Microprocessors and Microsystems 77 (2020) 103145

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

A NoC-based simulator for design and evaluation of deep neural networks

Kun-Chih (Jimmy) Chen[a,*], Masoumeh Ebrahimi[b], Ting-Yi Wang[a], Yuch-Chi Yang[a], Yuan-Hao Liao[a]

[a] Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan
[b] KTH Royal Institute of Technology, Stockholm, Sweden

ARTICLE INFO

Article history:
Received 5 March 2020
Accepted 20 May 2020
Available online 3 June 2020

Keywords:
Network-on-Chip
Neural network
NoC-Based neural network
Artificial neural network
Off-chip memory accesses,

ABSTRACT

The astonishing development in the field of artificial neural networks (ANN) has brought significant advancement in many application domains, such as pattern recognition, image classification, and computer vision. ANN imitates neuron behaviors and makes a decision or prediction by learning patterns and features from the given data set. To reach higher accuracies, neural networks are getting deeper, and consequently, the computation and storage demands on hardware platforms are steadily increasing. In addition, the massive data communication among neurons makes the interconnection more complex and challenging. To overcome these challenges, ASIC-based DNN accelerators are being designed which usually incorporate customized processing elements, fixed interconnection, and large off-chip memory storage. As a result, DNN computation involves large memory accesses due to frequent load/off-loading data, which significantly increases the energy consumption and latency. Also, the rigid architecture and interconnection among processing elements limit the efficiency of the platform to specific applications. In recent years, Network-on-Chip-based (NoC-based) DNN becomes an emerging design paradigm because the NoC interconnection can help to reduce the off-chip memory accesses while offers better scalability and flexibility. To evaluate the NoC-based DNN in the early design stage, we introduce a cycle-accurate NoC-based DNN simulator, called DNNoC-sim. To support various operations such as convolution and pooling in the modern DNN models, we first propose a DNN flattening technique to convert diverse DNN operation into MAC-like operations. In addition, we propose a DNN slicing method to evaluate the large-scale DNN models on a resource-constraint NoC platform. The evaluation results show a significant reduction in the off-chip memory accesses compared to the state-of-the-art DNN model. We also analyze the performance and discuss the trade-off between different design parameters.

1. Introduction

The notable benefits of artificial neural networks (ANNs) have led to the advancement in many real-world applications, such as speech recognition and image classification [1]. The accuracy in several of these applications has reached those of humans. ANN is composed of a large number of neurons which are arranged in layers, called input layer, hidden layers, and output layer. A neuron performs a simple multiply-accumulation (MAC) operation, and it is connected to all/part of the neurons in the next layer. Through these connections, the outputs of one layer become the inputs of the next layer, until the result is obtained in the output layer. To reach high accuracies in more complex applications, neural networks must get deeper, so-called Deep Neural Networks (DNN). On the other hand, the massive deployment of DNN-enabled applications on edge devices (*e.g.*, mobile devices) depends on powerful hardware platforms to execute extensive DNN operations. Current large-scale DNNs, however, involve complex communication, extensive computations, and storage requirements which are beyond the capability of current resource-constraint embedded devices. This has led to recent growing popularity on developing resource-constraint platforms for DNN computation [1].

Currently, the most common hardware platforms to execute DNN operations are CPU, GPU, FPGA, and ASIC. Advanced CPUs (such as 48-core Qualcomm Centriq 2400 [2] and 72-core Intel Xenon Phi [3]) with an impressive many-core computing power has enabled a faster execution of DNN operations. GPUs are another popular general-purpose computing platforms for DNN computation. This is because their intrinsic parallel computing features

* Corresponding author.
E-mail address: kcchen@mail.cse.nsysu.edu.tw (K.-C. (Jimmy) Chen).

https://doi.org/10.1016/j.micpro.2020.103145
0141-9331/

are well-matched with parallel operations in DNN computing. Although CPUs and GPUs provide general-purpose and high parallel computing capability to speed up the DNN computation, a general concern is that the offered high performance comes at the cost of high power consumption [4]. ASIC-based designs can offer higher throughput and power efficiency while suffering from lower flexibility and higher design cost. To exploit parallelism, the ASIC-based DNN designs are usually composed of many DNN processing elements (PEs), which are arranged in a 2D array [5]. FPGAs, based on programmable attributes, are also popular choices for running DNN computation [6]. They can reduce the design time and power consumption while enabling a fast prototype of the DNN accelerator. Compared with the ASIC-based designs, the FPGA-based designs are more flexible in the sense they are re-programmable. However, FPGAs usually come with limited logic and storage resources, which restricts the advantage of reconfigurability. In addition, to map a DNN model to the FPGA platform, it is necessary to handle the PE design as well as the data communication design, which makes the FPGA-based design still suffer from low design flexibility. Both ASIC-based or FPGA-based designs tend to optimize the DNN model for a particular DNN application (*e.g.*, pattern recognition). In other words, the efficiency of the optimized models drops significantly when running a different application on the same platform. In sum, despite the benefit of high power efficiency in the ASIC-based and FPGA-based designs, they suffer from low computational flexibility as they are configured for a specific DNN model or application.

