A Deep Neural Network Translator for Edge Site Implementation

1st Mery Diana
Graduate School of Science and Technology
Kumamoto University
Kumamoto, Japan
merydiana1004@gmail.com

2nd Masato Kiyama
Faculty of Advanced Science and Technology
Kumamoto University
Kumamoto, Japan
masato@cs.kumamoto-u.ac.jp

3rd Motoki Amagasaki
Faculty of Advanced Science and Technology
Kumamoto University
Kumamoto, Japan
amagasaki@cs.kumamoto-u.ac.jp

4th Masayoshi Ito
Mitsubishi Electric Engineering
Tokyo, Japan
ito.masayoshi@ma.mee.co.jp

5th Yuki Morishita
Mitsubishi Electric Engineering
Tokyo, Japan
morishita.yuki@ma.mee.co.jp

Abstract—Implementing deep learning models (DNN) on edge sites remains challenging. Besides the high complexity of DNN models in their architecture, programming directly in hardware languages requires high-level verification to render these models run on edge devices. By utilizing High-Level Synthesis or HLS technology to generate HDL (Hardware Description Language) containing the DNN model, in this study, we built a DNN translator. DNN translator translates the model and its weights from the Pytorch-based DNN model to C++. The generated C++ files can be synthesized using HLS tools such as Vitis HLS or Vivado HLS. HDLs resulting from the synthesis of generated C++ are provided with HLS Pragmas to optimize the design. This DNN translator provides a solution to expand the application of the DNN model to edge devices, especially FPGA, as one of the potential edge devices. In the implementation, our DNN translator successfully generated C++ files for synthesizing the Multi-layer Perceptron (MLP) and LeNet5 and implemented them in an FPGA, Arty A7-100.

Index Terms—Edge Site, Deep Neural Network Translator, High-Level Synthesis

I. INTRODUCTION

Bringing Artificial Intelligence (AI) to the edge site offers many advantages and challenges. It decreases the bandwidth usage where many devices connect to the cloud [1], offering scalability and privacy for end-user safety [2]. Moreover, it benefits to reduce the computation load on the cloud side. Besides these positive prospects, the limitations remain challenging, such as power and resource limitations [3]. Therefore, the resource awareness of the design of the Deep Neural Network (DNN) model and implementation process, including the generating and implementation tools, is crucial. High-Level Synthesis, or HLS, as a design technology for FPGA [4], [5] offered several benefits compared to the Register Transfer Level or RTL. HLS utilizes high-level code such as C or C++ language frequently used by hardware designers. Since it applies C or C++ language that is easy to verify, it lowers design cost, thus reducing the production time, HLS offers more efficiency [6]–[8]. In this study, we develop a translator for the DNN model from the Pytorch [9] framework to C++ for enabling DNN implementation on the FPGA as a potential edge device via the HLS technique. After getting the generated C++, we conducted the HLS using Vitis HLS to perform the synthesis and implementation on the Arty A7-100. As a result, the generated C++ passed the synthesis, and we obtained the estimate and post-implementation of resources and timing usage.

II. DNN TRANSLATOR

We developed the DNN translator to translate the DNN model to C++ as one of the high-level codes to perform HLS to accelerate the implementation of the DNN model in the edge device. The translator utilized Pytorch and cgen [10] library to generate structured code from Python. The DNN translator flows as shown in Fig.1 and Algorithm 1. The DNN model built by the Pytorch framework will pass the transformation of its architecture, weights, and biases to one or zero using the transform_one_or_zero function. This function flow is described in the Algorithm 2. Then, the DNN translator requires the tensor inputs that are also generated and transformed to zero or one by one_or_zero function as informed in the Algorithm 3. After finishing the transformation of the DNN model and tensor input, the DNN translator will execute the gen_test_case function to generate C++ for the model architecture and its parameters, such as weight, layers, and biases. As presented in the Algorithm 1, before conducting the generation for C++, the translator examined the output dimension of the DNN model and called the gen_model_cpp function to iterate over the layer inside the DNN model architecture. We also utilized cgen to specify the pragma of HLS inside the layer module. As an example, the DNN translator uses cPragma("HLS pipeline") to generate #pragma HLS pipeline in generated C++.
Algorithm 1 Transforms and generates the DNN model

