International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 14, Number 2, pages 145–156, July 2024

Advanced Implementation of DNN Translator using ResNet9 for Edge Devices

Mery Diana
Graduate School of Science and Technology, Kumamoto University
Kumamoto, 860-0862, Japan
merydiana1004@gmail.com

Masato Kiyama and Motoki Amagasaki
Faculty of Advance Science and Technology, Kumamoto University
Kumamoto, 860-0862, Japan
{masato, amagasaki} @cs.kumamoto-u.ac.jp

Masayoshi Ito and Yuki Morishita
Mitsubishi Electric Engineering
Tokyo, 102-0073, Japan
{ito, morishita}@ma.mee.co.jp

Received: February 15, 2024
Revised: May 5, 2024
Accepted: May 28, 2024
Communicated by Shoichi Hirasawa

Abstract

Resource limitations remain challenging in designing and implementing Deep Neural Networks (DNNs) on edge devices. The high complexity of DNN architectures and the development of these models using hardware languages require high-level verification to ensure they run on specific edge devices such as FPGA (Field Programmable Gate Array). To address these issues, the DNN translator was developed and performed well in the basic models such as MLP (Multi-layer Perceptron) and LeNet5. The DNN translator generates the DNN models and their parameters for performing the High-Level Synthesis or HLS technology in C++. In this study, we applied ResNet as a DNN model with more complex architecture from the CNNs (Convolutional Neural Networks) family. As a result, the generated C++ files for the ResNet9 and its weights successfully underwent synthesis and implementation on FPGA (Arty A7-100) using Vitis HLS.

Keywords: Edge Site, Deep Neural Network Translator, High-Level Synthesis, ResNet9

1 Introduction

Introducing Artificial Intelligence (AI) to edge sites presents numerous advantages and challenges. It leads to a reduction in bandwidth usage, particularly in scenarios where numerous devices are connected to the cloud [3]. This approach offers scalability and enhances privacy to ensure end-user safety, as highlighted by Xu et al. [18]. Additionally, it contributes to the alleviation of computation

[0]This paper is an extension of presented paper in the CANDAR Workshop 2023

load on the cloud side. Despite these promising prospects, there are formidable limitations, notably in terms of power and resource constraints [12]. Hence, maintaining a keen awareness of resources in the design of Deep Neural Network (DNN) models and their implementation processes is imperative. High-Level Synthesis (HLS), a design technology for Field-Programmable Gate Arrays (FPGA) [16, 20], offers several advantages over Register Transfer Level (RTL). HLS employs high-level code, such as C or C++, commonly used by hardware designers. More, utilizing easily verifiable C or C++ languages reduces design costs, thereby diminishing production time and enhancing efficiency [2, 10, 13].

In the previous study [5], we have developed a translator to convert DNN models from the PyTorch [15] framework into C++. This translation facilitates DNN implementation on an FPGA as a potential edge device using the HLS technique. Subsequently, the generated C++ code underwent HLS using Vitis HLS, conducting synthesis and implementation on the Arty A7-100. In this extended study, we performed the translator in ResNet9 from a CNN-based family as the deeper model compared to MLP and LeNet5. ResNet9 was constructed by utilizing several convolutional layers for residual layers [7] with more complex architecture than MLP and LeNet5. After getting the C++ files from the DNN translator, we performed the C synthesis and implementation using Vitis HLS. The synthesized C++ code successfully passed the synthesis phase, and we obtained estimates for resource utilization and timing during post-implementation. The rest of this document is as follows. Section II is an overview of related work, while Section III outlines the architecture of the DNN translator and DNN model. In Section IV, we introduce the evaluation conditions, present the results, and engage in a discussion. Finally, Section V summarizes our present research and outlines directions for future work.