One solution to improve the computational flexibility of DNN accelerators is to decouple the data communication and computation, and that would be possible through employing Network-on-Chip (NoC) interconnection. NoC is a modular packet-switched network, which enables a large number of PEs to communicate with each other through links and routers. NoC is proven as an efficient way to manage complex interconnection in many-core systems and thus to achieve high energy efficiency and performance [7]. The NoC-based design also comes with scalability, reliability, and parallelism features, which makes it even more attractive in DNN computation. Several NoC-based DNN accelerators have been presented so far [8–13]. Several aspects have to be considered when integrating NoC into DNN accelerators, which are the mapping algorithm, topology, and routing algorithm. The mapping algorithms decide how neurons should be clustered and mapped to the processing elements. Topology decides the number and location of routers in the platform and how PEs are interconnected. Routing algorithm defines the routes that data transfers between PEs. Thereby, based on a specific mapping approach, neurons are mapped to PEs where the neuron computation takes place, then the results are sent to other PEs following the involved routing algorithm. Since NoC-based designs decouple communication and computation, it is not necessary to create a specific dataflow to adapt the target DNN model, which substantially increases the computational flexibility. Due to these features, unlike other ASIC-based and FPGA-based platforms, NoC-based DNN accelerators can run different workloads while keeping high performance and power efficiency.

Fig. 1 summarizes the aforementioned platforms for DNN computation. CPUs and GPUs are general-purpose platforms which offer very high reconfigurability at runtime, and thus they support various DNN applications. However, they suffer from huge power consumption and high data transference latency between PEs and off-chip memory. In the contrary, ASIC-based and FPGA-based designs can be optimized for a specific DNN model, and thus they could reach an optimal power efficiency. In return, such optimization limits the platform reconfigurability, and as a result, a single platform is unable to support various DNN models efficiently. NoC-based DNN accelerators, on the other hand, are reconfigurable at runtime and have the benefit of higher power efficiency. Thereby, diverse DNN applications can be easily mapped and executed on the platform.

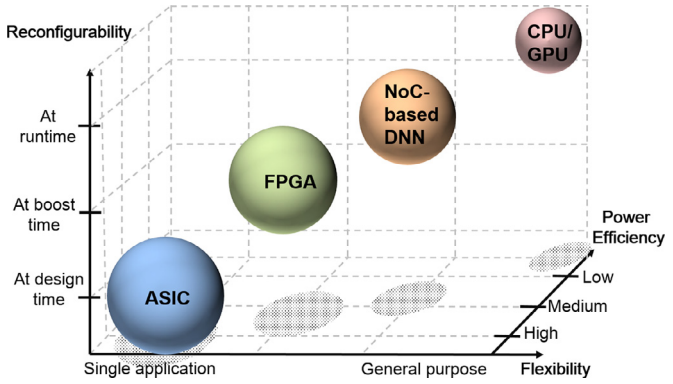

Fig. 1. Attributes of different DNN accelerator design paradigms.

As mentioned before, NoC is a proper platform to provide flexibility and adaptability to support various types of neural network model on a single chip [9,14,15]. To leverage the NoC-based DNN platform, it is necessary to develop a NoC-based hardware simulator. The simulator should support not only the neural network operations (*e.g.*, MAC and pooling) but also the NoC operations (*e.g.*, mapping and routing). In addition, the platform should provide hardware analysis such as power, throughput, transmission latency, as well as the software result (*e.g.*, classification precision). A NoC-based neural network simulator, called NN-Noxim, has been proposed by Chen et al. in [16]. This simulator reports classification precision, power consumption, and transmission latency based on the given DNN model and the NoC primary parameters (*e.g.*, NoC size, XY routing algorithm, and dimension-order mapping). However, this simulator can only simulate the neuron operation and data communication in the fully-connected layers. Other conventional DNN models that are composed of several convolution layers, pooling layers, and fully-connected layers, are not supported by this simulator. This simulator has been extended in [17] to support modern DNN models and various DNN operations (*e.g.*, convolution and pooling). However, this simulator assumes that the target DNN model should be small enough to fit the total PE computing capacity that is offered by the platform. In other words, the model should be mapped to the NoC platform at once, which is not practically viable. In addition, both mentioned simulators in [16] and [17] only support the mesh topology.

To simulate the large-scale DNN, in this paper we propose a cycle-accurate high-level NoC-based DNN simulator, called DNNoC-Sim. The simulation flow is shown in Fig. 2. As shown in this figure, first, the target DNN model is flattened into a MAC-like DNN model. For this purpose, the convolutional and pooling operations are converted into simple multiply-accumulate (MAC) operations. Afterward, the *Flattened DNN* is mapped to the generated NoC platform based on the selected NoC design parameters. As was stated, previous simulators are unable to map a large-scale DNN to the target NoC platform at once. To solve this problem, we propose a dynamic DNN slicing and mapping algorithm that performs the mapping task (a large-scale DNN into a small-sized NoC platform) in several mapping iterations. When the partial DNN model is mapped to the target NoC platform, DNNoC-Sim starts the DNN simulation until the whole DNN is simulated. The contributions of this paper are summarized as follows:

1. We have provided a pre-processing flow, which includes an algorithm to flatten various DNN operations into a MAC-like operation and a slicing algorithm to map the target model into the NoC-based platform.

