A Lego-based Neural Network Design Methodology with Flexible NoC

Kun-Chih (Jimmy) Chen, *Senior Member, IEEE,* Cheng-Kang Tsai, Yi-Sheng Liao, Han-Bo Xu, and Masoumeh Ebrahimi, *Senior Member, IEEE*

***Abstract*—Deep Neural Networks (DNNs) have shown superiority in solving the problems of classification and recognition in recent years. However, DNN hardware implementation is challenging due to the high computational complexity and diverse dataflow in different DNN models. To mitigate this design challenge, a large body of research has focused on accelerating specific DNN models or layers and proposed dedicated designs. However, dedicated designs for specific DNN models or layers limit the design flexibility. In this work, we take advantage of the similarity among different DNN models and propose a novel Lego-based Deep Neural Network on a Chip (*DNNoC*) design methodology. We work on common neural computing units (e.g., multiply-accumulation and pooling) and create some neuron computing units called *NeuLego* processing elements ($NeuLego_{PEs}$). These $NeuLego_{PEs}$ are then interconnected using a flexible Network-on-Chip (NoC), allowing to construct different DNN models. To support large-scale DNN models, we enhance the reusability of each $NeuLego_{PE}$ by proposing a Lego placement method. The proposed design methodology allows leveraging different DNN model implementations, helping to reduce implementation cost and time-to-market. Compared with the conventional approaches, the proposed approach can improve the average throughput by 2,802% for given DNN models. Besides, the corresponding hardware is implemented to validate the proposed design methodology, showing on average 12,523% hardware efficiency improvement by considering the throughput and area overhead simultaneously.**

***Index Terms*—Network on Chip (NoC), deep neural network (DNN), accelerator**

I. Introduction

With the advancement of machine learning technology, Deep Neural Networks (DNNs) have shown notable benefits in many real-world applications, such as object detection and speech recognition [1]. However, the high computational complexity and heavy data transmission between neuron layers make the DNN hardware implementation challenging. Therefore, it is necessary to find an efficient DNN design paradigm to reduce the design complexity of the DNN hardware implementation. In this direction, a large body of research has tried to optimize architectures of different DNN models [2]–[4]. Although these approaches reduce the design complexity to construct a DNN accelerator [2]–[5], they are usually developed based on a dedicated structure for specific DNN models or specific DNN layers, which reduces the design flexibility. In other words, designers need to spend a lot of time to find a proper hardware architecture for a given DNN model, which increases the time-to-market significantly.

K.-C. Chen is with the Department of Computer Sciences and Engineering, National Sun Yat-sen University, Taiwan.
E-mail: kcchen@mail.cse.nsysu.edu.tw

C.-K. Tsai and Y.-S. Liao are with the Department of Computer Sciences and Engineering, National Sun Yat-sen University, Taiwan.
E-mail: {angus, ethan}@cereal.cse.nsysu.edu.tw

H.-B. Xu and M. Ebrahimi are with the KTH Royal Institute of Technology, Sweden. E-mail: {hanbox, mebr}@kth.se

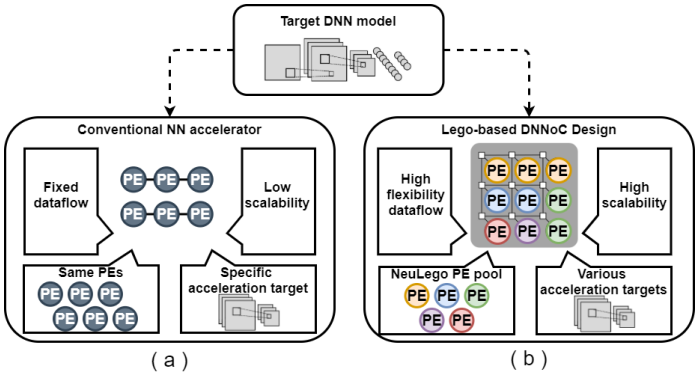

Fig. 1. (a) The conventional NoC-based DNN accelerator design lacks design flexibility; (b) the proposed Lego-based approach can increase the design flexibility and compatibility.

To increase the design flexibility and reduce the time-to-market of implementing different DNN models on hardware, Chen et al. first proposed a Lego-like design methodology to use several neuron computing blocks to construct a DNN model [6]. However, this approach assumes that the constructed DNN accelerator has to compute an entire DNN model, which is not viable to design large-scale DNN models. In addition, the high density of the data transmission between neuron layers worsens the design challenge to implement various kinds of DNN models. To further mitigate the complexity of the data transmission in DNN computing, many efficient interconnection networks for DNN accelerator design were investigated in recent years [7]. Among them, the Network-on-Chip (NoC) interconnection has attracted much attention because of the regular and structural characteristics [8]. Therefore, the NoC-based DNN design methodology has become emerging and been adopted in several works in recent years [5], [9]–[11]. However, most current NoC-based DNN designs are developed based on a certain DNN model, which lacks a systematic design methodology flow, as shown in Fig. 1(a). Therefore, designers still need to spend a lot of time to find proper hardware architectures according to different DNN models.

Because every kind of DNN models use similar operations with different permutations, we apply the concept of Lego-like design methodology, introduced in [6], to develop a systematic

design flow for different DNN model implementations. In this work, we analyze current popular DNN models, such as LeNet [12], AlexNet [13], ResNet-18 [14], and MobileNet v1 [15] to find similar neuron computing operations (e.g., multiply-accumulate operation and pooling operation). Then, different DNN models can be constructed based on the given hyperparameters (e.g., the kernel size in each convolution layer and the number of convolution layers, pooling layers, and dense layers). We identify and construct the common neural network computing functions (e.g., multiply-accumulation and pooling), called *NeuLego* processing elements ($NeuLego_{PEs}$), and connect them to each other by using NoC. Furthermore, we propose a Lego placement method to increase the reusability of the $NeuLego_{PEs}$ when placing them to the target NoC platform. This design concept enhances the scalability of *DNNoC* design methodology to large-scale DNN model implementations. The proposed Lego-based Deep Neural Network on a Chip (*DNNoC*) design methodology is shown in Fig. 1(b).

Although the design flexibility of *DNNoC* can be improved by using NoC interconnection, diverse dataflows in different DNN models and a massive data communication in large-scale DNN models still affect the system performance significantly [10]. To address these issues, we further employ a traffic load reduction method, which is composed of data compression and multicast routing. To sum up, the main contributions of this paper are

- ***Lego-based Deep Neural Network on a Chip (DNNoC) design methodology for higher design flexibility:*** We first construct the common function units used in modern DNN models, called $NeuLego_{PEs}$. We then use the $NeuLego_{PEs}$ and interconnect them using NoC to construct different DNN models. This design approach helps to increase design flexibility of DNN implementations significantly.
- ***Lego placement for the higher Lego reusability:*** To leverage the large-scale DNN implementations, we propose reusing the $NeuLego_{PEs}$ in the platform, which could minimize the number of mapped $NeuLego_{PEs}$ under the constraint of hardware implementation cost. In addition, we propose a dynamic mapping algorithm for being able to compute a large-scale DNN on a small-scale *DNNoC* platform. The implementation of $NeuLego_{PEs}$ is also presented in this work.
- ***Traffic load reduction for higher throughput:*** Due to the huge and complex communications among neurons, we adopt a packet compression method to reduce the packet size. In addition, we employ a novel multicasting approach to reduce the number of transmitted packets on *DNNoC*. In this way, the traffic load on the proposed *DNNoC* can be mitigated significantly.

To verify the proposed Lego-based *DNNoC* design methodology, we modified a cycle-accurate NoC simulator, ESYSim [16], to support neuron computing functions. In addition, the corresponding hardware overhead is analyzed. Because of the flexible and compatible design methodology, the proposed approach can improve average throughput by 2,802% and average hardware efficiency by 12,523% over the conventional design methodologies.

The rest of this paper is organized as follows. We will investigate the state-of-the-art in Section II. Section III will introduce the proposed Lego-based *DNNoC* design methodology. The architecture design for $NeuLego_{PEs}$ and the underlying NoC are described in Section IV. In Section V, we analyze the experimental results. Finally, we conclude this work in Section VI.

II. Related Works

A. DNN Accelerators

In [2], Zhang et al. proposed an FPGA-based DNN accelerator to accelerate the convolution operation. With the programmable feature of FPGAs, the authors evaluated the performance of the convolution operations in different DNN models. However, this work does not support the operations of other layers and thereby lacks computation flexibility. In DLAU [4], Wang et al. proposed a scalable deep learning accelerator unit to support different sizes of DNN models. Although DLAU supports different kernel sizes in the convolution layer of DNN, it does not support operations in other neuron layers, such as those in the pooling or dense layer. In Eyeriss [3], Chen et al. proposed to accelerate the convolution layer through data reuse. Nevertheless, it suffers from low computing flexibility because it cannot compute the dense layer efficiently. The reason is that data sharing in dense layers is much limited than the one in convolution layers. To increase the design flexibility, Chen et al. proposed *KASR-CNN* [6], which uses Lego-like design methodology to increase the computing flexibility of a CNN model. However, the entire CNN model should be computed at once by using this approach. Therefore, it is not viable to apply *KASR-CNN* to design large-scale DNN models.

B. NoC-based DNN Accelerators

In Neu-NoC [10], Liu et al. proposed a high efficient interconnection network to accelerate neuromorphic systems. However, the network does not support DNN operations. Eyeriss v2 [5] is an efficient DNN accelerator architecture for the compact and sparse DNN, which is an extension of Eyeriss by employing the NoC interconnection. Because of the flexible NoC interconnection in Eyeriss v2, it can support different kernel sizes for the convolution operation. Besides, due to the sparsity feature of DNN models, data compression is utilized to improve throughput and energy efficiency. However, because of the applied data reuse method in this structure, many duplicated data is transmitted on NoC. This results in a heavy traffic load on NoC and consequently reduces the throughput.

