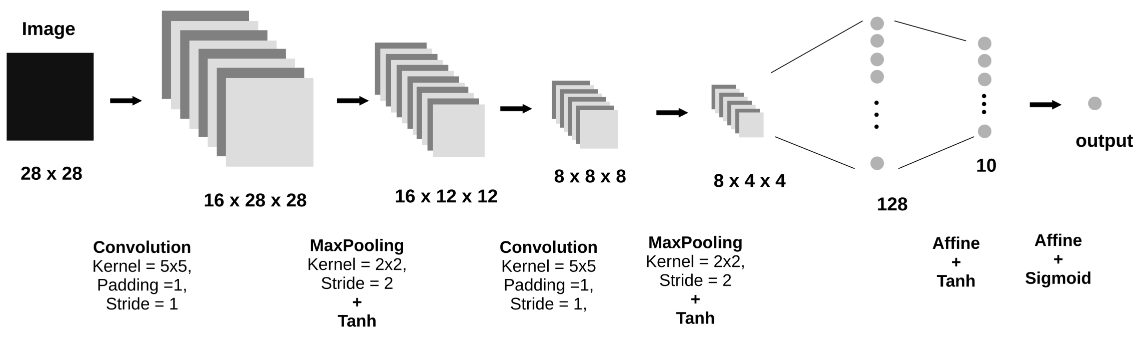


Figure 5.6: Binary CNN

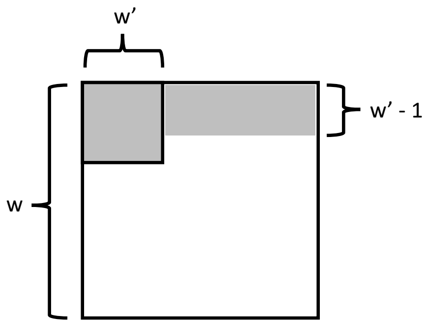


Figure 5.7: Image of buffer size

we build six hardware processes for binary CNN, corresponding to two convolution layers, two max-pooling layers, and two affine layers. Other layers are designed to be built-in to the previous layer. The reason for this design is that it will be able to test the pipelining at the layer level, which will be discussed later.

We use a lookup table for the activation layer such as Tanh and Sigmoid. As discussed in section 5.5.1, calculating the activation function with the low precision of 8-bit integers is challenging to achieve the correct result.

Pipelining the process

In the implementation of binary CNN, we use six hardware processes. Separating the hardware processes makes it possible to pipeline at the process level. We design each process to accumulate input data in a buffer until one operation is possible. When there is enough data in the buffer, it executes the operation and export the result. **Figure 5.7** depicts this process. Suppose the outer square presents the input matrix and the inner square presents the kernel matrix. The colored part of the figure should be accumulated in the buffer to

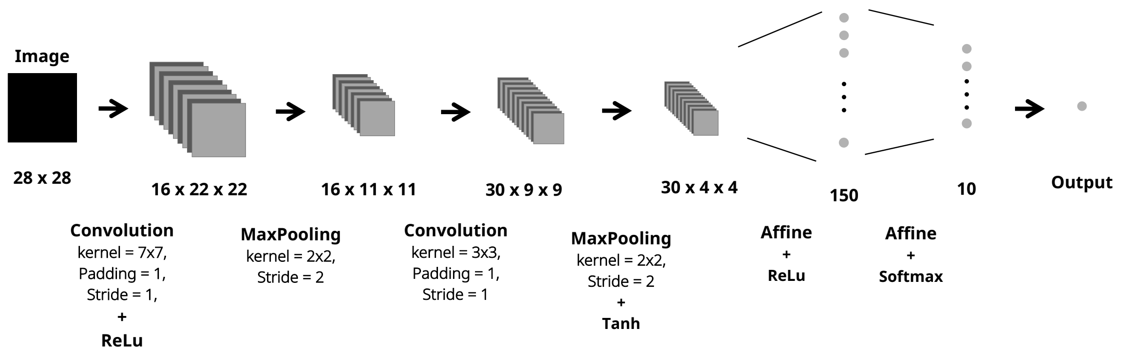


Figure 5.8: LeNet

calculate one element of the output matrix. The size of the colored part is as follows:

$$w * (w' - 1) + w'$$

This will be the size of the buffer. After calculating one output element, the buffer should be shifted to store the next input element. In a case like **Figure 5.7**, the buffer will be shifted by w'. After shifting, the buffer will receive a new input element. Likewise, if the buffer is full, the calculation of one output element will occur.