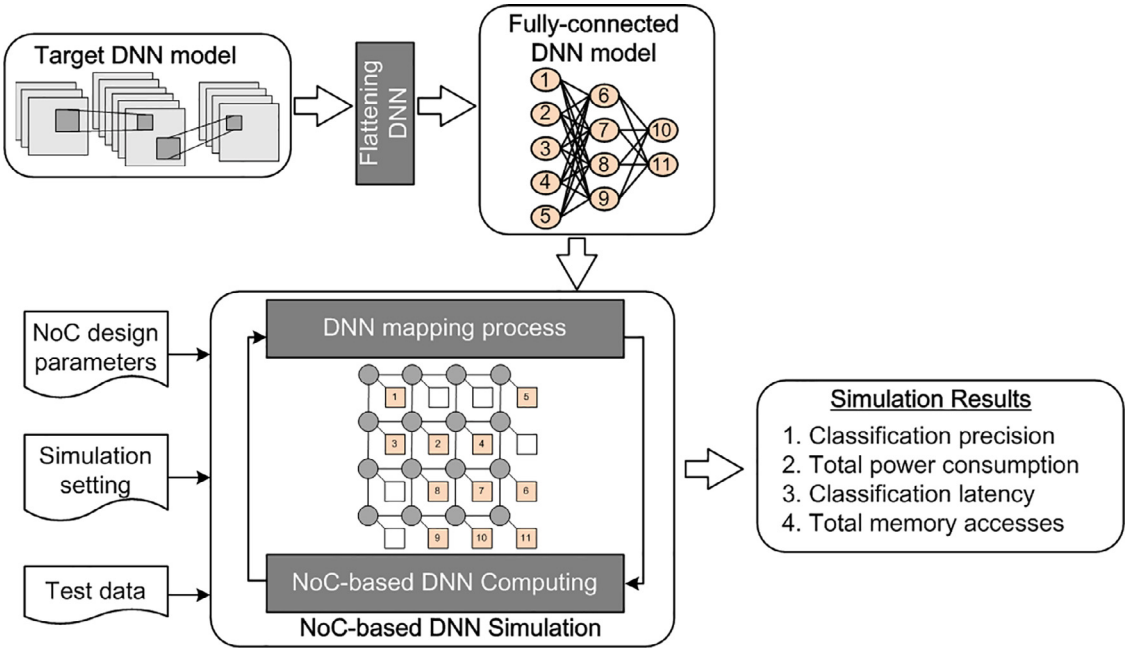

Fig. 2. The simulation flow of the proposed NoC-based DNN simulator, called DNNoC-Sim.

2. We extend a cycle-accurate high-level NoC simulator to support different types of DNN models. This simulator provides accurate hardware results, such as power consumption, throughput, and latency. The simulator reports the classification precision and verifies the result by Keras framework [18]. Using this tool, performance and memory access latency are also evaluated.
3. We make an extensive comparison between conventional and NoC-based DNN accelerators. We also analyze the trade-off between different design parameters in the NoC-based DNN platform.

The remainder of this paper is organized as follows. Section II reviews the background and analyzes current popular neural network simulators. The proposed NoC-based DNN simulator, DNNoC-Sim, which supports the proposed DNN flattening technique and DNN slicing method, are introduced in Section III. In Section IV, we compare different configurations and analyze the experimental results. Finally, Section V concludes this paper.

2. Background and related works

DNNs have been widely used to solve many complex real-world problems and achieve stunning performance in solving these problems. To execute the DNN models on edge devices, efficient resource-constraint hardware platforms are needed. NoC-based designs not only reduce the design complexity of the large-scale DNN hardware implementation but also provide high-performance and low-latency communication for DNN applications. In this section, we first investigate the conventional DNN simulators and the tools for the NoC simulation. Afterward, the NoC-based DNN simulators will be discussed.

2.1. Conventional neural network simulators

Deep Playground [19] is a neural network simulator based on the Tensorflow neural network framework. The simulator visualizes the operation of the neural network and the training process. Using this simulator, users can train the neural network at real-time, observe the training process, and get insight on how the training process works. The Deep Playground simulator, on the other hand, only simulates fully-connected neural networks (such as ANN [20]) and does not support convolutional neural networks such as AlexNet and VGG-net. Therefore, the flexibility and scalability of this tool is limited and not sufficient to run modern DNN models. In addition, Deep Playground does not give any hardware analysis report such as power consumption and latency.

NNtool [21] is a MATLAB toolbox to develop neural network models with straightforward commands. There are many neural network models provided in NNtool that can be examined and evaluated. Besides, MATLAB supports various mathematical calculations to analyze the results in detail. Hence, NNtool provides configurable parameters to facilitate the simulation of neural network models with different architectures. However, NNtool is a software-based simulator that does not support hardware behavior simulation. NNtool does not also support the NoC-based neural network model.

2.2. NoC platform simulators

Noxim is a SystemC-based NoC simulator proposed by Cataniz et al. [22]. Noxim became a widely used tool to simulate NoC traffic behavior. Based on the user-defined design parameters, such as NoC size and buffer size, Noxim first generates the corresponding NoC system and determines the overall transmission dataflow. Then, the packet transmission on the generated NoC platform can be simulated. Noxim reports different results such as throughput, latency, and power consumption, which can be leveraged to analyze the NoC design cost at the hardware level. Although Noxim is a helpful simulator to analyze the NoC behavior, it cannot be utilized in DNN computing as it does not support neuron operations in PEs and related clustering and mapping algorithms.