2 Related Work

In order to enlarge the implementation of DNN in the FPGA, several researches have been conducted. [19] developed Caffeine as the accelerator for convolutional layers and fully connected layers by using the FPGA. This Caffeine, integrated with the Caffe framework, boosts its performance to achieve better energy efficiency over the GPU implementation. Besides this merit, the implementation of Caffeine is limited to the convolutional layer and fully connected layer of the DNN model. [6] also proposed the CNN2Gate that parses the CNN model from several machine learning and performs the OpenCL synthesis tools to run the project in the FPGA. Compared to the HDL, since it applied the OpenCL, CNN2Gate may lack control for optimization at the low level, limitation in the syntax, and does not support dynamic memory handling [14]. Therefore, a DNN translator is required to parse the DNN architecture and its parameters to be implemented easily and user-friendly.

HLS is one of the techniques in hardware designing that is easier by utilizing C, C++, or System C [1]. Compared to the RTL technique, HLS can enhance efficiency due to its simplicity in verification, thus reducing the design cost. HLS, or high-level synthesis, offers advantages in designing the hardware compared to RTL technique. Pytorch as the framework is commonly usage in the design of the DNN model [11]. By using the Python ecosystem, Pytorch offers the imperative and Pythonic programming style that supports hardware accelerators such as GPU [15]. These conditions enhance the challenges to develop the DNN translator from DNN model that using Pytorch as the framework. By taking the pros of the Pytorch as the framework for the DNN model and HLS technique for designing the hardware for implementing them, a DNN translator as mentioned previouly was developed to generates the C++ for the DNN models and their parameters. In this paper, we explained the extention of implementation of the translator in the ResNet9 that built using pytorch framework. ResNet9 as one of the CNN-based model applied the ResNet architecture with the smaller parameters that makes ResNet9 as a potential DNN model for edge device. We also presented and analized the HLS results of Resnet9 in FPGA Arty7-100.

3 DNN Translator

We have developed a DNN translator designed to convert DNN models into C++ [5], serving as one of the high-level codes for High-Level Synthesis (HLS). This translation process aims to expedite the implementation of DNN models on edge devices. Our translator employs PyTorch and the cgen library [8] to produce structured code from Python. The workflow of the DNN translator is illustrated in Figure 1, with the corresponding algorithm detailed in Algorithm 1. The DNN model, constructed using the PyTorch framework, transforms its architecture, weights, and biases, employing the *transform_one_or_zero* function. The steps involved in this transformation are outlined in Algorithm 2. Subsequently, the translator necessitates tensor inputs, also subject to transformation through the *one_or_zero* function, as specified in Algorithm 3. The input size can be adjusted to match the DNN model's input size. For example, the tensor input applied to the MLP model is 1×10.

Upon completing the transformation of the DNN model and tensor inputs, the DNN translator executes the *gen_test_case* function to generate C++ code for the model's architecture and its parameters, including weight, layers, and biases. As indicated in Algorithm 1, before C++ generation, the translator examines the output dimensions of the DNN model and calls the *gen_model_cpp* function to iterate through the layers within the DNN model architecture. The DNN translator supports CNN-based model architecture through convolutional layers, fully connected layers, and flattening. DNN translators also translate ReLU (Rectified Linear Unit) as the activation function and Max-Pooling as the pooling layer. Additionally, cgen is utilized to define HLS pragmas within the layer module and *ap_int.h* from *HLS_arbitary_Precision_Types* [17] to define the generated code in integer (INT8). The translator optimizes the HLS of DNN models by providing the pragma HLS pipeline and pragma HLS allocation in generating a C++ file for the model architecture. For instance, the DNN translator employs `c.Pragma("HLS pipeline")` to generate `#pragma HLS pipeline` in the resulting C++ code. We inserted these HLS Pragmas directly into the convolutional layer, flatten layer, and linear layer by using cgen library (`c.Pragma`).