In [17], Mirmahaleh et al. presented a *mesh-based DNN* accelerator. To mitigate the computing complexity of the target DNN model, the authors employ a weight and neuron pruning method. By using the proposed neuron pruning method, the system performance (i.e., classification accuracy, throughput, etc.) is improved and the computing power consumption is reduced. Furthermore, the authors proposed the DNN mapping method to apply the characteristic of row-weight stationary

TABLE I
THE COMPARISON OF THE DIFFERENT DESIGN METHODOLOGIES

	On-chip interconnection	Neuron mapping strategy	Design methodology
FPGA-based DNN [2]	FPGA crossbar	Layer-wise mapping	Dedicated design
DLAU [4]	Dedicated dataflow	Layer-wise mapping	Dedicated design
Eyeriss [3]	PE array	Layer-wise mapping	Dedicated design
Neu-NoC [10]	NoC	Layer-wise mapping	Dedicated design
Eyeriss v2 [5]	NoC	Layer-wise mapping	Dedicated design
Mesh-based DNN [17]	NoC	Layer-wise mapping	Dedicated design
Tree-based DNN [11]	NoC	Layer-wise mapping	Dedicated design
CMesh-based DNN [18]	NoC	Layer-wise mapping	Dedicated design
UNPU [19]	NoC	Layer-wise mapping	Dedicated design
KASR-CNN [6]	Dedicated dataflow	Model-wise mapping	Lego-based design
Proposed *DNNoC*	NoC	Layer-wise mapping	Lego-based design

(RWS). This method could optimize the dataflow and lighten the traffic load on the NoC. However, this work only focuses on the dataflow between two successive neuron layers. Therefore, it cannot be easily extended to irregular DNN architecture (e.g., the neuron layers are not successive), such as ResNet-18 [14].

Through the topology exploration, Kwon et al. [11] proposed a *tree-based* NoC to reduce the latency of memory accesses. However, the corresponding design methodology for the *tree-based* neural network implementation has not been provided. Reza et al. [18] proposed to use *CMesh-based* NoC for the DNN construction. Because the *CMesh-based* NoC interconnection helps to reduce the data transmission latency, the system throughput can be improved. In addition, the *CMesh-based* NoC involves fewer routers and channels than the conventional NoC architecture, which brings the benefit of the lower area overhead. To further decrease the traffic load on NoC, the authors considered the correlation between layers in a DNN model and the dataflow between neurons to propose a mapping algorithm. However, this work only focuses on the computation flow in the dense layers. Therefore, the proposed approach cannot be easily extended to other kinds of neuron operations such as addition and pooling layers in contemporary DNN models. In [19], Lee et al. proposed UNPU architecture to accelerate the DNN operation by applying lookup table (LUT)-based bit-serial processing element (*LBPE*). Because of the fewer arithmetic operations, UNPU achieves not only a promising reduction of energy consumption but also a significant performance improvement. Besides, the authors select the NoC interconnection to further increase the flexibility of data transmission between *LBPEs*. However, the area overhead is considerable in the UNPU design because of using many lookup tables. Therefore, it is not an efficient way to realize a large-scale DNN model [20].

TABLE I shows the comparison between different design methodologies. Most of the current design methodologies focus on specific neuron operations, which lacks design flexibility. Although *KASR-CNN* [6] adopts Lego-based design methodology to improve design flexibility, the idea is based on mapping the entire DNN model to the given platform at once. Therefore, the *KASR-CNN* design method requires a large area to implement a large-scale DNN model, and thus limiting its applicability. On the other hand, the proposed

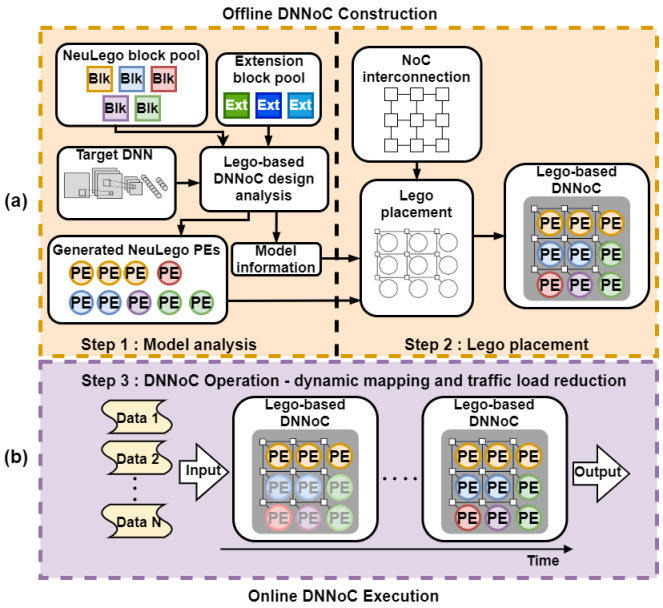

Fig. 2. Design flow of the proposed Lego-based *DNNoC* design methodology is composed of (a) offline *DNNoC* construction and (b) online *DNNoC* execution.

DNNoC uses NoC interconnection to leverage the Lego-based design methodology. By using the proposed layer-wise mapping strategy, the computing resources can be reused to process the neuron operations of different neuron layers. In this way, the proposed approach improves both design as well as computing flexibility.

III. THE PROPOSED LEGO-BASED DEEP NEURAL NETWORK ON A CHIP (*DNNoC*) DESIGN METHODOLOGY

The proposed Lego-based *DNNoC* design flow is shown in Fig. 2, which is composed of:

- ***DNNoC construction:*** In this stage, we use NoC interconnection to leverage the *DNNoC* design, which is performed offline.
- ***DNNoC execution:*** The *DNNoC* platform is used to compute the target DNN model at runtime.

A. DNNoC Construction

Fig. 2(a) shows that the *DNNoC Construction* stage can be further divided into (1) *Model Analysis phase* and (2) *Lego*

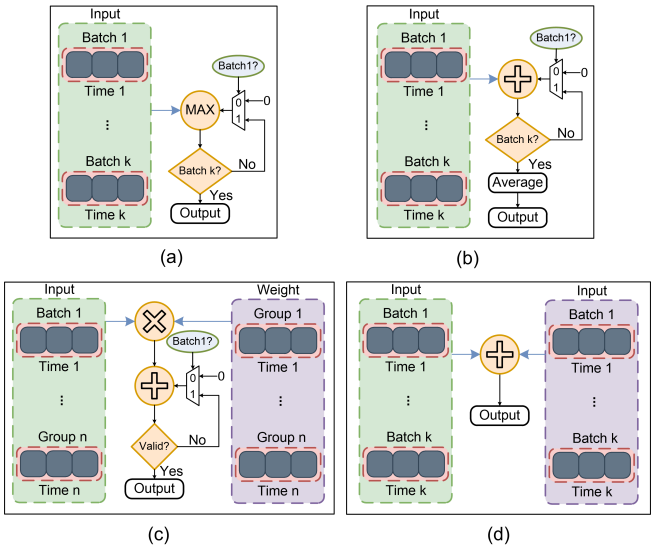

Fig. 3. *NeuLego* block design to perform (a) MaxPooling, (b) Global Average Pooling, (c) Multiply-accumulation, and (d) Addition.

Placement phase. The *Model Analysis phase* is used to analyze the target DNN model and determine the involved neuron operations. In this phase, the corresponding $NeuLego_{PEs}$ are generated. A $NeuLego_{PE}$ includes a specific neuron operation, called NeuLego block ($NeuLego_{blk}$), and a corresponding activation function block, called extension block. In this way, different $NeuLego_{blks}$ and extension blocks can be combined to form different $NeuLego_{PEs}$ and perform diverse neuron operations, which increases design flexibility. On the other hand, the *Lego Placement phase* is used to place the selected $NeuLego_{PEs}$ and interconnect them through NoC to construct the corresponding *DNNoC*. The details of the two phases will be introduced below.

1) NeuLego block design and model analysis: To design a proper $NeuLego_{blk}$, we first explore four popular DNN models (i.e., LeNet [12], AlexNet [13], ResNet-18 [14], and MobileNet v1 [15]) as design examples and extract the identical neuron operations to build a *NeuLego* block pool. The *NeuLego* block pool contains many common neuron operation units, and it can be easily extended to support new DNN models. The four common layers in our target DNN models are (1) *Max Pooling* (*MP*), (2) *Global Average Pooling* (*GAP*), (3) *Multiply-accumulation* (*MA*), and (4) *Addition* (*Add*). Therefore, we design four kinds of $NeuLego_{blks}$ (i.e., *MP*, *GAP*, *MA*, and *Add*), and each $NeuLego_{blk}$ is used to perform the corresponding neuron operation to the received input data. Obviously, the size of a $NeuLego_{blk}$ becomes large if it need to compute a massive amount of input data. Hence, we define a batch parameter to indicate the computing capacity of a $NeuLego_{blk}$ and limit its size. This may cause additional computation iterations before outputting a result. In other words, the batch parameter is used to define the number of inputs, in which the $NeuLego_{blk}$ can support for the neuron operation at each computing iteration. The processing flow of each kind of $NeuLego_{blk}$ is shown in Fig. 3.

As shown in Fig. 3(a), the *MP* block is used to find the maximum value among input data for the given kernel size and the channel. By defining the batch parameter, the *MP* block finds the partial maximum value at each computing iteration and obtain the global maximum value after several computing iterations. On the other hand, as shown in Fig. 3(b), the *GAP* operation is used to calculate the average of the input data, and the computing iterations depend on the batch parameter.

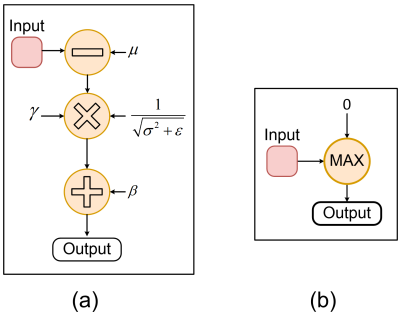

Fig. 4. Extension block design to perform (a) Batch normalization, and (b) ReLU.

The *MA* operation is usually adopted in convolution, depthwise convolution, and dense layers, which is used to multiply and accumulate the input data with the corresponding weight data, as shown in Fig. 3(c). The *MA* $NeuLego_{PE}$ is used to compute the convolution operations in a channel. In other words, the convolution operations of different channels are computed in different *MA* $NeuLego_{PEs}$. The *MA* $NeuLego_{PE}$ also supports the traditional multiply-accumulation operations in dense layers. The required computing iterations of the *MA* block depend on the given batch parameter. Finally, we design an *Add* block as shown in Fig. 3(d). The addition layer is widely employed in ResNet-18 [14]. It is also used to skip the connections between two successive layers and to achieve cross-layer operation. The *Add* operation will add two input data and produce the result where the size of output data is the same as the input data. Therefore, the adder in the *Add* block does not need the result from the previous computing iteration (i.e., without MUX in Fig. 3(d)).