ATLAS [23] is a java-based software simulator to evaluate the NoC system. Based on the user-defined design parameters, ATLAS can perform DC, AC, and transient analysis for silicon, binary, ternary, and quaternary material-based devices. ATLAS not only can provide a comprehensive power analysis but also detailed evaluation results on the NoC transmission behavior. Although ATLAS is a powerful tool for the detailed NoC analysis, it does not support any function for the neural network computation. Consequently, ATLAS

is still not an appropriate simulator to perform the relevant analysis and evaluation of NoC-based neural network hardware designs.

VisualNoC [24] is an open-source cycle-accurate full-system simulator for many-core embedded systems. This visualization simulator supports both network simulation and task mapping and helps to observe the system behavior and identify the system bottlenecks and deadlocks. VisualNoC records all events in the network happening in routers, links, and processing elements, and all these events can be reproduced for further analyses. In addition, the visualization feature of VisualNoC offers an intuitive way of analyzing the efficiency of different routing and mapping algorithms. This simulator, however, does not support DNN computation.

Booksim is an alternative and popular cycle-accurate NoC simulator, which is built in C++ and capable of supporting NoC hardware simulation [25]. Booksim comes with various topologies, routing algorithms, and other design parameters. By using this simulator, designers can optimize the traffic dataflow and analyze the overall system performance. Similar to other mentioned NoC simulators, Booksim does not support the required functions to perform neuron computing.

2.3. NoC-based neural network simulators

NN-Noxim is a nonproprietary cycle-accurate NoC-based neural network hardware simulator [22]. NN-Noxim has extended the functions of Noxim and is the first NoC simulator that simulates the neural network behavior. It is based on the mesh topology with user-defined parameters. After mapping all neurons to PEs, NN-Noxim processes the neural network on the generated NoC platform. Although the neural network operations in the fully-connected layers are implemented in NN-Noxim, it does not still support the common operations in well-known CNN models, such as convolution and pooling.

NN-Noxim has been further extended in CNN-Noxim [17] to implement NoC-based convolutional neural networks. Since CNN-Noxim supports several complicated operations such as convolution and pooling, it can simulate various CNN models on the mesh-based NoC platform. However, this simulator assumes that the target CNN model should be mapped to the NoC platform at once, which depends on the available on-chip resources and memory storage. As a result, CNN-Noxim needs to generate a very large NoC to simulate the large-scale CNN models. Although this approach works for small-sized DNN networks, it is not a practical simulator for current large-scale DNN models. Also, CNN-Noxim is limited to the mesh topology, which lacks design flexibility.

3. Proposed NoC-based DNN simulation platform for DNN model evaluation

In this section, we explain a complete process of executing a DNN network on the NoC-based simulation platform. First, in Section 3.1 we overview the Noxim simulator and then in Section 3.2 we introduce an enhanced simulator, called *DNNoC-Sim* capable of computing DNN operations. Then, we start by flattening the DNN network to reach a *Flattened DNN* model in Section 3.2.1. This step is achieved by converting all operations in the convolution, pooling, and fully-connected layers to the MAC-like operations. Afterward, in Section 3.2.2, we explain a proper clustering approach to cluster the *Flattened DNN* model. Each cluster is then mapped to a PE for execution. In Section 3.2.3, we introduce the dynamic DNN slicing and mapping algorithm to slice and map the DNN model to fit the resource-constraint NoC platform at runtime. Finally, in Section 3.2.4, we explain the computing flow and the control mechanism in a PE.

3.1. Overview of noxim

Noxim [22] is a SystemC-based simulator which supports mesh-based NoC simulation. Fig. 3 shows the transaction-level model (TLM) of Noxim, which describes a 3×3 mesh-based NoC platform. Based on the user-defined design parameters such as the mesh size and the input buffer size, Noxim can generate a top module of the target NoC platform. The top module is composed of several tile modules where the tile modules are connected through channels. The channel behavior depends on the design parameters such as the input buffer size, the number of virtual channels, and the network topology. Based on this hierarchical SystemC module design, Noxim provides detailed information about the power consumption, throughput, and latency, which helps the designers to analyze the NoC system. Although the TLM model in Noxim provides a flexible way to construct NoC in the system level, the simulator is limited to the mesh-based topology and lacks design flexibility.

Each tile module is composed of a *router module* and a *PE module*. The *router module* is used to simulate the behavior of packet delivery. Based on a particular routing algorithm (such as XY routing or west-first routing), the router assigns an appropriate output channel to a packet stored in the input buffer. The switching congestion happens if multiple packets request the same output channel. Usually, the switching congestion can be mitigated or even resolved by using virtual channels, a proper arbitration mechanism, or an advanced routing algorithm that provides higher path diversity for routing packets. Noxim supports many mature routing mechanisms such as X-Y routing, west-first adaptive routing and odd-even adaptive routing.

The *PE module* in Noxim is used to generate packets based on the underlying mapping and routing algorithm. A generated packet is composed of several flits as header, body, and tail. The header flit keeps the packet routing information such as the source and destination addresses and the cycle in which the packet is generated. The tail flit indicates the end of the packet, which leverages packet routing and packet switching. Finally, the body flits store the data that is going to be transferred from the PE. Since Noxim is a network-level simulator, the generated body flits do not carry any specific information and are empty. Thereby, Noxim cannot support any computation, such as those in DNN computing.