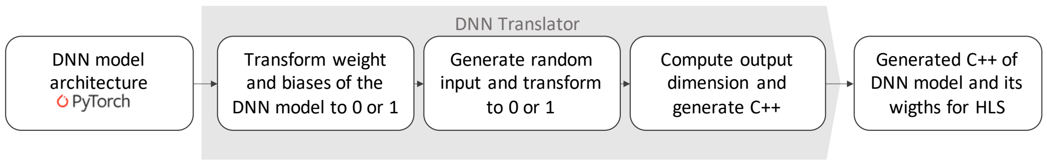

DNN Translator
DNN model architecture
PyTorch
Transform weight and biases of the DNN model to 0 or 1
Generate random input and transform to 0 or 1
Compute output dimension and generate C++
Generated C++ of DNN model and its wigths for HLS

Figure 1: Workflow of DNN Translator.

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: $model \leftarrow MLP()$
2: transform_one_or_zero($model$)
3: $i \leftarrow$ one_or_zero(torch.randn(1, 10))
4: gen_test_case($model$, $i$, tuple($model(i).size()[1:]$))
```

3.1 DNN Model

We employed basic models (MLP and LeNet5) in the previous study to assess and validate the DNN translator. Then, we enhanced the DNN translator implementation into the deeper DNN model, such as ResNet9. The initial model is the MLP and LeNet5, depicted in Figure 2(a) and Figure 2(b). We utilized the ResNet9 architecture as illustrated in Figure 2(c) with residual blocks to broaden

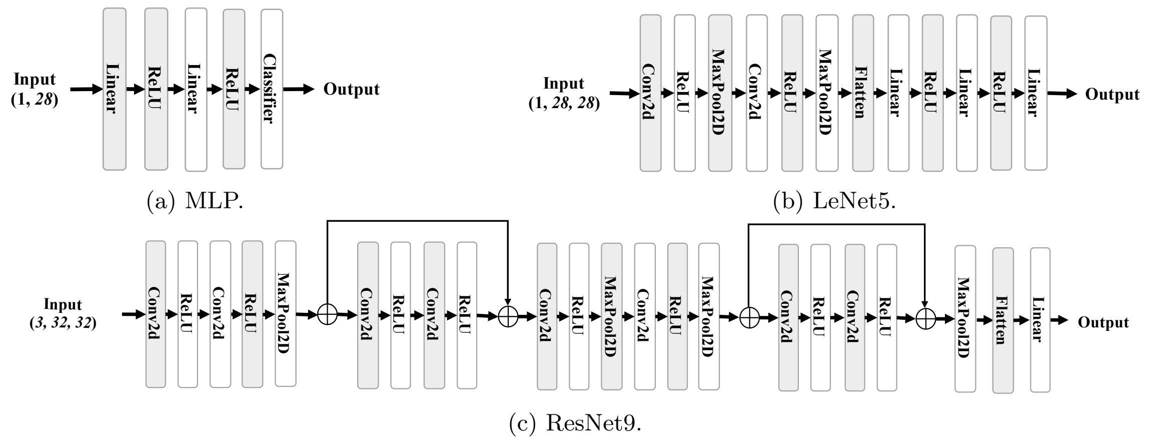
Algorithm 2 $transform\_one\_or\_zero$

```
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:         $m.weight.data \leftarrow$ one_or_zero($m.weight.data$)
 4:         if $m.bias \neq$ None then
 5:             $m.bias.data \leftarrow$ 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: $mask \leftarrow$ F.dropout(torch.ones_like($t$), $p = 0.5$) == 0.0
3: $result \leftarrow$ torch.where($mask, 0.0, 1.0$)
4: return $result$
5: end function
```

the scope of our DNN translator's applicability, particularly in the context of CNN-based models. Every residual block consists of two convolutional layers and ReLU layers. MNIST [4] dataset with size 28×28 is fed into the MLP model and LeNet5 model in model preparation, such as the training process. In the ResNet9 model, we applied CIFAR10 [9] as the input ($3 \times 32 \times 32$). The image input can be adjusted as implementation and the model architecture.



(a) MLP. (b) LeNet5. (c) ResNet9.

Figure 2: (a)Architecture of MLP, (b)LeNet5, and (c)ResNet9.

4 Evaluation

4.1 Evaluation Condition

We comprehensively evaluated the DNN translator and its output to ensure optimal performance in generating C++ files of ResNet9 for High-Level Synthesis (HLS). By employing the DNN translator,

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: input_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]”, 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 " + f”i = 0”, "i < s", "i + +", c.Block([loop_body]))
22:   if i ≠ indexes[−1] then
23:     loop_body.body.append(c.Statement("printf(
      n)"))
24:   end if
25: end for
26: main_func.body.append(loop_body)
27: print(main_func)
28: end procedure
```