Require: model: a neural network model
Require: input_tensor: an input tensor
Require: output_dim: output dimension(s)
Ensure: Generated C++
1: model ← MLP()  
2: transform_one_or_zero(model)  
3: i ← one_or_zero(torch.randn(1, 10))  
4: gen_test_case(model, i, tuple(model(i).size()[1 : ]))

Algorithm 2 transform_one_or_zero

Require: model
Ensure: Transformed model
1: for all m in model.modules() do  
2: if m is an instance of nn.Linear or nn.Conv2d then  
3: m.weight.data ← one_or_zero(m.weight.data)  
4: if m.bias is None then  
5: m.bias.data ← one_or_zero(m.bias.data)  
6: end if
7: else
8: pass
9: end if
10: end for

Algorithm 3 one_or_zero

Require: t: a tensor
Ensure: A tensor with elements replaced by 0 or 1
1: function OneOrZero(t)
2: mask ← F.dropout(torch.ones_like(t), p = 0.5) == 0.0
3: result ← torch.where(mask, 0.0, 1.0)
4: return result
5: end function

A. HLS Pragmas

We utilized HLS Pragmas in the generated C++ file to optimize the HLS of DNN models. HLS Pragmas provides several types of pragmas as informed in [11]. In this study, we applied the pragma HLS pipeline (#pragma HLS pipeline) and pragma HLS allocation for multiplication operation (#pragma HLS allocation instances=mul limit=4) and add operation (#pragma HLS allocation instances=add limit=8). The pragma HLS pipeline allows the new process of inputs every clock cycle. This pragma will accelerate the synthesis process of the DNN model. Moreover, pragma HLS allocation will specify the resource allocation to the operation listed in instances = list, which in our study is multiplication and add function.

B. DNN Models

We implied DNN models to test and validate our DNN translator. The first is the MLP model, as shown in Fig. 2(a). We also applied the LeNet5 in Fig. 2(b) to enlarge our DNN translator application in the CNN-based model.

Algorithm 4 gen_test_case

Require: model: a neural network model
Require: input_tensor: an input tensor
Require: output_dim: output dimension(s)
Ensure: Generated C++
1: procedure GenTestCase(model, input_tensor, output_dim, num_type, header)
2: output_dim ← (output_dim,) if isinstance(output_dim, int) else output_dim
3: main_func ← c.FunctionBody(c.FunctionDeclaration(c.Value("int", "main"), [], c.Block([])))
4: output_dim ← tuple(input_tensor.size()[1 : ])
5: main_func.body.append(set_array(c.Value(num_type, gen_array("input", input_dim)), input_tensor[0]))
6: main_func.body.append(set_array(c.Value(num_type, gen_array("output", output_dim)), torch.zeros(output_dim))
7: with open(header, "w") as h:
8: top_funcs ← gen_model_cpp(model, input_dim, output_dim, num_type, header = h)
9: print(c.Include("<ap_int.h>"))
10: print(c.Include(header, system=False))
11: print(top.fdecl)
12: for f in funcs do
13: print(f)
14: end for
15: print(top)
16: main_func.body.append(c.Statement(" + f"top.fdecl.subdecl.name(input, output)" + "))
17: indexes ← gen_index_seq("i", len(output_dim))
18: output_var ← "output" + ",".join(map(lambda x: f"x{x}", indexes))
19: loop_body ← c.Statement(" + f"printf("%d, ", (int{output_var} + ")")
20: for i, s in zip(reversed(indexes), reversed(output_dim)) do
21: loop_body ← c.For("int i = 0", "i < s", "i + +", c.Block([loop_body]))
22: if i != indexes[-1] then
23: loop_body.body.append(c.Statement("printf(" printf( ")"))
24: end if
25: end for
26: main_func.body.append(loop_body)
27: print(main_func)
28: end procedure
III. EVALUATION

A. Evaluation Condition

We evaluated the DNN translator and its result to ensure its performance generating C++ files for HLS. Since the DNN Translator translates the Pytorch-based DNN model, we achieved the generated C++ for the DNN model architecture with the weights and biases. Then, we performed the synthesis and implementation on Vitis HLS by utilising Arty A7-100. We conducted the HLS sequentially by performing C-Simulation, C-Synthesis, and Co-Simulation. We also exported the synthesis result to RTL and ran the place and route to achieve resource usage and timing summary after implementation, besides the estimation from the synthesis in Vitis HLS. Fig. 3 showed the workflow of HLS until the implementation of generated C++ from DNN models.