The method described above is slightly different from the actual implementation. The max-pooling layer is implemented in the same way as above, but the convolution layer is different. In the case of the convolution layer of binary CNN, the input matrix is three dimensions, and the kernel is also three dimensions. The concept of accumulating the buffer is the same, though there is no shifting to discard unnecessary elements. This is because unnecessary elements are not arranged in the first few elements as in two dimensions, which disallows discarding.

Using this method, we are able to pipeline the processes. We compare the pipelined and non-pipelined results.

5.5.3 LeNet

For the final case study, we implement the LeNet. Unlike the other two, LeNet classifies ten kinds of input data. This network can be referred to from **Figure 5.8**.

As with the design of binary CNN, we use six hardware processes to pipeline the process. We use a lookup table for the Tanh activation function. In the case of ReLu, we followed the definition of the function: if the input value is below 0, the output is 0.

Table 5.3: A result of case study including loading time of input image.

Network	Optimization	Latency	Throughput	BRAM Usage	LUT as Logic Util Rate
Logistic regression	-	347 μs	2884 fps	162 kB	16.06 %
	pipeline (loop)	330 μs	3031 fps	162 kB	16.61 %
Binary CNN	-	18772 μs	53.3 fps	200 kB	18.19 %
	pipeline (process)	18092 μs	55.3 fps	180 kB	18.65 %

Target: ZYBO@100MHz
Latency for Logistic regression: `average execution time of 100-discussion`
Latency for Binary CNN: `average execution time of 10-discussion`

Table 5.4: A result of case study without loading time of input image.

Network	Optimization	Latency	Throughput	BRAM Usage	LUT as Logic Util Rate
Logistic regression	-	61.5 μs	16260 fps	162 kB	16.60 %
	pipeline (loop)	45.5 μs	21978 fps	162 kB	16.61 %
Binary CNN	-	18489 μs	54.1 fps	200 kB	15.23 %
	pipeline (process)	17809 μs	56.2 fps	180 kB	15.68 %

Target: ZYBO@100MHz
Latency: `execution time for 1-discussion`

5.6 Results

5.6.1 Logistic regression and binary CNN

For logistic regression and Binary CNN, we have confirmed that our implementation shows the same inference as the original. Next, we measure the performance of our implementation. We measure two kinds of data. One is the total latency from data transmission to output of the FPGA. This data is listed in **Table 5.3**. We calculate the average of 100 inferences for logistic regression, and for Binary CNN, we use the average of 5 inferences. The other measured performance is latency without input data transmission latency, which is the network's pure computation latency. This data is listed in **Table 5.4**

The optimization column in these tables is information for comparing the applied optimization method. 'pipeline (process)' means that it pipelined at the process level, and 'pipeline (loop)' means that it pipelined one loop statement. This optimization method is discussed in section 5.5.

We have tested the optimization technique in the logistic regression's implementation using the HLS tool described in section 3.1.7. We have pipelined the loop statement in the affine layer. As a result, the throughput improved from 2884 fps to 3031 fps, resulting in a five percent performance improvement (**Table 5.3**). However, focusing on the computation time itself, throughput improved from 16260 fps to 21978 fps, resulting in a 35 percent performance improvement (**Table 5.4**).

In the implementation of Binary CNN, we experiment with pipelining in processes. As a result, throughput increased from 53.3 fps to 55.3 fps, resulting in a 3.8 percent performance improvement (**Table 5.3**). Performance improvement focusing on the computation time itself was increased from 54.1 fps to 56.2 fps, resulting in a 3.9 percent performance improvement (**Table 5.4**).

5.6.2 LeNet

In the implementation of LeNet, the inference results did not match with the NNC. When we check the inference result of 10000 images, six were different from NNC.