which translates PyTorch-based DNN models, we successfully obtained the generated C++ code encapsulating Resnet9's architecture, weights, and biases. Subsequently, we proceeded with the synthesis and implementation stages of Vitis HLS, utilizing the Arty A7-100 platform with the environment as described in Table 1.

Table 1: Evaluation Environment.

Hardware/ Software	Name	Specification
Hardware	Arty A7-100	LUTs : 15,850
		Block RAM : 4,860 Kbits
		Clock Management Tiles : 6
		DSP slices : 240
		Internal clock speeds exceeding 450MHz
		256MB DDR3L with a 16-bit bus @ 333MHz
		Internal clock : 450 MHz+
		16MB Quad-SPI Flash
		Powered : USB or 7V-15V
Software	Pytorch library	torch : 2.0.1
	Cgen Library	2020.1
	HLS Tool	Vitis HLS 2021.2

The HLS process was executed sequentially, encompassing C-Synthesis and Co-Simulation. Additionally, we exported the synthesis results to the Register Transfer Level (RTL) and conducted place and route operations to ascertain resource utilization and timing summaries post-implementation. These steps complemented the estimations derived from the synthesis in Vitis HLS 2021.2. The workflow of the HLS process, leading up to the implementation of the generated C++ code from DNN models is depicted in Figure 3.

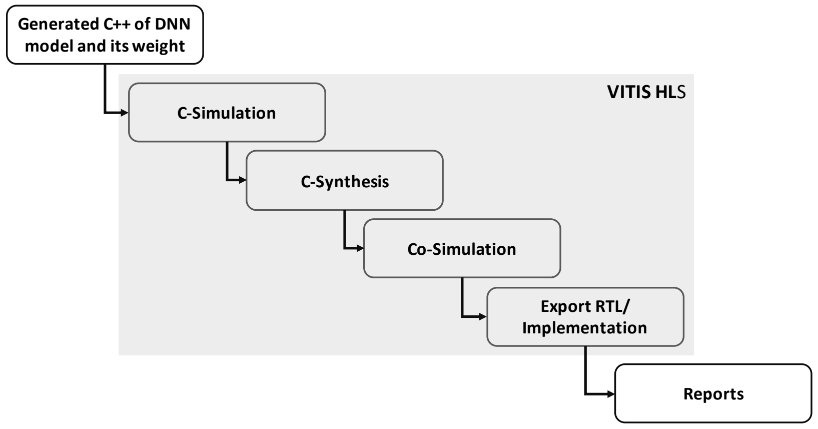

Generated C++ of DNN model and its weight
VITIS HLS
C-Simulation
C-Synthesis
Co-Simulation
Export RTL/ Implementation
Reports