In addition to the neuron computing for the data process, normalization layers or activation function layers are often used in contemporary DNN models. These two kinds of neuron functions are usually used to assist the feature extraction layers, such as convolution and dense layers, and to improve the training efficiency. Among them, the Batch Normalization (*BN*) layer and the Rectified Linear Unit (ReLU) layer [14] are widely used. Therefore, we further design these extension blocks, which can be concatenated to the end of the $NeuLego_{blks}$. Regarding the *BN* block, as shown in Fig. 4(a), it normalizes the output of the preceding $NeuLego_{blks}$. The *BN* operation can be expressed by:

$$f(x) = \frac{\gamma(x-\mu)}{\sqrt{\sigma^2+\varepsilon}} + \beta, \tag{1}$$

where x is the output of a certain $NeuLego_{blk}$. The μ and σ^2 are the mean and variance of the entire batch of training data. The γ and β are the parameters of the *BN*, which can be obtained in the training phase. Besides, ε is a constant

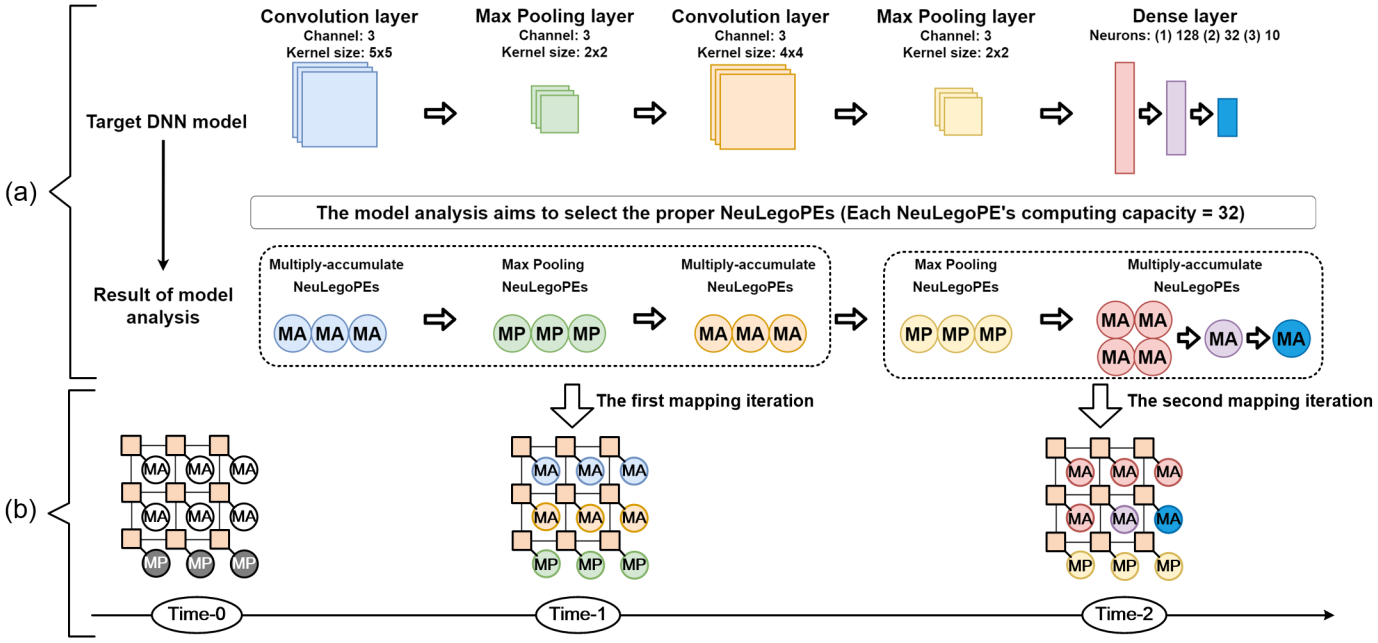

Fig. 5. Example of a (a) DNN analysis flow at design time and (b) the corresponding *DNNoC* computing flow at runtime.

to avoid the value of the denominator from zero. In the normalization stage of training, the mean and variance of the entire batch of training data will be calculated. However, in the inference phase, the data enters in sequence, usually without batching, so here we use the mean and variance learned by the model during the training phase to normalize the input data. Therefore, $\sqrt{\sigma^2+\varepsilon}$ can be calculated before the inference phase. On the other hand, the ReLU block sets the input data smaller than zero to zero, while the numbers greater than zero remain unchanged, as shown in Fig. 4(b). The ReLU function can be expressed by

$$f(x) = max(0, x), \tag{2}$$

where x is the output of a certain $NeuLego_{blks}$. The aforementioned extension blocks can be combined with variety of $NeuLego_{blks}$.

After analyzing the neuron operations, the corresponding $NeuLego_{blks}$ and extension blocks are selected for the given DNN model. These blocks should be placed and interconnected in the *DNNoC* platform. To increase the reusability of the processing elements, we group several identical $NeuLego_{blks}$ along with their corresponding extension blocks and assign them to one $NeuLego_{PEs}$. The number of $NeuLego_{blks}$ in a $NeuLego_{PE}$ depends on the computing capacity of the $NeuLego_{PE}$, determined by design specification. In this work, the size of a $NeuLego_{PE}$ is defined as the number of $NeuLego_{blks}$ in the $NeuLego_{PE}$. Therefore, the required number of $NeuLego_{PEs}$ in the constructed *DNNoC*, $Num(NeuLego_{PE})$, can be represented by:

$$Num(NeuLego_{PE}) = \sum_{i=1}^{L}\sum_{j=1}^{K}\frac{Num(NeuLego_{blk})_{i,j}}{C_j}, \tag{3}$$

where L represents the number of layers, and K indicates the number of $NeuLego_{blk}$ types. With this definition, $Num(NeuLego_{blk})_{i,j}$ shows the number of $NeuLego_{blks}$ of type j in the i^{th} layer. Besides, C_j indicates the computing capacity of a $NeuLego_{PE}$ to execute the operations in the j-type block.

Fig. 5(a) illustrates an example of the proposed model analysis flow. First, for a given DNN model, the required number of the $NeuLego_{blks}$ and their types will be extracted. In this figure, DNN is composed of two convolution layers with different kernel sizes, two max-pooling layers with identical channel size, and three dense layers with different sizes. Besides, in this example, we assume each $NeuLego_{PE}$ can compute 32 neuron operations in each iteration, which may correspond to the operations in max-pooling, global average pooling, multiply-accumulation, and addition layers. Also, we assume that each $NeuLego_{PE}$ is used to process the data from the same data dimension. As a result, each *MA* $NeuLego_{PE}$ for the convolution layer and each *MP* $NeuLego_{PE}$ for the max-pooling layer compute the data of the same channel. Regarding the example in Fig. 5(a), the first convolution layer has 3 channels and the computation within each channel is 25 (i.e., less than 32). Therefore, three *MA* $NeuLego_{PEs}$ are enough to compute the operations, each assigned to one channel. Similarly, three *MP* $NeuLego_{PEs}$ are required to compute the first max pooling layer; three *MA* $NeuLego_{PEs}$ are used to compute the second convolution layer; and three *MP* $NeuLego_{PEs}$ are needed to compute the second max pooling layer. On the other hand, each *MA* $NeuLego_{PE}$ for the dense layer assists with the data process in the same layer. For example, four, one, and one *MA* $NeuLego_{PEs}$ are selected for the first, second, and third dense layer, respectively, which is determined by the total number of neurons in each dense layer and the supported computing capacity of each PE (i.e., the $NeuLego_{PE}$ supports 32 neuron operations at one time). Based on this layer-wise model analysis, the proper number

of $NeuLego_{PEs}$ can be determined, and the required number of $NeuLego_{PEs}$ is 18 in this example.

2) Lego placement with NoC interconnection: After generating the required $NeuLego_{PEs}$, it is necessary to place and interconnect them by using a NoC interconnection. However, the area overhead would be large if all determined $NeuLego_{PEs}$ are mapped to the NoC platform at the same time. As mentioned before, DNNs have similar computing behaviors in different neuron layers. Therefore, we have an opportunity to employ fewer $NeuLego_{PEs}$ to compute similar neuron computing, such as convolution and multiply-accumulation.

To take advantage of this opportunity, we propose to share the computing resources and find the proper number of $NeuLego_{PEs}$ for a given DNN model. To support all operations in the target DNN model, the number of $NeuLego_{PEs}$ for each $NeuLego_{PE}$ type is determined based on the worst-case consideration. In this work, we select the maximum number of $NeuLego_{PEs}$, which will be needed at the same time for each $NeuLego_{PE}$ type. Hence, the total number of $NeuLego_{PEs}$, $TNum_{PE}$, in a *DNNoC* construction, can be modeled by

$$TNum_{PE} = \sum_{j=1}^{K} arg \max_{i=1}^{L} Num(NeuLego_{PE})_{i,j}. \quad (4)$$

As an example in Fig. 5(a), the largest number of required *MA* $NeuLego_{PEs}$ and *MP* $NeuLego_{PEs}$ are four and three, respectively, at the same time. Therefore, it is enough to use seven $NeuLego_{PEs}$ to compute this target DNN model.