5.7 Discussion

5.7.1 The communication time

When comparing the latencies in **Table 5.3** with the latencies in **Table 5.4**, there is a consistently constant difference of about 284 μs. We believe that this consistently constant difference of latency is the time require for image data transmission to the FPGA.

The latency of binary CNN, including data transmission, is 18489 μs (without pipelining). 284 μs is about 1.0 percent of the entire latency for binary CNN. However, the entire latency of logistic regression is 374 μs (without pipelining), which leads to 284 μs to be about 82 percent of the entire latency for logistic regression. Transferring of input data is being a huge bottleneck for logistic regression. Testing other communication primitives such as prefetch BC will be worthy as future work.

5.7.2 Activation function

We built the sigmoid function using the Taylor series in section 5.5.1. We suspected the loss using the Taylor series would be negligible. We thought the loss in quantizing the results would be much bigger to be able to hide the loss in the Taylor series. We used multiple Taylor series for multiple domains. We found out that the output does not monotonically increase at the boundary of each domain. We expect this is due to the low precision calculations of 8-bit integers.

To solve this problem, we took a different approach in section 5.5.2. We prepared 256 ($= 2^8$) size output an array that corresponds to 256 cases of input. We made a lookup table.

Apart from the difficulty of making, but in terms of performance and circuit size, we believe that comparing the lookup table and implementing it as a function is worthy of discussion.

5.7.3 Pipelining

Observed waveform of binary CNN

In binary CNN, we have tested the pipelining at the process level. We observed the waveform to debug its behavior.

Figure 5.9 shows the signal between hardware processes conv1, mpool1, and conv2. These processes represent convolution, max-pooling, and convolution layer. The signal in the figure represents data transmission between processes. The signal marked as two in **Figure 5.9** represents the data transmission from mpool1 to conv2. As **Figure 5.9** shows, there is a gap between data transmission between mpool1 and conv2. This gap's latency represents the latency of accumulating the buffer of the mpool1 process. This figure shows the mpool1 process begins to calculate and print output element to next process after accumulating input element.

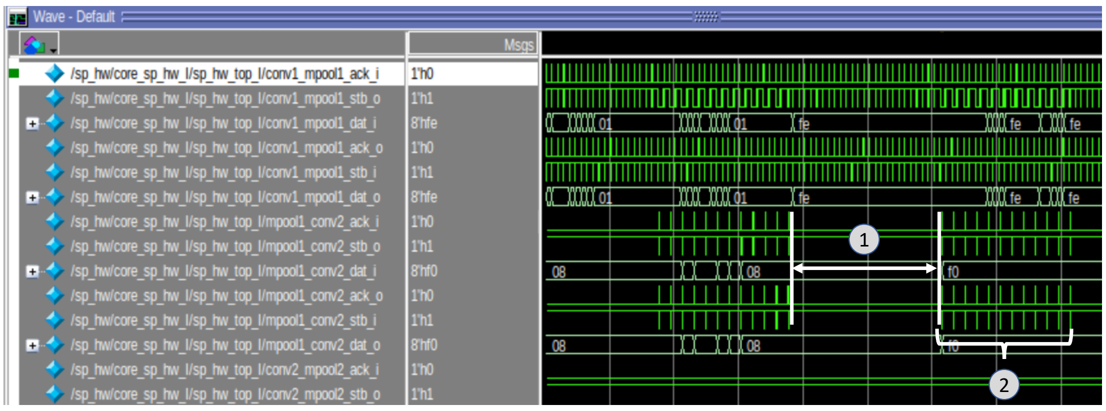
Figure 5.11 is a waveform of binary CNN which is not pipelined. In **Figure 5.11**, mpool1 prints output element after receiving all input elements from conv1. This is clearly different behavior from **Figure 5.9**.

However, there is a limitation on our process level pipelined model. **Figure 5.10** shows the same model as **Figure 5.9** but with larger scale. The gap marked as one is a latency before the conv2 process begins to print the result. **Figure 5.10** shows conv2 begins to print result after receiving most of the mpool1's output. The convolution process of our DNN model receives three dimensions of the input matrix. Moreover, the convolution process requires matrix multiplication of a kernel matrix with the same depth as the input matrix. This leads requiring of accumulating most of the elements in the pipelining design.