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

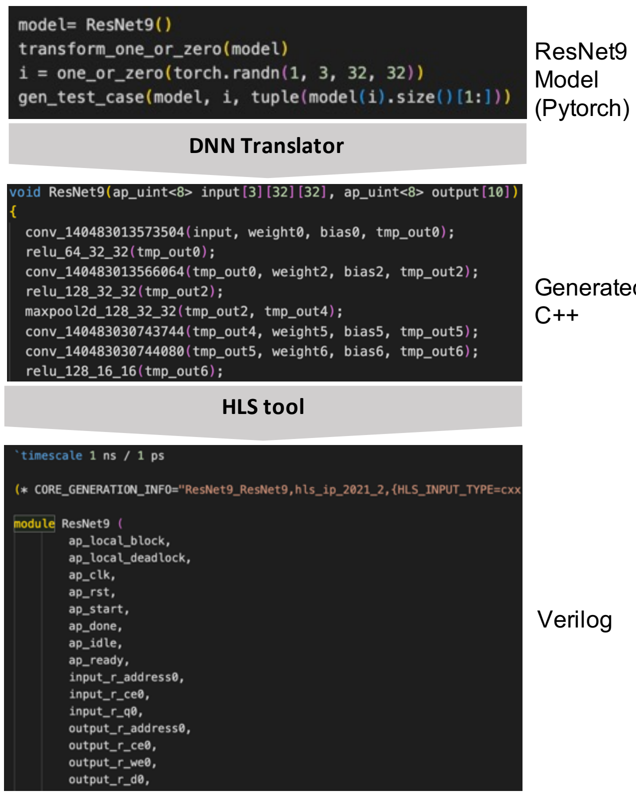
4.2 Result

Since the DNN translator generates the models in INT8, we performed experiments to define their accuracy in similar data types before generating the C++ using the DNN translator. We also calculated the accuracy of the baseline model in floating point (FP32) as the comparison. As shown in Table 2, MLP model accuracy is 94.44% in FP32 and 94.14% in INT8. Thus, for ResNet9, model accuracy decreased from 87.29% to 87.13%. In contrast to both models, LeNet5 accuracy slightly increased from 98.31% in FP32 and 98.33% in INT8. Based on the results, the model accuracy in INT8 is nearly equivalent to the baseline model in FP32.

Table 2: Accuray of DNN models.

No	Model	Dataset	Accuracy(%)	
			FP32	INT8
1	MLP	MNIST	94.44	94.14
2	LeNet5	MNIST	98.31	98.33
3	ResNet9	CIFAR10	87.29	87.13

The translator successfully generated the C++ files from the DNN models. Documentation detailing the outcomes of the DNN translator and HLS tools for ResNet9 is provided in Figure 4. Subsequent to the synthesis and implementation phases, we acquired Hardware Description Language (HDL) files, including Verilog. The assessment encompassed estimations and implementations of resource usage for DNN models on the Arty A7-100 platform.



```
model= ResNet9()
transform_one_or_zero(model)
i = one_or_zero(torch.randn(1, 3, 32, 32))
gen_test_case(model, i, tuple(model(i).size()[1:]))
```

```
void ResNet9(ap_uint<8> input[3][32][32], ap_uint<8> output[10])
{
  conv_140483013573504(input, weight0, bias0, tmp_out0);
  relu_64_32_32(tmp_out0);
  conv_140483013566064(tmp_out0, weight2, bias2, tmp_out2);
  relu_128_32_32(tmp_out2);
  maxpool2d_128_32_32(tmp_out2, tmp_out4);
  conv_140483030743744(tmp_out4, weight5, bias5, tmp_out5);
  conv_140483030744080(tmp_out5, weight6, bias6, tmp_out6);
  relu_128_16_16(tmp_out6);
```

```
`timescale 1 ns / 1 ps

(* CORE_GENERATION_INFO="ResNet9_ResNet9,hls_ip_2021_2,{HLS_INPUT_TYPE=cxx

module ResNet9 (
        ap_local_block,
        ap_local_deadlock,
        ap_clk,
        ap_rst,
        ap_start,
        ap_done,
        ap_idle,
        ap_ready,
        input_r_address0,
        input_r_ce0,
        input_r_q0,
        output_r_address0,
        output_r_ce0,
        output_r_we0,
        output_r_d0,
```

Figure 4: The documentation of generated files from DNN Translator and HLS tool.

The resource usage estimates for MLP, LeNet5, and ResNet9 are outlined in Table 3, Table 4, and Table 5, respectively. We also obtained the summaries of the timing estimation. The estimated timing for the models falls below the specified target, specifically below 10 ns. Table 6 illustrates that the MLP model successfully met the required timing, achieving an estimated timing of 6.823 ns, while LeNet5 and ResNet9 reached an estimated timing of 7.248 ns.

