Code Conversion Method on Process-in-Memory Platform

Jinhui Liu, Chen Zhao*, Fangzhou Du and Xuwen Liu

Department of Computer Science and Technology, Xidian University, Xi'an, Shaanxi 710126, China

*Corresponding author email: chzhao0922@126.com

Abstract. Process-in-Memory (PIM) has become a favorable solution for data intensive applications with its advantages of placing processing units in or near memory, reducing the movement of data between memory and processing unit. However, most recent PIM platforms have relatively large limitations in terms of computing flexibility and types of computing, and programming is complicated, which cannot provide a unified programming framework. Therefore, it becomes a challenge to deploy the code to the hardware, which severely restricts the application development on the PIM platform. This paper presents a method of code conversion based on the PIM platform, which allows users to develop applications on the PIM platform with the high-level programming language, makes the PIM platform more accessible in the development of data intensive applications.

Keywords: PIM, code conversion method, NNs, Algorithm library.

1. Introduction

The traditional computer system uses a von Neumann architecture where the processor and memory are separated, with the processor as the center, and the memory is connected to the processor through a high-speed data bus. However, the data movement between the processor and the memory in the conventional computer system leads to serious performance degradation and significant power consumption, which is well known as the "Memory Wall" problem [1]. Process-in-Memory (PIM) is a very promising solution to this problem by implementing logical operations in Memory, providing high bandwidth, massive parallelism, and high energy efficiency, eliminating the overhead of moving data between CPU and Memory[2]. Neural networks(NNs) have been demonstrated strong capabilities in handling emerging applications such as language recognition, speech recognition, image classification, video segmentation, and games. In particular, deep neural networks(DNNs) have shown superior effectiveness in various image processing, classification problems, video processing, speech recognition, computer vision, and healthcare [3, 4, 5].

However, neural network applications are mostly developed in high-level programming language [6], such as Python, C++, JAVA, etc. Meanwhile, recent PIM architectures mostly accept hardware description languages (HDLs) as their only input[7]. State-of-the-art high-level synthesis tools are mostly designed for certain architectures, such as Vivado HLS developed by Xilinx [8], which cannot provide support for the PIM platform. Therefore, when deploying a neural network application to a PIM platform, developers need to understand the hardware architecture of the system and the application should be developed with HDLs, which have a longer period and higher difficulty in development.

In this paper, we present a general method to convert the high-level language to HDL for the PIM platform, enabling users to develop with the high-level language directly without understanding the hardware architecture, which shortens the development period and makes the PIM platform more accessible. The simulation results prove the validity and feasibility of the proposed method.
2. Method
In this section, we introduce the code conversion architecture. The code conversion architecture of deep learning models based on the PIM platform is shown in Figure 1. The architecture provides complete tool chains of deploying deep learning models to the PIM architecture. Users only need to focus on coding with the high-level programming language, which implements the training and optimization of the deep learning model, without paying attention to hardware details. Through the code conversion architecture, the efficient deployment of deep learning models on the memory architecture can be accomplished.

The detailed process of code conversion is shown in Figure 2. Firstly, the node information, which includes the node name, the operation of the node, the data type and the interconnection relationship between the nodes, is extracted from the calculation graph. Secondly, the proposed method checks whether the operation of the node is involved in the algorithm library, which is coded in the hardware description language. The algorithm library contains basic modules of common operations and the data types of the algorithms are changeable. It makes the method has high versatility. If the operation of the node is included in the algorithm library, the proposed method extracts the operation algorithm, and modifies its data type to ensure that the data type is consistent with the extracted node data type. However, if the operation of the node cannot be found in the algorithm library, the conversion will be aborted. Finally, the top-level file is created according to the interconnection relationship between the nodes, and the code conversion from the high-level language to the bottom-level hardware description language is realized.

**Figure 1.** Code conversion architecture.
2.1. Generation and Extraction of Computational Graph

Front-end deep learning frameworks are usually based on existing common frameworks, such as Tensorflow, MxNet, ONNX, etc. They all use the computational graph to represent mathematical calculations, which include nodes and edges. The nodes in the graph generally represent one of the followings: (1) the mathematical operations; (2) the beginning/end nodes of input/output data; (3) the endpoint of the reading/writing persistent variables. The edges in the graph represent the input-output relationships between nodes.

However, different frameworks have different descriptions of the computational graph, and if a conversion method is individually developed for each framework, the workload is heavy, and the code maintenance is difficult. Therefore, we use the cross-framework Open Neural Network Exchange (ONNX) developed by Microsoft to realize the unification of the computational graph. TensorFlow is adopted as the preliminary work of code conversion and other neural network models are converted into the TensorFlow PB model [9] through ONNX in this paper.

NNs are trained and deployed in two stages. In the stage of training, after the training samples are imported into the neural network model, Intermediate results can be obtained by forward propagation algorithm. Then, the Error Back Propagation (BP) algorithm calculates the loss function according to the input value and output value obtained by the neural network. Finally, NNs solve loss function and update the parameters. This process is generally implemented on the CPU or GPU. In the deployment stage, the deployment and implementation of the neural network model are mainly accomplished on the actual computing devices, such as FPGA, PIM, CPU, GPU, etc.