Efficiency

In order to discuss the efficiency of the pipelining, we will discuss the performance gains and the increased LUT usage from pipelining. We will discuss the performance gain with the result without data transmission of an input image, represented in **Table 5.4** and discussed in section 5.6.

Logistic regression We observe about 35 percent improvement of computation speed by pipelining on loop in the process. At the same time, logic LUT usage rate increased from 16.60 % to 16.61 % (**Table 5.4**). The increase is about 0.1 percent. We believe that it

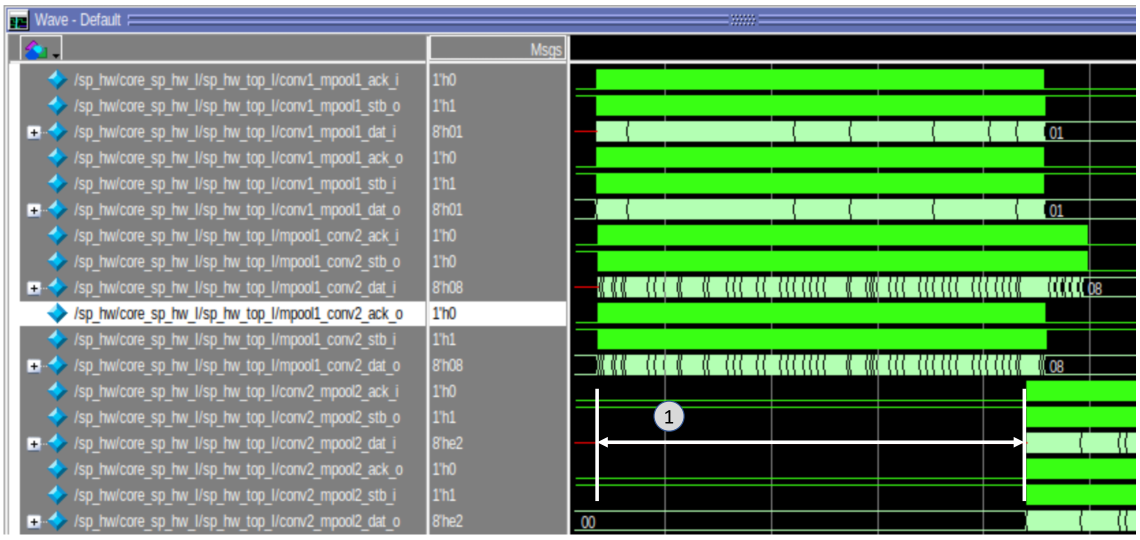


```
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_ack_i   1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_stb_o   1'h1
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_dat_i   8'hfe
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_ack_o   1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_stb_i   1'h1
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_dat_o   8'hfe
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_ack_i   1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_stb_o   1'h1
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_dat_i   8'hf0
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_ack_o   1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_stb_i   1'h1
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_dat_o   8'hf0
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv2_mpool2_ack_i   1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv2_mpool2_stb_o   1'h1
```

1: Latency of accumulating the buffer at mpool1 process.

2: Data transmission from mpool1 to conv2.

Figure 5.9: The waveform of the model pipelined at the process level



```
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_ack_i   1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_stb_o   1'h1
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_dat_i   8'h01
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_ack_o   1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_stb_i   1'h1
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_dat_o   8'h01
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_ack_i   1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_stb_o   1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_dat_i   8'h08
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_ack_o   1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_stb_i   1'h1
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_dat_o   8'h08
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv2_mpool2_ack_i   1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv2_mpool2_stb_o   1'h1
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv2_mpool2_dat_i   8'he2
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv2_mpool2_ack_o   1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv2_mpool2_stb_i   1'h1
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv2_mpool2_dat_o   8'he2
```

1: Latency of accumulating the buffer at conv2 process.