Table 3: Estimation of resource usage for MLP.

Name	BRAM_18K	DSP	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	0	-	14,454	28,809
Memory	0	-	32	24
Multiplexer	-	-	-	151
Register	-	-	97	-
Total	**0**	**0**	**14,495**	**28,984**
Available	**270**	**240**	**126,800**	**63,400**
Utilization (%)		**0**	**11**	**45**

Table 4: Estimation of resource usage for LeNet5.

Name	BRAM_18K	DSP	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	0	-	16,896	30,417
Memory	6	-	832	126
Multiplexer	-	-	-	2,335
Register	-	-	24	-
Total	**6**	**0**	**17,752**	**32,878**
Available	**270**	**240**	**126,800**	**63,400**
Utilization (%)	**2**	**0**	**14**	**51**

Table 5: Estimation of resource usage for ResNet9.

Name	BRAM_18K	DSP	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	235
FIFO	-	-	-	-
Instance	45	-	12,904	30,333
Memory	310	-	632	126
Multiplexer	-	-	-	2,335
Register	-	-	24	-
Total	**355**	**0**	**35,08**	**55,046**
Available	**270**	**240**	**126,800**	**63,400**
Utilization (%)	**131**	**0**	**27**	**86**

Table 6: Timing summary of MLP, LeNet5, and ResNet9.

Model	Clock	Target	Estimated	Uncertainty
MLP	ap_clk	10 ns	6.823 ns	2.7 ns
LeNet5	ap_clk	10 ns	7.248 ns	2.7 ns
ResNet9	ap_clk	10 ns	7.248 ns	2.7 ns

5 Discussion

After successfully running the translation of the DNN model, we received the model architecture and its weight in the C++ files. DNN translator translated models into INT8 and generated the pipelines for synthesis as shown in Figure 4. The generated C++ files from the DNN translator provide the layers that built the DNN model as the baseline model developed in the Pytorch framework into C++. For example, the DNN translator entirely generated two residual blocks of ResNet9 that are similar to the original model's architecture. Then, we conducted HLS using the generated C++ files to evaluate their performance and gain the HDL files for each model. The result shows that these generated files were well synthesized in the Vitis HLS. We received information related to performance estimation, such as a summary of the timing and utilization of resource usage for implementing the DNN models.

We also performed the place and route to analyze the implementation resource usage and final timing on the Arty A7-100 as a candidate for edge device. The results in Table 7 facilitate comparing the implemented resource usage and the earlier estimated resource usage for each model. Table 5 showed that ResNet9 utilized all the memory resources and exceeded the resource availability in the Arty7-100 as the estimation. For the MLP model, the implementation successfully passed the Critical Path Delay (CP), with a consumption of 5.615 ns post-synthesis and 8.853 ns post-implementation. Similarly, LeNet5 met the final timing requirements for the Critical Path, achieving 6.748 ns post-synthesis and 9.598 ns post-implementation. Even though ResNet consumed 10.121 ns over the CP required timing in the post-synthesis, it passed the post-implementation timing with 9.815 ns. These conditions show that a deeper model loads more complexity in operation and longer timing, which consumes more resource availability in the hardware. However, in the implementation, the DNN models passed the CP timing required, which is less than 10 ns, as informed in Table 7. ResNet accomplished adeptly in the Arty7-100 since the post-implementation showed acceptable resource usage and timing summary. The result of synthesis and implementation confirmed that the generated C++ files of the DNN translator performed well in the HLS tool.

Table 7: Implementation of resource usage for MLP, LeNet5, and ResNet9.