3.2. DNNoC-Sim: A cycle-accurate high-level NoC-based DNN simulator

Most of the current simulators support either neural network operations or traffic behavior on the NoC platform. Therefore, they are not proper choices to simulate the NoC-based DNN operations. Although the two recent NoC-based ANN simulators (*i.e.*, NN-Noxim [16] and CNN-Noxim [17]) can be used to execute DNN computing on the NoC platform, they generate a very large NoC size to simulate a large-scale DNN model, which is not practically feasible. Moreover, some operations in recent DNN models, such as convolution and pooling, cannot be efficiently computed. To address these issues, in this work, we propose a cycle-accurate high-level NoC-based DNN simulator, called DNNoC-Sim, which is an extension of Noxim [22]. DNNoC-Sim supports

1. ***DNN Flattening***: To increase the computing flexibility of the NoC platform, various DNN operations in the convolution, pooling, and fully-connected layers are converted into MAC-like operations. After this process, the DNN model is flattened into the *Flattened DNN* model.
2. ***Neuron Clustering***: To facilitate the neuron mapping on the NoC platform and to reduce the NoC traffic, the neurons in *Flattened DNN* are clustered where each cluster is called *big neuron*.

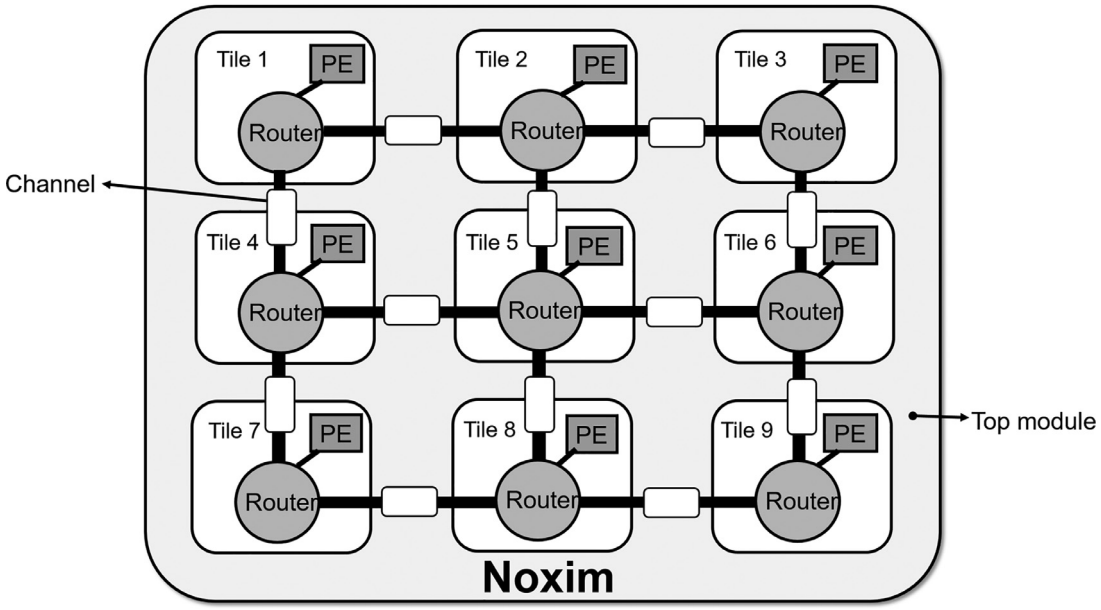

Fig. 3. The transaction-level model (TLM) overview of Noxim.

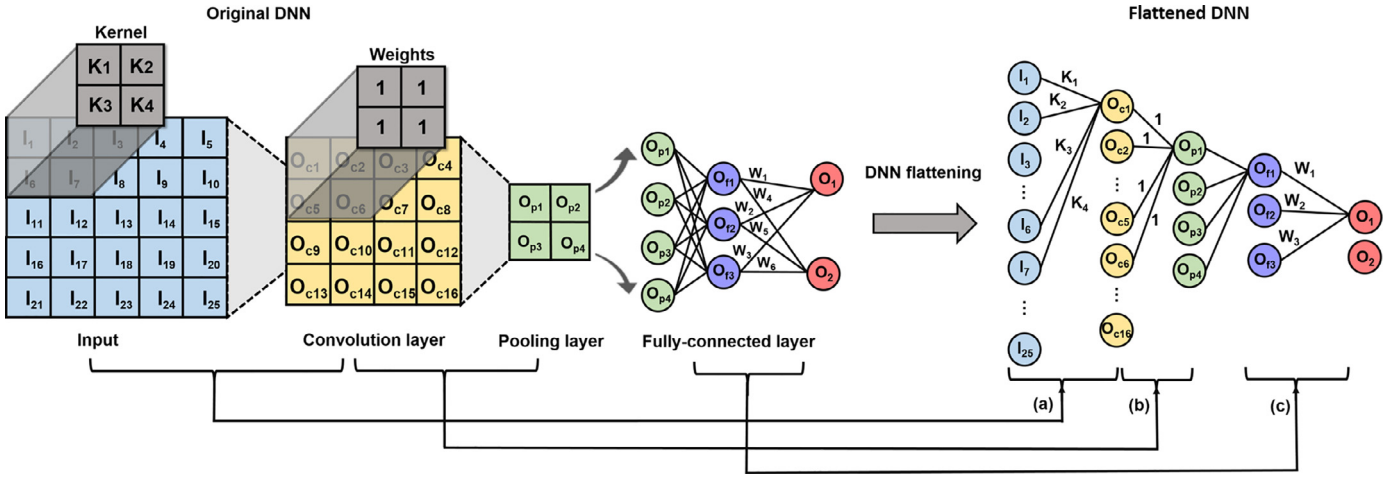

Fig. 4. Flattening the DNN model into *Flattened DNN*; the interconnection of (a) the convolution layer and (b) the pooling layer are converted into a partially-connected layer; (c) the interconnection structure of a fully-connected layer is maintained.