Figure 5.10: The overview waveform of the model pipelined at the process level

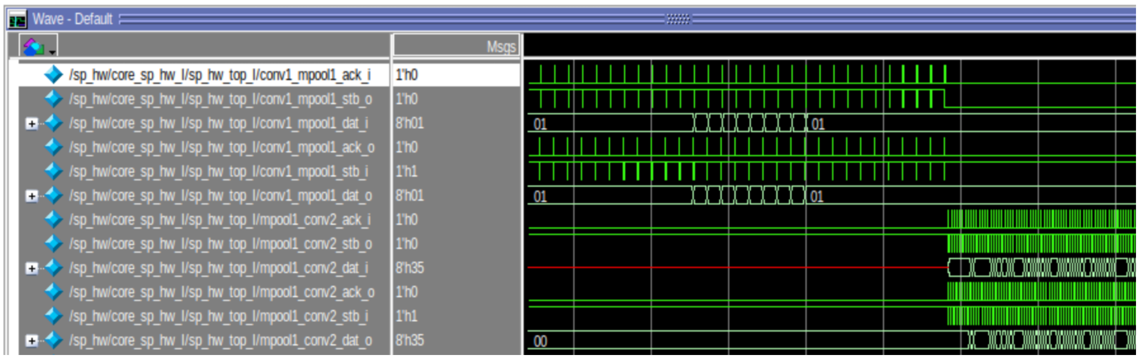

Wave - Default
Msgs
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_ack_i 1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_stb_o 1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_dat_i 8'h01
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_ack_o 1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_stb_i 1'h1
/sp_hw/core_sp_hw_l/sp_hw_top_l/conv1_mpool1_dat_o 8'h01
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_ack_i 1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_stb_o 1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_dat_i 8'h35
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_ack_o 1'h0
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_stb_i 1'h1
/sp_hw/core_sp_hw_l/sp_hw_top_l/mpool1_conv2_dat_o 8'h35
01
01
00

Figure 5.11: The waveform of the model that did not pipeline the process

showed high-efficiency use of logic LUT when considering the performance improvement. The amount of BRAM usage is the same as 160 kB.

Binary CNN We observe about 3.9 percent performance improvement by pipelining at the process level. At the same time, logic LUT usage rate is increased from 15.23 % to 15.68 % (**Table 5.4**). The increase is about 0.45 percent. It is relatively inefficient compared to the above case, considering the degree of performance improvement.

However, BRAM usage has decreased from 200 kB to 180 kB, which can be said it is an advantage of the optimization. We suspect the BRAM has decreased because the buffer accumulating the previous layer results has decreased in each process.

5.7.4 Problems of LeNet

In the results, we found out the inference results of LeNet were different from NNC in few cases. To find out the cause, we did the following:

To find out the result of one matrix multiplication, we design a network only with one affine layer and quantization layers, as **Figure 5.12**. If we disable the affine layer's bias, the output will be the result of multiplying the input matrix and parameter matrix. This result can make us understand the matrix multiplication behavior in NNC. Matrix multiplication is also used in the convolution layer. Therefore, observing a single affine layer can help us understand the overall operation of NNC.

Next, we implement this network in the same method as the case study. When comparing the result (matrix), about half of the matrix elements showed different values. The error is at most 1; hence it is not a big error. However, we believe the accumulation of errors during the operation results in difference in the output for some cases.

The reason for the error was not known.

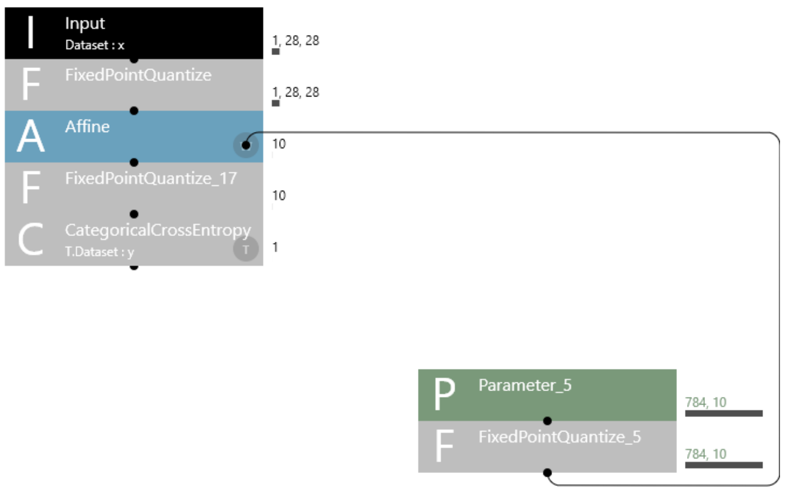


Figure 5.12: Affine layer-only network

Chapter 6

Conclusion

6.1 Summary

In this paper, we review the possibility of implementing existing DNN frameworks. To achieve this, we implement three networks created and trained from NNC. In order not to use floating-point arithmetic and have the same inference result, all the data in the network was quantized from the stage of training.