With the information about $TNum_{PE}$, the proper NoC size can be determined. Considering an $N \times N$ NoC, N can be defined by:

$$N = \left\lceil \sqrt{TNum_{PE}} \right\rceil. \quad (5)$$

Fig. 6 shows the flowchart of the proposed mapping algorithm to efficiently map $NeuLego_{PEs}$ on the target $N \times N$ NoC. The algorithm includes three steps: (1) $NeuLego_{PE}$ pool creation; (2) row-based $NeuLego_{PE}$ alignment; and (3) $NeuLego_{PE}$ filling and NoC interconnection. Fig. 7 illustrates a mapping example to construct a *DNNoC* for the given DNN model in the example of Fig. 5(a). First, by using the results of the model analysis, the corresponding $NeuLego_{PE}$ pool is created, as shown in Fig. 7(a). The $NeuLego_{PE}$ pool includes $TNum_{PE}$ $NeuLego_{PEs}$ (i.e., 7 $NeuLego_{PEs}$ in this example) with multiple types. In the row-based $NeuLego_{PE}$ alignment stage, a proper number of $NeuLego_{PEs}$ related to different $NeuLego_{PE}$ types will be selected from the $NeuLego_{PE}$ pool. Then, the selected $NeuLego_{PEs}$ will be placed along a row of the target $N \times N$ NoC. To facilitate the data exchange on the *DNNoC*, the identical $NeuLego_{PE}$ types are placed close to each other. Different types of $NeuLego_{PE}$ are placed in different rows as much as possible, as shown in Fig. 7(b). After aligning the $NeuLego_{PEs}$ along the rows, some empty $NeuLego_{PE}$ placement slots might be left, as shown in Fig. 7(b). If there are still some $NeuLego_{PEs}$ in the $NeuLego_{PE}$ pool, they will be placed into these empty $NeuLego_{PE}$ slots, by filling the rows in an ascending order. Otherwise, the identical-type $NeuLego_{PEs}$, which might

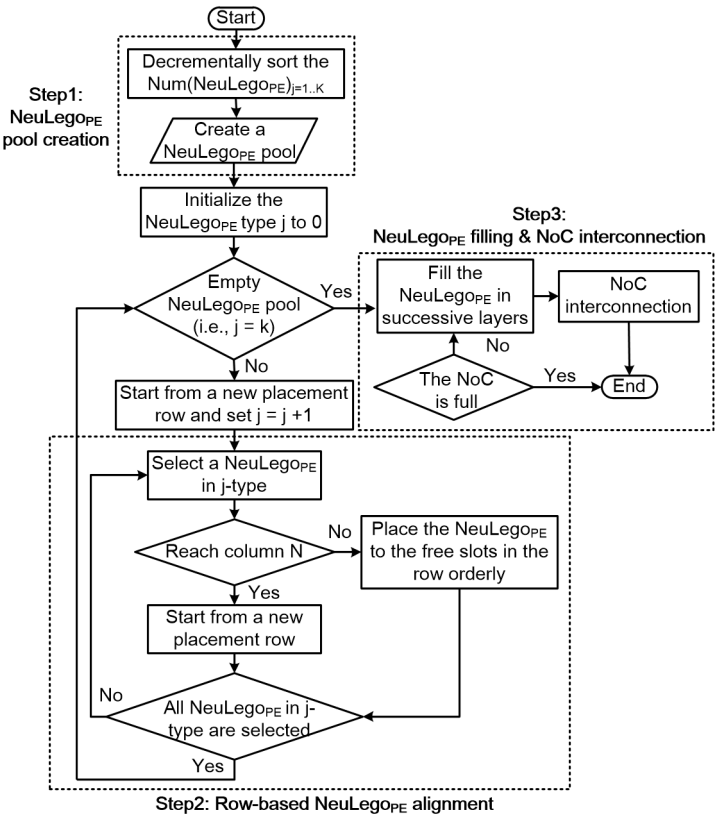

Fig. 6. The proposed $NeuLego_{PE}$ placement and *DNNoC* construction flow.

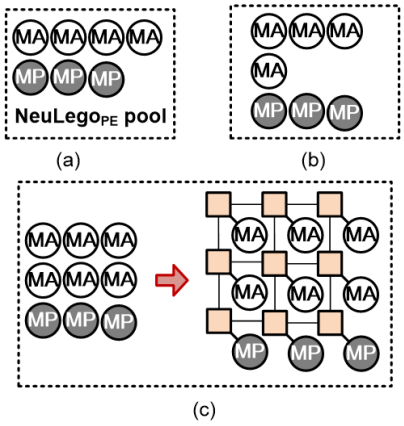

Fig. 7. The *DNNoC* construction flow includes (a) $NeuLego_{PE}$ pool creation, (b) row-based $NeuLego_{PE}$ alignment, and (c) $NeuLego_{PE}$ filling and NoC interconnection.

be needed in the successive layers, will be used to fill in these empty $NeuLego_{PE}$ placement slots. As shown in the example of Fig. 7(c), the *MA* $NeuLego_{PEs}$ are selected to fill in the empty $NeuLego_{PE}$ placement slots because *MA* $NeuLego_{PEs}$ are used in successive dense layers. In this way, we can provide more computing resources for those frequently used $NeuLego_{PE}$ types, and thus improving the system performance. After placing each $NeuLego_{PE}$, the corresponding *DNNoC* can be constructed by using the NoC interconnection, as shown in Fig. 7(c).

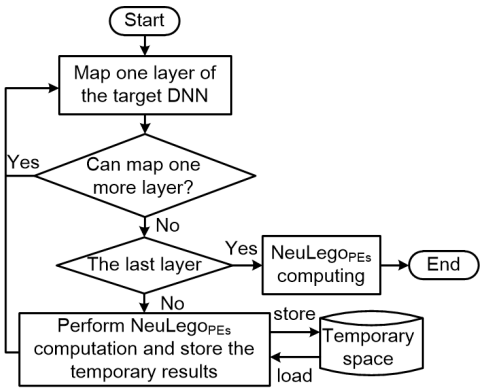

Fig. 8. The flow chart of the proposed layer-wise dynamic mapping algorithm.

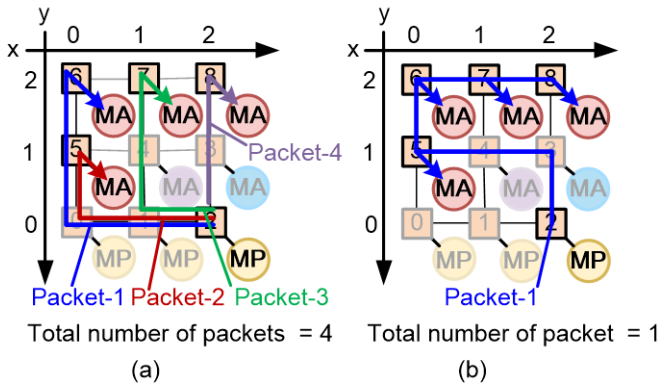

Fig. 9. (a) Many duplicated data delivered on the NoC, which worsens the traffic load, and (b) the Hamiltonian multicast routing algorithm can reduce the traffic load significantly.

B. DNNoC Execution

After constructing the corresponding *DNNoC*, it will be used to compute the target DNN model in the *DNNoC Execution* stage, as shwon in Fig. 2(b). As mentioned before, we can use fewer $NeuLego_{PEs}$ to compute large-scale DNN model by reusing the computing resource in each $NeuLego_{PE}$, which helps to reduce the area overhead of the constructed *DNNoC*. However, it is difficult to compute a large-scale DNN on the resource-limited *DNNoC* platform, which includes different types of $NeuLego_{PEs}$. Therefore, the challenges in this stage are (1) how to map a large-scale DNN model to the limited-size *DNNoC* platform and (2) how to reduce the heavy traffic load on the *DNNoC*. The two design challenges will be addressed below.

1) Dynamic mapping for large-scale DNN computing: To address the first problem of this stage, we propose a layer-wise dynamic mapping metho, and the flowchart is shown in Fig. 8. The only assumption is that the available computing resources in the constructed *DNNoC* should be sufficient to fit the largest layer of the target DNN model. Fig. 5(b) shows an example of this mapping method. Because the constructed *DNNoC* platform in this example includes six *MA* $NeuLego_{PEs}$ and three *MP* $NeuLego_{PEs}$, we can compute the neuron operations from the first to the third neuron layers of the target DNN at once (i.e., the first mapping iteration in Fig. 5(b)). After completing the computations of the first mapping iteration, the temporary results will be stored in the off-chip memory. To complete the entire DNN computation, the neuron operations in the three *MP* $NeuLego_{PEs}$ in the Max Pooling layer and six *MA* $NeuLego_{PEs}$ in the three Dense layers are required to be computed as well. Fortunately, the available computing resources in *DNNoC* are sufficient to support the neuron operations in the remaining layers, and thus they are mapped in the second mapping iteration. In this way, a large-scale DNN can be computed on a resource-limited *DNNoC* platform.

2) Traffic load reduction for throughput improvement: Although the proposed *DNNoC* design methodology leverages the DNN accelerator design, massive data communication in large-scale DNN models still affect the system performance significantly [10]. The reason is that the outputs of the current neuron layer are required by many neurons in the next layer, and thus many duplicated packets should be transmitted on the NoC platform. In addition, because of the ReLU function, many identical computing results from the different $NeuLego_{blks}$ in the same $NeuLego_{PEs}$ will be packetized in a packet, which enlarges the packet size during the data transmission.

Fig. 9(a) illustrates an example when the node ID 2, which includes an *MP* $NeuLego_{PE}$, aims to send packets to node IDs 5, 6, 7, and 8. Obviously, it should create four packets to send the identical data when applying the conventional unicast routing algorithm. The corresponding header format is shown in Fig. 10(a). Note that the length of the *Packet Body* depends on the computing capacity (C_j in Equation (3)) of the current $NeuLego_{PE}$. Besides, the bitwidth of each computing result in the *Packet Body* is assumed as B bits in this work. Hence, the required bitwidth of a unicast routing packet, $BW(Pkt_{unicast})$, can be modeled as

$$BW(Pkt_{unicast}) = 6 + 4\lceil logN \rceil + C_j \times B \qquad (6)$$

in an $N \times N$ *DNNoC*. Therefore, it is not cost-efficient to use several packets to transmit identical data, which leads to the heavy traffic load.