After this process, the *Flattened DNN* will be changed to *Clustered DNN*.

3. ***Dynamic DNN Slicing and Mapping***: To map a large-scale DNN to a small-sized NoC platform, the *Clustered DNN* is dynamically divided into several slices, and then each slice is mapped to the NoC platform within one mapping iteration.
4. ***Generic Neuron Computing and PE Micro-Architecture***: To compute different types of DNN operations in a shared-resource NoC platform, each PE supports generic neuron computing. The computing flow and the control mechanism of a generic PE are presented.

In this section, we introduce the aforementioned novel NoC-based DNN evaluation flow with a realization of the proposed DNNoC-Sim tool.

3.2.1. DNN flattening process

As shown in Fig. 4, the original DNN is usually composed of several convolution layers, pooling layers and fully-connected layers. In the convolution layer, a pre-defined kernel is convoluted with the input to extract some features from the input. The kernel size depends on the image size as well as the user experience. The pooling layer usually accepts the output of the convolution layer and is used to reduce the data dimensionality (*i.e.*, downsize a group of data to a single data). The maximum and average pooling are the most popular pooling methods employed in the conventional DNN models [26]. Finally, in the fully-connected layer, a neuron in one layer is connected to all neurons in the next layer. The fully-connected layer is used to classify its inputs based on the extracted features from the upstream convolution and pooling layers.

Conventional DNN models involve various computing types, such as convolution, pooling, and MAC. To facilitate the execution of different DNN models on the same homogeneous NoC platform, it is necessary to unify the computing types. For this purpose, we investigate the computing behavior of these operations. First, we formulate the two-dimensional convolution operation by:

$$O_{c(i,j)} = \sum_{k=1}^{n} \sum_{l=1}^{n} \left(K_{(m,l)} \times I_{(i+k-1,j+l-1)} \right), \tag{1}$$

where K represents the pre-defined kernel; n represents the target kernel size; and I and O represent the input and output of the convolution layer, respectively. If the convolution is three dimensional, the kernel remains the same while the inputs come from other dimensions, such as the R, G, B dimensions in the application of image recognition. As a general rule, the input should be extended into additional dimensions, and the same operation in Equation (1) should be expanded with identical K.

As illustrated in Fig. 4(a), the two dimensional MAC operation in Equation (1) can be flattened into multiple one dimensional MAC operations. This forms the basic operation of each neuron that can be expressed by:

$$\begin{aligned}
O_{c1} &= K_1 \times I_1 + K_2 \times I_2 + K_3 \times I_6 + K_4 \times I_7,\\
O_{c2} &= K_1 \times I_2 + K_2 \times I_3 + K_3 \times I_7 + K_4 \times I_8,\\
O_{c3} &= K_1 \times I_3 + K_2 \times I_4 + K_3 \times I_8 + K_4 \times I_9,\\
O_{c4} &= K_1 \times I_4 + K_2 \times I_5 + K_3 \times I_9 + K_4 \times I_{10},\\
O_{c5} &= K_1 \times I_6 + K_2 \times I_7 + K_3 \times I_{11} + K_4 \times I_{12},\\
O_{c6} &= K_1 \times I_7 + K_2 \times I_8 + K_3 \times I_{12} + K_4 \times I_{13},\\
O_{c7} &= K_1 \times I_8 + K_2 \times I_9 + K_3 \times I_{13} + K_4 \times I_{14},\\
O_{c8} &= K_1 \times I_9 + K_2 \times I_{10} + K_3 \times I_{14} + K_4 \times I_{15},\\
O_{c9} &= K_1 \times I_{11} + K_2 \times I_{12} + K_3 \times I_{16} + K_4 \times I_{17},\\
O_{c10} &= K_1 \times I_{12} + K_2 \times I_{13} + K_3 \times I_{17} + K_4 \times I_{18},\\
O_{c11} &= K_1 \times I_{13} + K_2 \times I_{14} + K_3 \times I_{18} + K_4 \times I_{19},\\
O_{c12} &= K_1 \times I_{14} + K_2 \times I_{15} + K_3 \times I_{19} + K_4 \times I_{20},\\
O_{c13} &= K_1 \times I_{16} + K_2 \times I_{17} + K_3 \times I_{21} + K_4 \times I_{22},\\
O_{c14} &= K_1 \times I_{17} + K_2 \times I_{18} + K_3 \times I_{22} + K_4 \times I_{23},\\
O_{c15} &= K_1 \times I_{18} + K_2 \times I_{19} + K_3 \times I_{23} + K_4 \times I_{24},\\
O_{c16} &= K_1 \times I_{19} + K_2 \times I_{20} + K_3 \times I_{24} + K_4 \times I_{25}.
\end{aligned} \tag{2}$$

Thereby, to compute the outputs of the convolution layer, 16 neurons (*i.e.*, 64 MAC operations in this example) should be calculated using the same weights (W_1, W_2, W_3, and W_4). This operation partially connects the I layer to the O_c layer, shown in Fig. 4(a)).