For the first two case studies, we confirm that inferences from implementation were the same as those from NNC. However third case study shows the inference did not match exactly. This is an area that must be fixed through further research.

We test two approaches for finding out an efficient design. One is pipelining the matrix array (Logistic Regression), and the other is pipelining the layers (Binary CNN). As a result, we found performance gain.

6.2 Future Work

We believe that contribution of this paper will help in the future development of a tool that automatically implements the network from NNC. This tool's image is presented in **Figure 6.1**. The development of this tool is our final goal.

These are some works that should be done in the future for this goal.

- Find out the reason why the inference results are different in LeNet.
- Optimize each layer using the feature provided by HLS, such as pipelining the loop. In which explained in section 3.1.7.
- Test various communication primitives described in section 3.1.4.

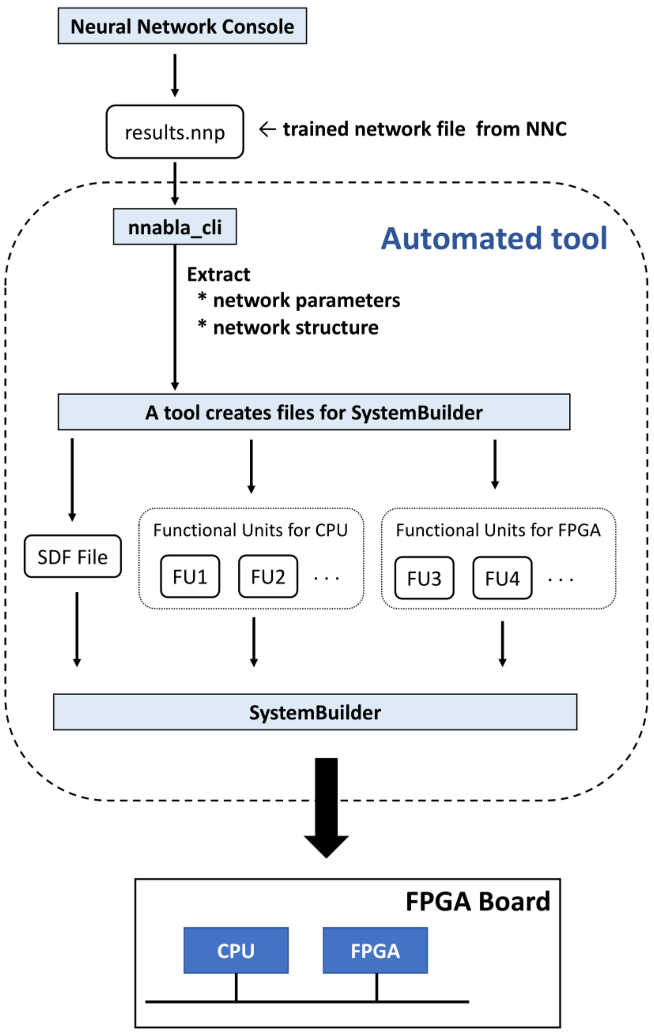


Figure 6.1: Automated tool

- Implement every layer that NNC supports.
- Build a tool that extracts network structures and create files for SystemBuilder.
- Explore a design of the network and make the optimal decision.

Bibliography

[1] Google LLC. Tensorflow. `https://www.tensorflow.org/`. Accessed: 2021-01-30.

[2] Facebook, Inc. PyTorch. `https://pytorch.org`. Accessed: 2021-01-30.

[3] Micron Technology, Inc. Memory speeds and compatibility. `https://www.crucial.com/support/memory-speeds-compatability`. Accessed: 2020-11-19.

[4] Micron Technology, Inc. GDDR6X. `https://www.micron.com/products/ultra-bandwidth-solutions/gddr6x`. Accessed: 2020-11-19.