To solve this problem, we employ the Hamiltonian routing algorithm [21] to implement the multicast routing. Fig. 9(b) illustrates an example, and the corresponding packet format is shown in Fig. 10(b). Different from the packet format for unicast routing in Fig. 10(a), all destination IDs should be recorded in the *Packet Header*. If the packet arrives at one of its destination nodes, the data in the packet will be copied to the local $NeuLego_{PE}$ for further computation. Meanwhile, the bit corresponding to this destination ID in the *Packet Header* is set to 0, indicating that this destination has been reached. Afterward, this packet proceeds to the next destinations until reaching the last destination node (i.e., the node ID 8 in Fig. 9(b)). Since all destination IDs are recorded in the *Packet Header*, the total number of required packets can be reduced significantly. However, a longer *Packet Header* is still required, which worsens the traffic congestion on the *DNNoC*. For an $N \times N$ *DNNoC*, the required bitwidth for a

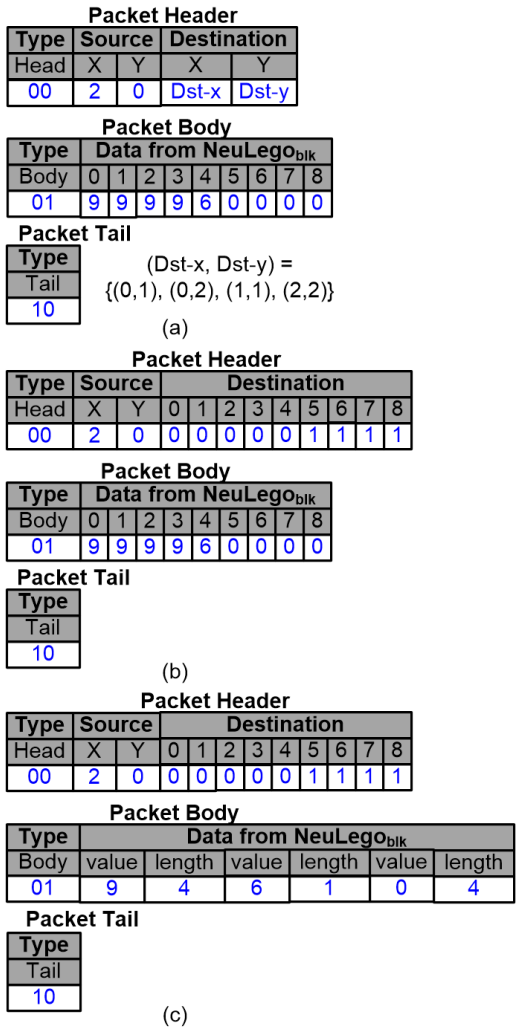

Fig. 10. (a) The conventional multicast routing delivers several identical packets to different destinations; (b) In Hamiltonian multicast routing, the *Packet Body* stores the entire data; (c) The run-length encoding method can further compress the successive data efficiently in Hamiltonian multicast routing.

multicast routing packet, $BW(Pkt_{multicast})$, can be modeled by

$$BW(Pkt_{multicast}) = 6 + 2\lceil logN \rceil + N^2 + C_j \times B, \quad (7)$$

where C_j represents the computing capacity of the current j-type $NeuLego_{PE}$ and B is the bitwidth of each data in *Packet Body*. Obviously, this kind of multicast routing packet is longer than the conventional unicast routing packet, which counteracts the advantage of multicast routing.

To further solve the problem of transmission bandwidth at runtime, we compress the data by considering the fact that there are many successive data in a packet after the ReLU function. The run-length encoding algorithm [22] is a popular method applied in contemporary DNN accelerators [3] to fold the successive data and reduce the data size before storing it to the external memory. In this work, we use the run-length encoding algorithm to fold the successive data in a packet and reduce the traffic load on the *DNNoC*. The run-length encoding algorithm is used to compress the series of identical data in the *Packet Body* into two fields (i.e., value field and length field), as shown in Fig. 10(c). The required bitwidth of the length field is equal to $\lceil logC_j \rceil$, which depends on the computing capacity of the current $NeuLego_{PE}$. On the other hand, as mentioned before, we require B to represent the data in the value field in Fig. 10(c). In case of data series in the *Packet Body* including S kinds of successive identical data sub-series and I individual data, the required bitwidth of a multicast routing packet with run-length encoding data would be equal to $6 + 2\lceil logN \rceil + N^2 + (S + I) \times (B + \lceil logC_j \rceil)$.

Obviously, the required bitwidth of the *Packet Body* after the run-length encoding process highly depends on the permutation of the data series with C_j data in the *Packet Body* of the conventional multicast routing packet, as shown in Fig. 10(b). Therefore, it is necessary to analyze the expected size of the multicast routing packet with run-length encoding data. For a case of data series with C_j B-bit data, there are $(2^B)^{C_j}$ kinds of possible data series. Among them, there are $(\sum_{i=0}^{C_j-1} \binom{C_j-1}{i} \cdot 2^B \cdot (2^B-1)^i \cdot (i+1))$ cases, which include at least one successive identical data sub-series in the data series with C_j data. In addition, we need $(B + \lceil logC_j \rceil)$ bits to represent an encoding unit (i.e., pair of (value, length) in Fig. 10(c).) Therefore, the expected bitwidth of the packet size with run-length encoding data, $BW(Pkt_{mul\_run})$, is:

$$\begin{aligned} BW(Pkt_{mul\_run}) = {} & 6 + 2\lceil logN \rceil + N^2 \\ & + \frac{\sum_{i=0}^{C_j-1} \binom{C_j-1}{i} \cdot 2^B \cdot (2^B-1)^i \cdot (i+1)}{(2^B)^{C_j}} \\ & \times (B + \lceil logC_j \rceil). \end{aligned} \quad (8)$$

IV. Hardware Architecture Design of $NeuLego_{PEs}$

In order to facilitate the *DNNoC* design methodology, the architecture of a constructed *DNNoC* is shown in Fig. 11, which includes (1) $NeuLego_{PE}$ architecture designs for each neuron computation, and (2) router and network interface (NI) designs for each $NeuLego_{PE}$ interconnection. The controller sends the control signal to each $NeuLego_{PE}$ to catch the necessary data from the global buffer and receive information about the current mapping situation. The global buffer size is determined by the maximum number of parameters (i.e., weights) in a single neuron layer of the target DNN model. Based on the global control signals, the PE controller controls the movement of the necessary input data or weight data from the local data buffer of the current *DNNoC* tile to the $NeuLego_{PE}$ for further neuron operations. Then, the results produced in a $NeuLego_{PE}$ will be transmitted to other $NeuLego_{PEs}$ through NoC interconnection. This is the case if multiple neuron layers are mapped to the NoC platform at the current mapping iteration.

A. $NeuLego_{PE}$ Architecture Design

As mentioned before, the required $NeuLego_{PEs}$ will be determined after the model analysis. Besides, each $NeuLego_{PE}$ may contain several $NeuLego_{blks}$, which are used to perform the corresponding neuron operations. The four mentioned

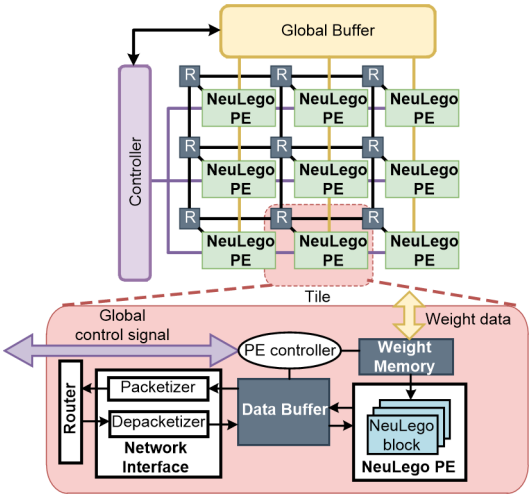

Fig. 11. Each $NeuLego_{PE}$ can be interconnected by using NoC interconnection to construct a *DNNoC* architecture.

kinds of $NeuLego_{PEs}$ can be further classified into memory-less $NeuLego_{PEs}$ and memory-based $NeuLego_{PEs}$. In memory-less $NeuLego_{PEs}$, such as *MP* and *Add*, the computing results can be obtained without extra weight input from the weight memory within a tile. On the contrary, in memory-based $NeuLego_{PEs}$, such as *GAP* and *MA*, additional weight information is needed from weight memory to compute the results.

Fig. 12(a) shows the architecture of the memory-less $NeuLego_{PE}$, which is composed of several $NeuLego_{blks}$. These blocks are executed in parallel. When the memory-less $NeuLego_{PE}$ collects all necessary data, the *Block Controller* of each $NeuLego_{blk}$ will sequentially capture a batch of data from the local *Input Buffer* for further neuron operation (i.e., *MP* or *Add*) until reaching the final computation results. After obtaining the final results, the *Block Controller* will activate the D flip-flop and output the results of this $NeuLego_{blk}$. Meanwhile, the *Block Controller* resets the *Weight-less Neuron Operator* for the next computing iteration.

Fig. 12(b) shows the architecture of the memory-based $NeuLego_{PE}$. Similarly, the memory-based $NeuLego_{PE}$ uses several $NeuLego_{blks}$ to perform parallel neuron operations (i.e., *GAP* or *MA*). First, the *Block Controller* catches the batch of data for each $NeuLego_{blk}$ from the local *Input Buffer*, and the *Weight-based Neuron Operator* is used to perform the corresponding neuron operation until producing the final computation results. The required weights for the neuron computation are stored in the *Weight Memory* in the *DNNoC* tile. After computing the final results, the *Block Controller* activates the D flip-flop to output the results. It also resets the *Weight-based Neuron Operator* for the next computing iteration.

In addition to the $NeuLego_{PE}$ design, as mentioned before, two extension PEs are designed to perform batch normalization and ReLU operations. The extension PEs are used to perform the corresponding data post-processing for the output of some $NeuLego_{PEs}$. An extension PE is composed of several extension blocks, and the number of extension blocks should

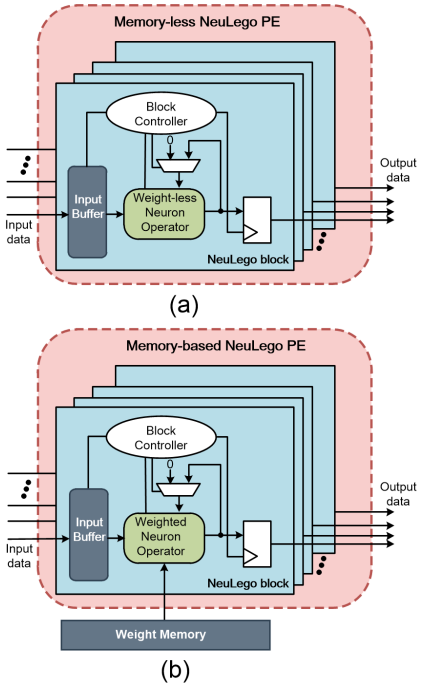

Fig. 12. The $NeuLego_{PE}$ can be classified into (a) memory-less $NeuLego_{PE}$ and (b) memory-based $NeuLego_{PE}$.