For the pooling layer, we utilize the maximum pooling operation as an efficient way to capture the most important features in the sub-sampling operations [27]. The behavior of the maximum pooling operation is to divide the input map into several sub-maps and then capture the maximum of each sub-map as the output. Based on the definition of the maximum pooling, the outputs O_{p1}, O_{p2}, O_{p3}, and O_{p4} can be computed by:

$$\begin{aligned}
O_{p1} &= Max(F_1, F_2, F_5, F_6),\\
O_{p2} &= Max(F_3, F_4, F_7, F_8),\\
O_{p3} &= Max(F_9, F_{10}, F_{13}, F_{14}),\\
O_{p4} &= Max(F_{11}, F_{12}, F_{15}, F_{16}),
\end{aligned} \tag{3}$$

Fig. 4(b) demonstrates the maximum pooling operation. To unify different DNN computations, we need to convert the operations in Eq. (3) to the MAC-like operations. Each pooling layer output can also be represented by:

$$O_{p(r,s)} = P_{k=1}^{m} P_{l=1}^{m} \left(W \times I_{(r+k-1,s+l-1)}\right), \tag{4}$$

where I and O_p represent the input and output of the pooling layer; m represents the pooling size; W is equal to 1; and $P_{j=1}^{N} I_j$ is defined as:

$$P_{j=1}^{N} I_j = argmax(I_j). \tag{5}$$

In this way, the pooling operation can also be expressed with MAC-like operations. By flattening Eq. (4), we reach to 4 neurons to obtain the whole pooling results. As shown in Fig. 4(b), similar to the convolution layer, the F layer and the O_p layer are partially connected.

In the fully-connected layer, each neuron in one layer is fully-connected to all neurons in the next layer. Each partial output can be calculated by the matrix multiplication. For example, as shown in Fig. 4(c), the final output of O_1 and O_2 can be presented as:

$$\begin{bmatrix} O_1 \\ O_2 \end{bmatrix} = \begin{bmatrix} W_1 & W_2 & W_3 \\ W_4 & W_5 & W_6 \end{bmatrix} \times \begin{bmatrix} O_{f1} \\ O_{f2} \\ O_{f3} \end{bmatrix}. \tag{6}$$

Besides, the general representation of the operation in the fully-connected layer can be expressed by:

$$O_v = \sum_{k=1}^{n \times v} W_k \times O_{f(k\%(n+1)+(v-1))}. \tag{7}$$

where n is the number of neurons in a layer, and i is the number of outputs after neuron computation in this layer.

The DNN expansion has been described by a flowchart in Fig. 5. The computations in the fully-connected layer are naturally described by MAC-type operations. As was also explained, the computations in the convolution and pooling layers should be converted to MAC-like operations, given in Eqs. (1) and (4), respectively. In this way, different layers in the DNN model can be converted to a fully-connected or partially-connected neural network, that is called *Flattened DNN* in this paper. As it will be explained in Section 3.2.4, the reason for converting the operations to MAC-like operation is to share resources in hardware.

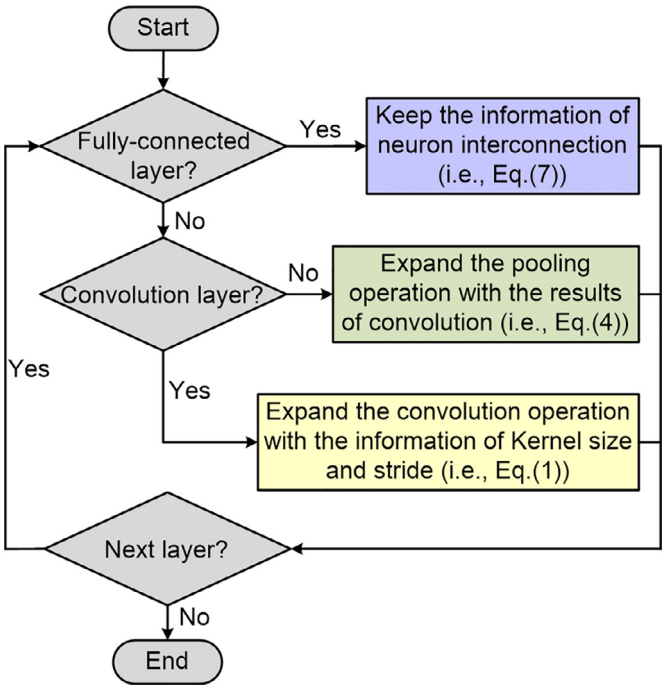

Fig. 5. The flowchart of the DNN flattening process.

3.2.2. Clustering strategy

After a PE completes its execution, the results are packaged and delivered to the next PE. As a DNN network involves thousands to millions of neuron computations, heavy traffic load will be generated if one packet is delivered per single neuron computation. Therefore, to reduce the network traffic load and to improve computational efficiency, the computation of multiple neurons can be assigned to one PE [9,14,15]. For this purpose, the neurons in the *Flattened DNN* model are divided into several groups where each group is called *big neuron*. The maximum number of neurons in a group (*i.e.*, the *big neuron* size) depends on the computing capacity of a PE. The DNN model after this grouping is called *Clustered DNN*.