B. Result

The documentation of the DNN translator and HLS tools result is informed in Fig 4. The DNN translator successfully generated the C++ from the DNN model. After finishing the synthesis and implementation, we obtained the HDL files, such as Verilog, estimation and implementation of resource usage for MLP and LeNet5 in the Arty A7-100.

The estimation of resource usage for the MLP and the LeNet5 are presented in Table I and Table II. We also achieved the summary for timing estimation. The estimated timing for both models is lower than the timing target, below 10 ns. As shown in Table III, the MLP model passed the required timing by achieving 6.823 ns for the estimated timing and LeNet5 estimated 7.248 ns.

Fig. 3: The operations in Vitis HLS from generated C++

<table>
<thead>
<tr>
<th>Name</th>
<th>BRAM_18K</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
<th>URAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>99</td>
<td>0</td>
<td>3,504</td>
<td>6,372</td>
<td>0</td>
</tr>
<tr>
<td>Available</td>
<td>270</td>
<td>240</td>
<td>126,800</td>
<td>63,400</td>
<td>0</td>
</tr>
<tr>
<td>Utilization (%)</td>
<td>36</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Table I: Estimation of resource usage for MLP

<table>
<thead>
<tr>
<th>Name</th>
<th>BRAM_18K</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
<th>URAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>6</td>
<td>0</td>
<td>13,560</td>
<td>32,794</td>
<td>0</td>
</tr>
<tr>
<td>Available</td>
<td>270</td>
<td>240</td>
<td>126,800</td>
<td>63,400</td>
<td>0</td>
</tr>
<tr>
<td>Utilization (%)</td>
<td>2</td>
<td>0</td>
<td>10</td>
<td>51</td>
<td>0</td>
</tr>
</tbody>
</table>

Table II: Estimation of resource usage for LeNet5

IV. DISCUSSION

After performing the experiments on the DNN translator with the MLP and LeNet5 as the DNN models, we completed
Since the DNN translator constructed the pragma HLS Pipeline, the generated C++ performed these pipelines during synthesis. As shown in Table V, the MLP model utilized the pipelines in each linear layer operation. Compared to MLP, LeNet5 applied more pipelines during the synthesis. LeNet5 used the pipelines in 2D convolutional layers, reshape operations, and linear layers, as shown in Table VI. Based on this current result, generated C++ as the result of the DNN translator performed well in the HLS tool. The implementation also showed acceptable resource usage and timing summary.

### TABLE V: Pragma HLS Pipeline for MLP

<table>
<thead>
<tr>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipeline</td>
<td>hls-mlpa/mdlpa.cpp:10 in linear_10x100</td>
</tr>
<tr>
<td>pipeline</td>
<td>hls-mlpa/mdlpa.cpp:34 in linear_100x84</td>
</tr>
<tr>
<td>pipeline</td>
<td>hls-mlpa/mdlpa.cpp:58 in linear_84x10</td>
</tr>
</tbody>
</table>

### V. CONCLUSION AND FUTURE WORK

We proposed a DNN translator using the Pytorch framework in the present study. The generated C++ from the translator could easily be synthesized in HLS tools such as Vitis HLS or Vivado HLS. The DNN translator also provided the HLS Pragma that enhanced the synthesis efficiency. As a result, generated C++ from the DNN models in the Pytorch framework could be implemented in the edge device as an FPGA. By synthesizing using the HLS, the DNN models can rapidly expand their implementation in the edge site. In future work, we would develop the application of the DNN translator into another type of DNN models and measure the inference process of our synthesized DNN model in FPGA, then evaluate the results and behaviour.

### REFERENCES