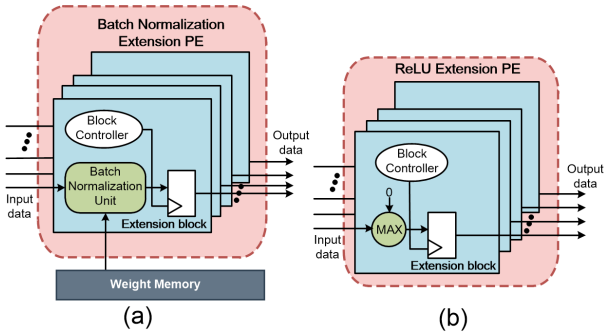

Fig. 13. (a) Batch normalization extension PE and (b) ReLU extension PE are used to post-process the data from the $NeuLego_{PEs}$.

be equal to the number of $NeuLego_{blks}$ of the connected $NeuLego_{PE}$. Fig. 13(a) shows the extension PE architecture for the batch normalization in Equation (1). Note that the parameters (i.e., μ, $\sqrt{\sigma^2+\varepsilon}$, γ, and β in Equation (1)) for the operation of the batch normalization is captured from the *Weight Memory* in the *DNNoC* tile. On the other hand, Fig. 13(b) shows the ReLU extension PE, which is used to perform the ReLU activation function in Equation (2). With the $NeuLego_{PEs}$ and the extension PEs, the design flexibility of the proposed *DNNoC* design can be improved significantly. In other words, we can construct the target *DNNoC* by combining the proper $NeuLego_{PEs}$ and the extension PEs. Please note that the extension PE can be considered as a post-processor of the $NeuLego_{PEs}$. If a $NeuLego_{PE}$ needs it, it can be patched to the output of the $NeuLego_{PE}$. Hence, the $NeuLego_{PE}$ and the extension PE should be placed in the same *DNNoC* tile.

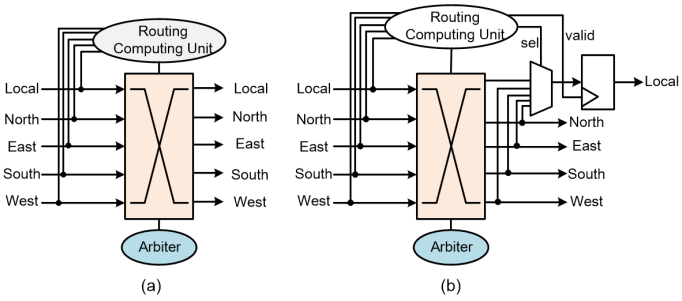

Fig. 14. (a) The conventional router design cannot support Hamiltonian routing algorithm, and (b) the proposed router architecture.

B. Router and Interface Design for DNNoC Interconnection

To support the multicast routing to deliver the compressed packets from a $NeuLego_{PE}$, the router and the network interface (NI) should be properly designed. Fig. 14(a) shows the conventional router design. When a packet arrives at the router through an input port, the routing computing unit will analyze the routing information in the header of the packet. Then, this packet will be delivered to the corresponding output port with the assistance of the crossbar. Fig. 14(b) shows the proposed router architecture, supporting the Hamiltonian routing algorithm. In addition to the original structure in the conventional router architecture, the routing computation unit is additionally used to determine whether the current node is one of the destinations of the multicast packet. The packet will be replicated to the local direction by using a MUX, and the *sel* signal is assigned to indicate the direction, from where the replicated packet comes from. Meanwhile, the *valid* signal is set to let the replicated packet pass through the D flip-flop to the local direction. On the contrary, the *sel* signal will be unknown, and the *valid* signal will not be set when the current node is not one of the destinations. In this way, we can block the invalid packets, preventing them to enter the local direction.

In addition to the router design, NI design is also an important component in the proposed *DNNoC* because it is used to generate the multicast packets and to depacketize the received packets. Fig. 15(a) shows that the NI is located at the position between the $NeuLego_{PE}$ and the router. The NI is predominately composed of a packetizer and depacketizer, and the corresponding architectures are shown in Fig. 15(b) and Fig. 15(c), respectively. The packetizer (Fig. 15(b)), first counts the length of each replicated value to make the corresponding encoding unit (i.e., a pair of (value, length) in Fig. 10(b)). After having an encoding unit, in case the incoming data is different from the previous one, the *enable* signal in Fig. 15(b) will be raised, and the built encoding unit will be captured by the *Body Encoder* to concatenate with the existing encoding unit(s). Meanwhile, the counter will be reset to count for the next successive duplicated value. When the $NeuLego_{PE}$ sends out all computing results, the *Done* signal is raised to represent that the body of the packet, including several encoding units, is generated. Afterward, the *Packing* unit packetizes the corresponding header, body, and tail to form a complete packet. On the other hand, the depacketizer

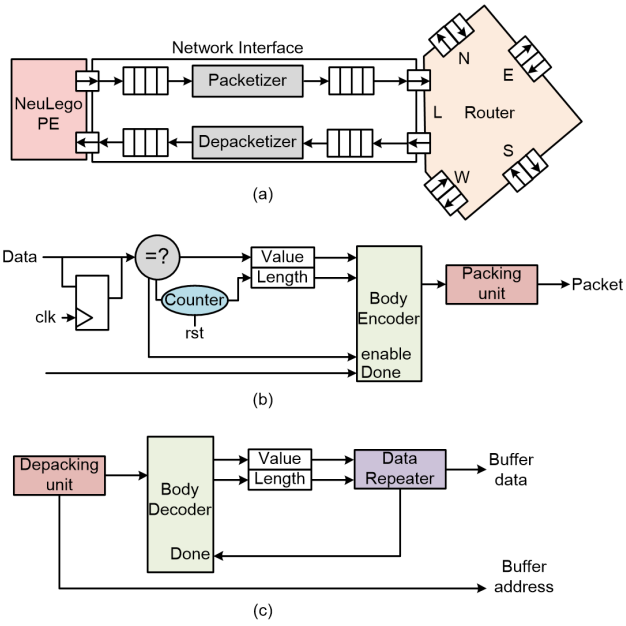

Fig. 15. (a) NI is used to be the interface between the $NeuLego_{PE}$ and the router, which is composed of (b) packetizer and (c) depacketizer.

in Fig. 15(c) decomposes the received packets and extracts the header and body of the packet. The extracted data from the body will be analyzed by the *Body Decoder* unit to capture the information of each encoding unit, which includes the length of the replicated data. By using the information of the encoding unit, the *Data Repeater* unit will generate the corresponding length of the replicated data and store them to the data buffer for further $NeuLego_{PE}$ process. After completing an encoding unit process, the *Done* signal will be raised, and the *Data Repeater* unit will wait for the next encoding unit from the *Body Decoder*.

V. Experimental Results and Discussion

To verify the proposed design methodology, we employ a NoC simulator, called ESYSim [16]. Although the ESYSim supports the analysis of traffic behavior on the defined NoC platform, it does not support any neural computing functions. In this work, we modified ESYSim to support different neural computing functions. To launch the *DNNoC* simulation, we input the defined DNN model file, including the neuron operation type, model parameters, and the interconnection between neurons. In this way, the modified ESYSim can realize high-level simulation to enable early architectural exploration. In this section, we will analyze the performance and the hardware cost of the proposed *DNNoC* design methodology by using this simulator. In the following experiments, four different common DNN models with different datasets are analyzed (i.e., LeNet [12] with the MNIST [23] dataset and ResNet-18 [14], AlexNet [13], and MobileNet v1 [15] with the ImageNet [24] dataset). To place the $NeuLego_{PEs}$ on the NoC platform, we employ the *Direct-X* [25] algorithm along with the proposed dynamic mapping strategy. Since the bandwidth limitation between the off-chip memory and global buffer affects the overall performance, similar to [5], we

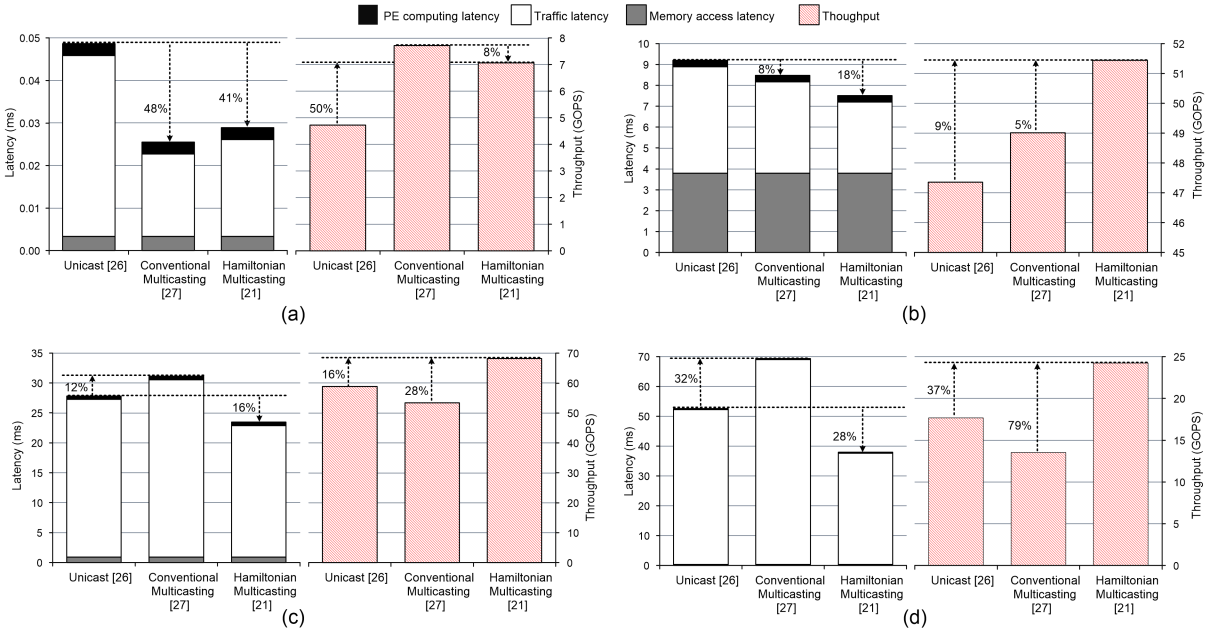

Fig. 16. The performance comparison by using different routing strategies under (a) LeNet, (b) AlexNet, (c) ResNet-18, and (d) MobileNet v1 models.