[5] Google LLC. Tensorflow lite for microcontrollers. `https://www.tensorflow.org/lite/microcontrollers`. Accessed: 2020-11-16.

[6] Shinya Honda, Hiroyuki Tomiyama, and Hiroaki Takada. SystemBuilder: A system level design environment (in Japanese). *The IEICE Transactions on Information and Systems (Japanese Edition)*, 88(2):163–174, Feb. 2005.

[7] Shinya Honda, Hiroyuki Tomiyama, and Hiroaki Takada. RTOS and codesign toolkit for multiprocessor systems-on-chip. In *2007 Asia and South Pacific Design Automation Conference*, pages 336–341. IEEE, 2007.

[8] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. Dnnexplorer: A framework for modeling and exploring a novel paradigm of fpga-based dnn accelerator. *arXiv preprint arXiv:2008.12745*, 2020.

[9] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *FPGA 2017*, pages 65–74. ACM, 2017.

[10] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *TRETS 2018*, 11(3):16, 2018.

[11] Xilinx, Inc. PYNQ - Python productivity for Zynq. `http://www.pynq.io/`. Accessed: 2020-11-19.

[12] Hiroki Nakahara, Tomoya Fujii, and Shimpei Sato. A fully connected layer elimination for a binarizec convolutional neural network on an FPGA. In *FPL 2017*, pages 1–4. IEEE, 2017.

[13] Haruyoshi Yonekawa and Hiroki Nakahara. On-chip memory based binarized convolutional deep neural network applying batch normalization free technique on an FPGA. In *IPDPSW 2017*, pages 98–105. IEEE, 2017.

[14] Xilinx, Inc. SDSoC Development Environment. `https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html`. Accessed: 2020-11-19.

[15] Ken Sakamura. ITRON: An overview. In *TRON Project 1987 Open-Architecture Computer Systems*, pages 29–34. Springer, 1987.

[16] AUTOSAR Partners. AUTOSAR Classic Platform. `https://www.autosar.org/standards/classic-platform/`. Accessed 2021-01-31.

[17] Kazutoshi Wakabayashi. CyberWorkBench: integrated design environment based on c-based behavior synthesis and verification. In *In VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT)*, pages 173–176. IEEE, 2005.

[18] Xilinx, Inc. Vivado High-Level Synthesis. `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`. Accessed: 2020-9-26.

[19] Nagoya University Graduate School of Informatics. *SystemBuilder User Manual (in Japanese)*, 2020.

[20] Shinya Honda, Takayuki Wakabayashi, Hiroyuki Tomiyama, and Hiroaki Takada. RTOS-centric hardware/software cosimulator for embedded system design. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 158–163, 2004.

[21] Sony Network Communications Inc. *Neural Network Console Version 1.80 Instruction Manual*, 2020.

[22] ISO. ISO/IEC 9899:2018 - Information technology — Programming languages — C, 2018.

[23] Sony Network Communications Inc. Project - neural network console. `https://dl.sony.com/project/`. Accessed: 2020-11-19.

[24] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST Database. `http://yann.lecun.com/exdb/mnist/`. Accessed: 2020-11-18.

[25] ImageMagick Studio LLC. Imagemagick. `https://imagemagick.org/script/index.php`. Accessed 2021-02-12.

[26] Sony Network Communications Inc. Neural Network Libraries. `https://github.com/sony/nnabla`. Accessed: 2021-01-23.

[27] Onursal Çetin, Feyzullah Temurtaş, and Şenol Gülgönül. An application of multilayer neural network on hepatitis disease diagnosis using approximations of sigmoid activation function. *Dicle Medical Journal/Dicle Tip Dergisi*, 42(2), 2015.

Acknowledgment

I want to express my deep gratitude to Professor Edahiro and Professor Honda, for their guidance and critiques of this research work. I would also like to thank to Dr. Yamamoto for his useful advice and assistance in keeping my progress on schedule. I would also like to extend my thanks to the people at the Edahiro laboratory to offer me the resources in running the program.

Efficient Hardware Design of Quantized DNN Inference Using HLS and DNN Framework 2021年2月 KIM Hyunjae