The code conversion framework proposed in this paper focuses on the deployment of the Neural Network. Firstly, users build models through the Deep Learning framework, and generate model files, then convert them into a unified computational graph format through ONNX. It contains the node information of the network and the interconnection relationship between the nodes, as shown in Figure 3(a), and then obtains the single node information (Figure 3(b)) through the analysis of the graph file.
2.2. HDL Algorithm Library

This paper collects the calculation types of common neural network models, as shown in Table 1. It can be seen that common neural network models are generally composed of a convolutional layer, fully-connected layer, pooling layer, and activation function. Based on the hardware description language, this paper builds a set of algorithms library oriented to the application of the neural network, which covers the basic layer and operations of common NNs. Meanwhile, the algorithm library takes full use of the high parallelism of the PIM platform and makes the algorithm library run efficiently on the PIM platform by optimizing the parallelism of HDL algorithm library.

| NN     | Layer | CONV | Full connect | Pooling | Activation function |
|--------|-------|------|--------------|---------|--------------------|
| DNN    |       | CONV5-1 | FC          | Max2    | ReLu               |
| LeNet-5|       | FC-500 | FC-10       |         |                   |
|        | CONV11-4 | FC-4096 | Max3        | ReLu    |                   |
|        | CONV3-1 | FC-1000 |            |         |                   |
|        | CONV5-1 |         |             |         |                   |
| AlexNet| CONV1- | FC-1024 | Max2        | ReLu    |                   |
|        | CONV3-1 | FC-4096 |            |         |                   |
|        | CONV7-2 |         |             |         |                   |
| Yolo   | CONV1- | FC-1000 | Avg7        | ReLu    |                   |
|        | CONV3-1 |         |             |         |                   |
| Mobilenet| CONV1- |         |             |         |                   |

Taking LeNet-5, a relatively simple convolutional neural network, as an example, the input two-dimensional image goes through two convolutional layers to the pooling layer, then through the fully-connected layer, and finally uses the activation function to classify it as the output layer. In Table 1, CONV5-1 represents that the size of the convolution kernel is 5*5, the step size is 1; FC-500 refers to that the fully-connected layer with 500 neurons; Max2 indicates that the maximum pooling with a size of 2*2; and the activation function is ReLu.

3. Simulation

We take the common fully-connected layer in a neural network as an example. Each node in the fully-connected layer is connected to every node in the previous layer, which is used to synthesize the features extracted from the previous layers. In a convolutional neural network structure, multiple convolutional layers and pooling layers are connected with one or more fully-connected layers. Similar to the Multilayer Perceptron (MLP), each neuron in the fully-connected layer is connected with all neurons in...
the previous layer, and the fully-connected layer can integrate local information in the convolutional layer or pooling layer.

Figure 4 shows the fully-connected layer in a neural network. A linear relationship is acquired from inputs and outputs, and an intermediate output is obtained from formulation (1), then through a neuron activation function: \( f(x) \).

\[
z = \sum_{i=1}^{m} w_i x_i + b \tag{1}
\]

where \( x_i \) is the input of the fully-connected layer. \( w_i \) is the weight of the input \( x_i \). \( b \) is the bias term.

![Figure 4. Fully-connected layer of a neural network.](image)

Taking a single neuron as an example, \( x_1 \) and \( x_2 \) were respectively random numbers less than 100, \( w_1, w_2 \) and \( b \) term are respectively set as 1, -1, 1. The activation function is represented as formulation (2) \( \text{Relu}(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases} \), and the following simulation waveform (Figure 5) was obtained to verify the feasibility of the method.

![Figure 5. Experimental simulation waveform.](image)

Figure 5 exhibits the inputs(\( x_1, x_2 \)) and the output(\( y \)) of the fully-connected layer. For example, when \( x_1 = 23, x_2 = 16 \) (both in decimal notation here), \( y = \text{Relu}(z) \) and \( z = w_1 x_1 + w_2 x_2 + b \). So \( z = 8 \), and because \( z > 0 \), \( y = z = 8 \). Apparently, the results show that the proposed method is feasible.

4. Conclusion

This paper describes a code conversion method based on a PIM platform. The proposed method allows users to develop applications on the PIM platform with the high-level programming language, makes the PIM platform more accessible in the development of data intensive applications. The simulation results show the validity and effectiveness of the proposed method.

Acknowledgement

This paper is supported by National Nature Science Foundation of China under Grant 61572385. This paper is also supported by Fund of State Key Laboratory of Intense Pulsed Radiation Simulation and Effec under grant SKLIPR2011.
References

[1] Wulf W A, Mckee S A. Hitting the Memory Wall: Implications of the Obvious[J]. Acm Sigarch Computer Architecture News, 1995.

[2] Reagen B, Whatmough P, Adolf R, et al. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators[J]. ACM SIGARCH Computer Architecture News, 2016, 44(3):267-278.

[3] Cavigelli L, Bernath D, Magno M, et al. Computationally efficient target classification in multispectral image data with DNNs[C] International Society for Optics and Photonics, 2016.

[4] Srinivas K, Rani B K, Govrdhan A. Applications of data mining techniques in healthcare and prediction of heart attacks[J]. IJCSE, 2010, 2(02): 250-255.

[5] Mikolov T, Karafiát M, Burget L, et al. Recurrent neural network based language model[C] Eleventh annual conference of the international speech communication association. 2010.

[6] Park S U, Kim T P, Lee M Z, et al. Method of RTL Debugging When Using HLS for HW Design: Different Simulation Result of Verilog & VHDL[C] 2018 International SoC Design Conference (ISOCC). IEEE, 2018: 273-274.

[7] Rongshi D, Yongming T. Accelerator Implementation of Lenet-5 Convolution Neural Network Based on FPGA with HLS[C] 2019 3rd International Conference on Circuits, System and Simulation (ICCSS). IEEE, 2019: 64-67.

[8] Georgopoulos K, Chrysos G, Malakonakis P, et al. An evaluation of vivado HLS for efficient system design[C] International Symposium Elmar. IEEE, 2016.

[9] J. Ambrosi et al., "Hardware-Software Co-Design for an Analog-Digital Accelerator for Machine Learning." 2018 IEEE International Conference on Rebooting Computing (ICRC), McLean, VA, USA, 2018, pp. 1-13.