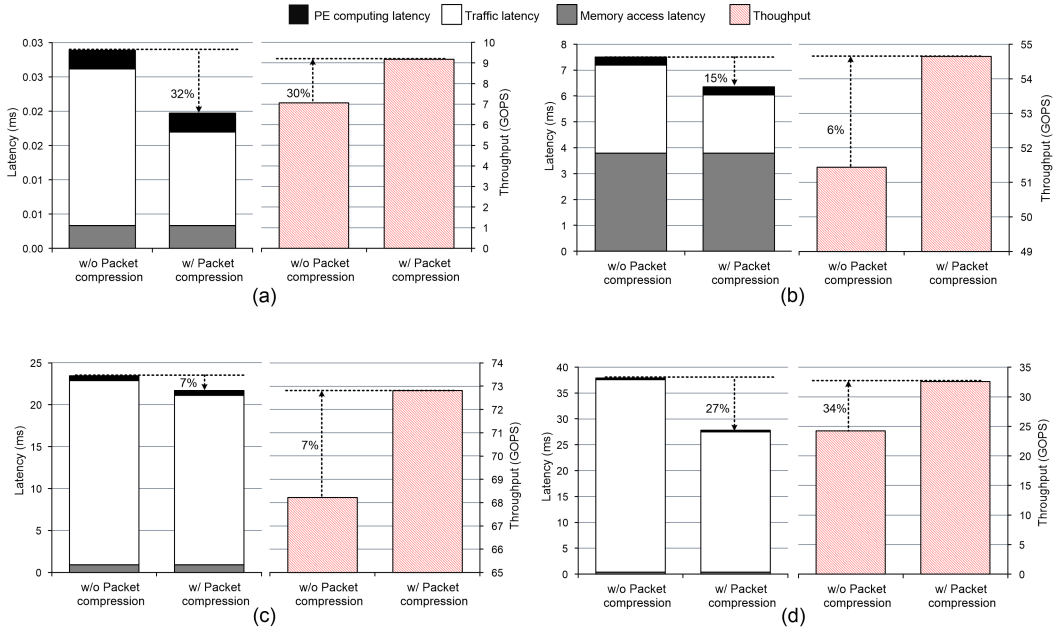

Fig. 17. The performance analysis after using packet compression techniques under (a) LeNet, (b) AlexNet, (c) ResNet-18, and (d) MobileNet v1 models.

employ DDR4-3200 SDRAM memory, providing a bandwidth of 25600 MB/s for read and write operations.

A. Performance Comparison and Analysis

To explore the performance of the *DNNoC*, we first analyze the performance improvement on a 4x4 *DNNoC* according to different DNN models when the Hamiltonian multicast routing algorithm and the packet compression method are applied. As discussed in the $NeuLego_{PE}$ design, we group several $NeuLego_{blks}$ into a $NeuLego_{PE}$ to increase the reusability of the processing elements. In this experiment, the size of $NeuLego_{PE}$ is set to 2, 64, 64, and 64 for LeNet, AlexNet, ResNet-18, and MobileNet v1, respectively. In addition, we assume that each $NeuLego_{blk}$ supports 32 neural operations at each computing iteration (i.e., the batch parameter is 32 similar to the example in Fig. 5).

Fig. 16 shows the performance comparison between different methods to support multicasting (i.e., the conventional unicast [26], the conventional multicast [27], and the Hamiltonian multicast routing [21]). The unicast routing [26] is based on the XY routing algorithm while the conventional multicast routing in [27] extends the XY routing to an XY-based multicast routing. In the following experiments, we adopt the Giga operations (i.e., the multiply-accumulate operations) per

TABLE II
THE ANALYSIS OF THE DESIGN TRADE-OFF ACCORDING TO THE DIFFERENT DESIGN PARAMETERS

	PE size	NoC size	Area (um^2)				Power (mW)				Throughput (GOPS)	HE_score ($\times 10^{-6}$)
			Global buffer	PE	Router	Total	Global buffer	PE	Router	Total		
LetNet [12]	**1**	**6x6**	45,454	2,639,877	1,247,071	3,932,402	1	116	76	193	9.59	2.44
	2	**4x4**	45,454	1,914,914	558,587	2,518,955	1	85	34	120	9.17	3.64
	4	**3x3**	45,454	1,911,478	317,618	2,274,550	1	85	19	105	9.25	**4.07**
AlexNet [13]	**32**	**6x6**	7,408,983	18,010,344	1,247,071	26,666,398	185	1,580	76	1,841	55.00	2.06
	64	**4x4**	7,408,983	8,464,522	558,587	16,432,092	185	1,185	76	1,446	54.65	3.33
	128	**3x3**	7,408,983	5,278,709	317,618	13,005,309	185	1,210	34	1,429	54.22	**4.17**
ResNet-18 [14]	**32**	**6x6**	4,409,027	9,522,587	1,247,071	15,178,685	110	1,333	76	1,519	68.83	4.54
	64	**4x4**	4,409,027	4,692,185	558,587	9,659,799	110	1,076	34	1,220	72.80	7.54
	128	**3x3**	4,409,027	3,156,769	317,618	7,883,414	110	1,149	19	1,278	75.85	**9.62**
MobileNet v1 [15]	**32**	**6x6**	1,499,978	5,278,709	1,247,071	8,025,758	37	1,210	76	1,323	34.96	4.36
	64	**4x4**	1,499,978	2,806,017	558,587	4,864,583	37	1,118	76	1,231	32.58	6.70
	128	**3x3**	1,499,978	2,095,800	317,618	3,913,396	37	1,118	19	1,174	31.44	**8.03**

Note: the HE_score means the hardware efficiency score, and the bold value means the design with better design parameters in the current design space.

second (GOPS) as the metric of the throughput comparison. Compared with a conventional unicast approach, the Hamiltonian multicast routing algorithm can improve throughput from 7% to 49%. The reason is that the Hamiltonian multicast routing algorithm uses a smaller packet size to transmit the data, which mitigates the traffic load on the network. On the other hand, throughput drops by 8% in our proposed approach for the LeNet model. The reason is that the LeNet model does not require too many packets to complete the model computations, which makes the traffic load on the *DNNoC* very light. Hence, the advantage of the Hamiltonian routing algorithm, which is not a minimal routing algorithm, may be counteracted. When the target DNN models (e.g., AlexNet, ResNet-18, and MobileNet v1) become large, the Hamiltonian multicast routing algorithm could mitigate the traffic load on *DNNoC* and improve the throughput by 5% to 79%. Obviously, the traffic latency dominates the performance of the *DNNoC*. Therefore, we employ the run-length encoding method to compress the data in a packet, which helps to reduce the traffic load on the NoC platform. Fig. 17 shows the advantage of the packet compression method. Compared with the method without any compression technique, it can further reduce the latency by 7% to 32% and improve throughput by 6% to 34%. The reason is that the run-length encoding method can use smaller packet size to transmit more information on *DNNoC*.

B. Analysis of Design Trade-off and Hardware Overhead

As mentioned before, the NoC size as well as the $NeuLego_{PE}$ size affect the throughput and the hardware cost. To analyze the design trade-off, we implement the corresponding *DNNoC* designs under TSMC 40nm process for different target DNN models. The 16-bit data precision is adopted in this work.

TABLE II shows the implementation results of the proposed *DNNoC* design methodology. Obviously, the hardware overhead is large if $NeuLego_{PEs}$ include fewer $NeuLego_{blks}$ (i.e., smaller $NeuLego_{PE}$ size). However, for most layer-by-layer DNN models (e.g., LeNet, AlexNet, and MobileNet v1), the throughput can be improved due to higher parallelization. On the contrary, the throughput will be degraded when $NeuLego_{PE}$ size becomes larger. The reason is that the bigger $NeuLego_{PE}$ generates larger packets including more data. When $NeuLego_{PEs}$ receive these large packets, they need to spend more time to perform depacketization. Because the process of the depacketization to extract data is performed in sequence, the $NeuLego_{PE}$ will receives the data in sequence, which reduces the benefit of parallel computing using the NoC interconnection. Consequently, the computing latency becomes longer, which reduces throughput accordingly. On the other hand, the *DNNoC* throughput for the DNN model with cross-layer dataflow, such as ResNet-18, is better when the $NeuLego_{PE}$ size is large. Because of the cross-layer dataflow in ResNet-18, the packet transmission latency increases when using the layer-wise mapping strategy. Therefore, the *DNNoC* with small-sized $NeuLego_{PE}$ may cause heavy traffic load on *DNNoC*, which reduces the throughput significantly.

To consider the throughput and the hardware cost simultaneously, we define a hardware efficiency score (HE_score) as

$$HE\_score = \frac{Throughput}{area}, \tag{9}$$

As shown in Table II, we can find proper design parameters for different DNN models in the current design space (i.e., the bold HE_score). Based on these design parameters, we compare the proposed *DNNoC* with state-of-the-art, as shown in TABLE III. For a fair comparison, the involved clock frequency and bit precision are set to 200 MHz and 16-bit, respectively. On the other hand, we adopt ARM Cortex A9 processor to assist with the neuron operations in Eyeriss [3] and Eyeriss v2 [5] because they only support the neuron operations in convolution and dense layers. As mentioned before, the conventional design methodologies focus on some specific operations and lack design flexibility. For the conventional DNN models (e.g., AlexNet) with common neuron operations (e.g., convolution, max pooling, ReLU, and multiply-accumulation), the overall throughput of Eyeriss [3] and Eyeriss v2 [5] is low. The reason is that Eyeriss [3] and Eyeriss v2 [5] employ the characteristic of data reuse to accelerate the convolution operations and involved software to assist with other kinds of neuron operations. Hence, the overall throughput is worse than other approaches (i.e., UNPU and the proposed *DNNoC*), which are adopting hardware to

TABLE III
THE PERFORMANCE COMPARISON BETWEEN THE RELATED WORKS AND OUR PROPOSED APPROACH

	Eyeriss [3]		**Eyeriss v2 [5]**		**UNPU [19]**		**Proposed *DNNoC***	
Technology	65nm		65nm		65nm		40nm	40nm
Area (Gate count)	1,176k gates		2,695k gates		13,680k gates		3,371k gates	1,012k gates
On-chip SRAM	181.5 KB		246 KB		256 KB		3,760 KB	816 KB
Frequency	200 MHz		200 MHz		200 MHz		200 MHz	
Bit precision	16-bit		16-bit		16-bit		16-bit	
DNN model	**AlexNet**	**MobileNet v1**	**AlexNet**	**MobileNet v1**	**AlexNet**	**MobileNet v1**	**AlexNet**	**MobileNet v1**
Inference/sec (Overall)	0.92	0.69	43.19	1.63	74.35	0.71	53.69	32.79