Name	**MLP**	**LeNet5**	**ResNet9**
SLICE	1,026	4,076	13,920
LUT	2,100	11,232	43,448
FF	2,967	12,299	20,899
DSP	0	0	0
BRAM	2	8	270
LATCH	0	0	0
SRL	17	24	13
CLB	0	0	0
Final Timing			
CP required	10 ns	10 ns	10 ns
CP achieved post-synthesis	5.615 ns	6.748 ns	10.121 ns
CP achieved post-implementation	8.853 ns	9.598 ns	9.815 ns
Timing	met	met	met

Since the DNN translator constructed the pragma HLS Pipeline to optimize the synthesis of DNN models, the generated C++ performed well in the pipelines during synthesis. Compared to MLP and LeNet5, ResNet9 applied more pipelines during the synthesis. As shown in Table 8, Table 9, and Table 10, every model utilized the pipelines in each linear layer operation. However, ResNet9 used the pipelines in 2D convolutional layers, reshape operations, and linear layers over the MLP and LeNet5, as shown in Table 10. The number of pipeline operations differs for each DNN model. It depends on the stack of the DNN model's layers. The deeper models, such as ResNet9, operated more pipelines than the MLP and LeNet5 models. ResNet applied two convolutional layers for each block, increasing the number of synthesis pipelines. Thus, this result indicates that the DNN translator generated the HLS Pragmas as assigned and executed in HLS.

Table 8: Pragma HLS Pipeline for MLP.

Type	**Location**
pipeline	hls-mlpa7/mlpv1.cpp:10 in linear_10x100
pipeline	hls-mlpa7/mlpv1.cpp:34 in linear_100x84
pipeline	hls-mlpa7/mlpv1.cpp:58 in linear_84x10

Table 9: Pragma HLS Pipeline for LeNet5.

Type	**Location**
pipeline	hls-lenet5/lenet5.cpp:9 in conv_140204914556112
pipeline	hls-lenet5/lenet5.cpp:12 in conv_140204914556112
pipeline	hls-lenet5/lenet5.cpp:86 in conv_140204914552944
pipeline	hls-lenet5/lenet5.cpp:89 in conv_140204914552944
pipeline	hls-lenet5/lenet5.cpp:158 in reshape_16_5_5
pipeline	hls-lenet5/lenet5.cpp:161 in reshape_16_5_5
pipeline	hls-lenet5/lenet5.cpp:164 in reshape_16_5_5
pipeline	hls-lenet5/lenet5.cpp:176 in linear_400x120
pipeline	hls-lenet5/lenet5.cpp:200 in linear_120x84
pipeline	hls-lenet5/lenet5.cpp:224 in linear_84x10

6 Conclusion and Future Work

In this research, we introduced the advanced implementation of a DNN translator leveraging the PyTorch framework in building ResNet9 architecture. The C++ files of ResNet9 generated by the translator can be seamlessly synthesized using HLS tools like Vitis HLS. Notably, the DNN translator also incorporates HLS Pragmas, thereby improving the synthesis efficiency of DNN models. Consequently, the generated C++ files from DNN models using the PyTorch framework can be effectively implemented on edge devices such as Arty7-100.

Through synthesis using HLS, the deployment of DNN models at the edge site can be significantly accelerated. As a future study, our future endeavors will involve extending the application of the DNN translator to various DNN models, not only from the CNN-based model. We will enhance the translator's performance in handling the DNN model with complex architecture. Therefore, the HLS tools can effectively synthesize the files generated by the DNN translator.

References

[1] Donald G. Bailey. The advantages and limitations of high level synthesis for fpga based image processing. In *Proceedings of the 9th International Conference on Distributed Smart Cameras*, ICDSC '15, page 134–139, 2015.

[2] Yosi Ben-Asher and Nadav Rotem. The benefits of using variable-length pipelined operations in high-level synthesis. *ACM Trans. Embed. Comput. Syst.*, 13(3), dec 2013.

[3] Olivier Debauche, Saïd Mahmoudi, Sidi Ahmed Mahmoudi, Pierre Manneback, and Frédéric Lebeau. A new edge architecture for ai-iot services deployment. *Procedia Computer Science*, 175:10–19, 2020.

[4] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

Table 10: Pragma HLS Pipeline for ResNet9.

Type	Location
pipeline	hls-resnet92/testresnet92.cpp:9 in conv_140483013573504
pipeline	hls-resnet92/testresnet92.cpp:12 in conv_140483013573504
pipeline	hls-resnet92/testresnet92.cpp:68 in conv_140483013566064
pipeline	hls-resnet92/testresnet92.cpp:71 in conv_140483013566064
pipeline	hls-resnet92/testresnet92.cpp:145 in conv_140483030743744
pipeline	hls-resnet92/testresnet92.cpp:148 in conv_140483030743744
pipeline	hls-resnet92/testresnet92.cpp:191 in conv_140483030744080
pipeline	hls-resnet92/testresnet92.cpp:194 in conv_140483030744080
pipeline	hls-resnet92/testresnet92.cpp:250 in conv_140483030744032
pipeline	hls-resnet92/testresnet92.cpp:253 in conv_140483030744032
pipeline	hls-resnet92/testresnet92.cpp:322 in conv_140483030744608
pipeline	hls-resnet92/testresnet92.cpp:325 in conv_140483030744608
pipeline	hls-resnet92/testresnet92.cpp:394 in conv_140483030744464
pipeline	hls-resnet92/testresnet92.cpp:397 in conv_140483030744464
pipeline	hls-resnet92/testresnet92.cpp:440 in conv_140483030744704
pipeline	hls-resnet92/testresnet92.cpp:443 in conv_140483030744704
pipeline	hls-resnet92/testresnet92.cpp:512 in reshape_256_2_2
pipeline	hls-resnet92/testresnet92.cpp:515 in reshape_256_2_2
pipeline	hls-resnet92/testresnet92.cpp:518 in reshape_256_2_2
pipeline	hls-resnet92/testresnet92.cpp:530 in linear_1024x10

[5] Mery Diana, Masato Kiyama, Motoki Amagasaki, Masayoshi Ito, and Yuki Morishita. Deep neural network translator for edge site implementation. In *Proceeding of 2023 Eleventh International Symposium on Computing and Networking Workshops (CANDARW)*, page 52, Matsue, Japan, nov 2023.

[6] Alireza Ghaffari and Yvon Savaria. Cnn2gate: An implementation of convolutional neural networks inference on fpgas with automated design space exploration. *Electronics*, 9(12), 2020.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[8] Andreas Kloeckner and Contributors. cgen - code generation library. `https://github.com/inducer/cgen`, 2020. accessed May 12.

[9] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

[10] Marcos T. Leipnitz and Gabriel L. Nazar. High-level synthesis of approximate designs under real-time constraints. *ACM Trans. Embed. Comput. Syst.*, 18(5s), oct 2019.

[11] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari. Pytorchfi: A runtime perturbation tool for dnns. In *Proceeding of 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 25–31, 2020.

[12] Javier Mendez, Kay Bierzynski, M. P. Cuéllar, and Diego P. Morales. Edge intelligence: Concepts, architectures, applications, and future directions. *ACM Trans. Embed. Comput. Syst.*, 21(5), oct 2022.

[13] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen

Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[14] Katharina Ostaszewski, Philip Heinisch, and Hendrik Ranocha. Advantages and pitfalls of opencl in computational physics. In *Proceedings of the International Workshop on OpenCL*, IWOCL '18, New York, NY, USA, 2018.

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. https://pytorch.org, 2021.

[16] Zi Wang and Benjamin Carrion Schafer. Learning from the past: Efficient high-level synthesis design space exploration for fpgas. *ACM Trans. Des. Autom. Electron. Syst.*, 27(4), 2022.

[17] Xilinx, Inc. Xilinx vitis high-level synthesis (hls) documentation, 2022. accessed May 14.

[18] Dianlei Xu, Tong Li, Yong Li, Xiang Su, Sasu Tarkoma, Tao Jiang, Jon Crowcroft, and Pan Hui. Edge Intelligence: Architectures, Challenges, and Applications, June 2020. arXiv:2003.12172 [cs].

[19] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(11):2072–2085, 2019.

[20] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. Comba: A comprehensive model-based analysis framework for high level synthesis of real applications. In *Proceeding of 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 430–437, 2017.