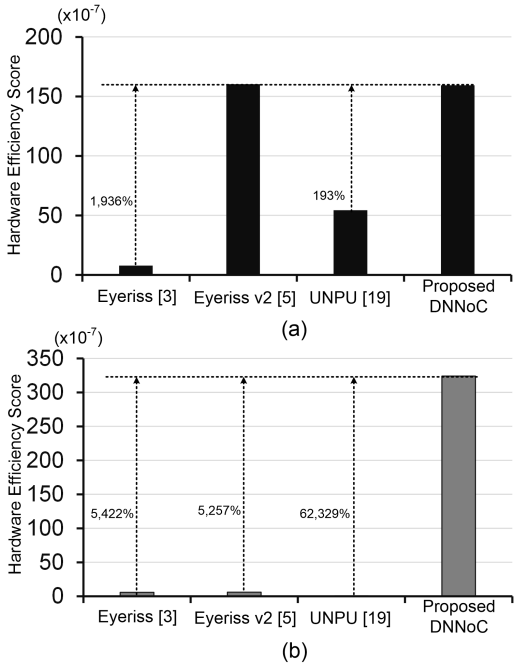

Fig. 18. The comparison of hardware efficiency between the proposed approach and the related works.

compute all kinds of neuron operations. The overall throughput of UNPU is the best because it flattens each kind of DNN operations into similar multiply-accumulation operations. Then, the UNPU takes advantage of lookup tables to accelerate the computations. However, the area overhead of UNPU is significant because of employing a large lookup table in each PE. On the other hand, the current design methodologies [3], [5], [19] cannot provide an efficient design flow to implement the contemporary DNN model (e.g., MobileNet v1) because of the low design flexibility. Therefore, the proposed *DNNoC* design methodology brings the advantage of better throughput than the related works.

To analyze the comparison of hardware efficiency between the proposed *DNNoC* and the related works, we use HE_score in Equation (9) to evaluate each design according to different DNN models in TABLE III. Fig. 18 shows that the proposed design methodology helps to improve the hardware efficiency by 193% to 1,936% over the related works under the AlexNet model. The results of hardware efficiency of our proposed approach and Eyeriss v2 are almost the same because the *DNNoC* includes a larger global buffer (i.e., on-chip SRAM), which counteracts the advantage of the performance improvement. As mentioned before, the global buffer size depends on the largest number of parameters (i.e., weights) in a single neuron layer of the target DNN model, which is usually the dense layer. For a modern DNN model with fewer parameters, such as MobileNet v1, the advantage of the proposed design methodology is more significant. As shown in Fig. 18, the proposed approach can improve the hardware efficiency by 5,257% to 62,329% under MobileNet v1 model over the other related works [3], [5], [19].

VI. CONCLUSION

In this paper, a novel Lego-based Deep Neural Network on Chip (*DNNoC*) design methodology is introduced. First, we define some common neuron computing units, called $NeuLego_{blks}$. To increase the hardware reutilization, we grouped several identical $NeuLego_{blks}$ to build a $NeuLego_{PE}$. We analyzed the target DNN model and extracted the number of $NeuLego_{PEs}$, needed in the platform. Then, we used the NoC interconnection to interconnect the involved $NeuLego_{PEs}$. To reduce the traffic load on the *DNNoC* platform, we further employed a packet compression method and a multicast routing algorithm. The experimental results confirmed the fact that the proposed design methodology could improve the average throughput by 2,802% and average hardware efficiency by 12,523%. Consequently, the proposed *DNNoC* design methodology has the benefits of high scalability and compatibility to be used for the design of future DNN accelerators.

ACKNOWLEDGMENT

This work was supported by the Ministry of Science and Technology under the grant MOST 110-2221-E-110-026-MY3, TAIWAN; MOST 110-2218-E-110 -009, TAIWAN; and the STINT and VR projects, SWEDEN.

REFERENCES

[1] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[2] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15, 2015, p. 161–170.

[3] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[4] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "Dlau: A scalable deep learning accelerator unit on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 513–517, 2017.

[5] Y. Chen, T. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE J. Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[6] K.-C. Chen, Y.-W. Huang, G.-M. Liu, J.-W. Liang, Y.-C. Yang, and Y.-H. Liao, "A hierarchical k-means-assisted scenario-aware reconfigurable convolutional neural network," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 1, pp. 176–188, 2021.

[7] S. M. Nabavinejad, M. Baharloo, K.-C. Chen, M. Palesi, T. Kogel, and M. Ebrahimi, "An overview of efficient interconnection networks for deep neural network accelerators," *IEEE J. Emerging and Selected Topics in Circuits and Systems*, vol. 10, no. 3, pp. 268–282, 2020.

[8] H. Zheng, K. Wang, and A. Louri, "Adapt-noc: A flexible network-on-chip design for heterogeneous manycore architectures," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 723–735.

[9] K.-C. J. Chen, M. Ebrahimi, T.-Y. Wang, and Y.-C. Yang, "Noc-based dnn accelerator: A future design paradigm," in *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip*, 2019.

[10] X. Liu, W. Wen, X. Qian, H. Li, and Y. Chen, "Neu-noc: A high-efficient interconnection network for accelerated neuromorphic systems," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018, pp. 141–146.

[11] H. Kwon, A. Samajdar, and T. Krishna, "Rethinking nocs for spatial neural network accelerators," in *2017 Eleventh IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2017, pp. 1–8.

[12] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in neural information processing systems*, 1990, pp. 396–404.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, 2012, pp. 1097–1105.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[15] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017.

[16] J. Wang, Y. Huang, M. Ebrahimi, L. Huang, Q. Li, A. Jantsch, and G. Li, "Visualnoc: A visualization and evaluation environment for simulation and mapping," in *Proceedings of the Third ACM International Workshop on Many-Core Embedded Systems*, 2016, p. 18–25.

[17] S. Y. H. Mirmahaleh and A. M. Rahmani, "Dnn pruning and mapping on noc-based communication infrastructure," *Microelectronics Journal*, vol. 94, p. 104655, 2019.

[18] M. F. Reza and P. Ampadu, "Energy-efficient and high-performance noc architecture and mapping solution for deep neural networks," in *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip*, 2019.

[19] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H. Yoo, "Unpu: An energy-efficient deep neural network accelerator with fully variable weight bit precision," *IEEE J. Solid-State Circuits*, vol. 54, no. 1, pp. 173–185, Jan 2019.

[20] K.-C. Chen and Y.-C. Yang, "An arbitrary kernel-size applicable noc-based dnn processor design with hybrid data reuse," in *IEEE International Midwest Symposium on Circuits and Systems*, 2021, pp. 1–4.

[21] X. Lin, P. McKinley, and L. Ni, "Deadlock-free multicast wormhole routing in 2-d mesh multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 793–804, 1994.

[22] A. Birajdar, H. Agarwal, M. Bolia, and V. Gupte, "Image compression using run length encoding and its optimisation," in *2019 Global Conference for Advancement in Technology (GCAT)*, 2019, pp. 1–6.

[23] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[25] K.-C. (Jimmy) Chen, M. Ebrahimi, T.-Y. Wang, Y.-C. Yang, and Y.-H. Liao, "A noc-based simulator for design and evaluation of deep neural networks," *Microprocessors and Microsystems*, vol. 77, p. 103145, 2020.

[26] W. Zhang, L. Hou, J. Wang, S. Geng, and W. Wu, "Comparison research between xy and odd-even routing algorithm of a 2-dimension 3x3 mesh topology network-on-chip," in *2009 WRI Global Congress on Intelligent Systems*, vol. 3, 2009, pp. 329–333.

[27] Z. Lu, B. Yin, and A. Jantsch, "Connection-oriented multicasting in wormhole-switched networks on chip," in *IEEE Computer Society Annual Symp. Emerging VLSI Technologies and Architectures*, 2006.

Kun-Chih (Jimmy) Chen (IEEE S'10-M'14-SM'21) currently is an Associate Professor of Computer Science and Engineering Department of National Sun Yat-Sen University. His research interests include Multiprocessor SoC (MPSoC) design, Neural network learning algorithm design, Reliable system design, and VLSI/CAD design. Dr. Chen served as Technical Program Committee (TPC) Chair and General Chair of the International Workshop on Network on Chip Architectures (NoCArc) in 2018 and 2019. Besides, he was a Guest Editor of IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS), Journal of Systems Architecture (JSA), and Nano Communication Network (NanoComNet). Dr. Chen received the IEEE Tainan Section Best Young Professional Member Award, TCUS Young Scholar Innovation Award, Exploration Research Award of Pan Wen Yuan Foundation, the Best Paper Award of VLSI-DAT and the Best Student Paper Award of IEEE ISCAS. He is an IEEE senior member and ACM member.

Cheng-Kang Tsai received the B.S. degree in information and computer engineering from Chung Yuan Christian University, Taoyuan, Taiwan, in 2019. He is currently working toward the M.S. degree at the Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan. His research fields interest in the Network-on-Chip (NoC) system design and neural network accelerator design.

Yi-Sheng Liao received his B.S. degree from Tunghai University, Taiwan, in Electrical Engineering in 2019. Currently, he is pursuing his M.S. degree at Department of Computer Science and Engineering, National Sun Yat sen University, Taiwan. His research fields interest in the neural network accelerator design and network on chip system design.

Han-Bo Xu is a master student studying embedded systems at KTH Royal Institute of Technology. Currently engaged in the research of deep neural network application in network on chip.

Masoumeh (Azin) Ebrahimi received a Ph.D. degree with honors from University of Turku, Finland in 2013 and MBA jointly from the University of Turku and EIT-ICT School in 2015. She has led several national and international projects such as EU-MarieCurie-Vinnova, Academy of Finland, and Vetenskapsrådet (VR), STINT, and SSF. She is currently an associate professor at KTH Royal Institute of Technology, Sweden and an Adjunct Professor at the University of Turku, Finland. Her scientific work contains more than 100 publications, including journal articles, conference papers, book chapters, edited proceedings, and edited special issue of journal. She actively acts as a guest editor, organizer, and program chair in different venues and conferences. Her main areas of interest include interconnection networks and neural network accelerators. She is a member of the European Network on High Performance and Embedded Architecture and Compilation (HiPEAC) and IEEE Senior Member.