The clustering strategy directly affects the communication load in the NoC platform. For example, the *Flattened DNN* model in Fig. 6(a) contains 4 layers and 13 neurons. By using the specific

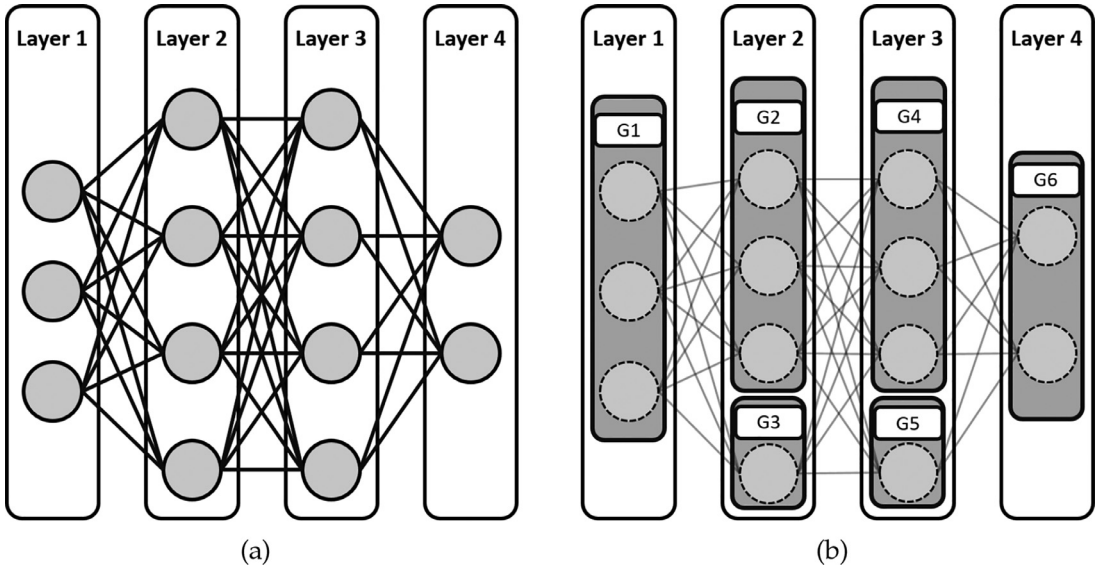

Fig. 6. The (a) original *Flattened DNN* and the *Clustered DNN* by using (b) proposed clustering algorithm.

clustering strategy, the neurons in different layer will be divided into different groups based on the group-size constraint (*i.e.*, the number of neurons in a clustering neuron group,) as shown in Fig. 6(b). Each big neuron is then mapped to one PE where the location of a PE on the NoC platform depends on the underlying mapping strategy. The computing results of a PE should be delivered to other PEs in which the big neuron has connectivity with. In this way, the output of a big neuron becomes the input of several other big neurons in the next layer, which simplifies the computing flow. For example, In Fig. 6(b), the result of G1 should be sent to the corresponding PEs in G2 and G3.

To select a proper clustering strategy, we analyze the characteristics of the common neural network computing flow. Because of the feed-forward computing flow in the common neural network operation, the input of each neuron layer usually reuses the output results (*i.e.*, partial sum) from the previous neuron layer. Therefore, it has been proven that the output reuse computing strategy benefit to the neural network computation [28]. By following this computing behavior, the inter-layer clustering strategy is adopted in this work to leverage the output reuse computing strategy. In this way, the feed-forward computing flow of the DNN model can be reserved, which mitigates the traffic load on the NoC platform.

3.2.3. Dynamic DNN slicing and mapping

After clustering the *Flattened DNN*, each *big neuron* of the *Clustered DNN* should be mapped to the target NoC platform. In recent years, many kinds of large-scale DNN models, such as VGG-16 or AlexNet, are proposed to solve more complex problems and achieve higher accuracy. To map the whole large-scale DNN model to the NoC platform, either the NoC size should be enlarged or the PE computing capacity should be increased. By enlarging the NoC size, more *big neurons* can be mapped to the platform. On the other hand, by increasing the PE computing capacity, each PE can handle larger *big neuron* sizes. Both solutions are not practically feasible when dealing with large-scale DNN models. Thereby, a design challenge is to map a large-scale DNN model to the size-limited NoC platform.

To address the mentioned problem, we propose a DNN slicing method. The minimum requirement of the proposed method is that the largest layer of the *Clustered DNN* (*i.e.*, the layer that contains the largest number of *big neurons*) should fit the NoC platform at once. In this way, we can guarantee that at least one layer

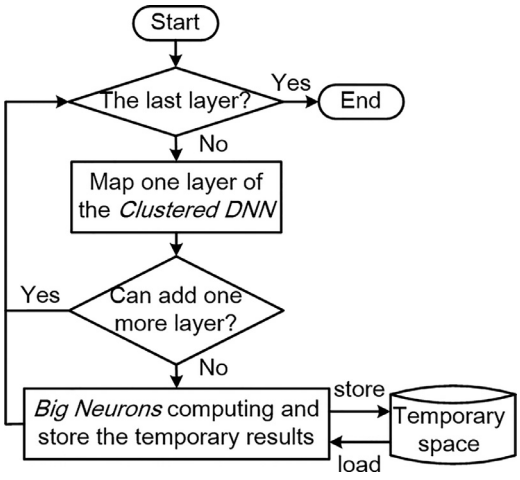

Fig. 7. The flowchart of the proposed Dynamic DNN Slicing method.

can be mapped to the NoC platform at a time. Fig. 7 shows the simulation flow of the proposed dynamic DNN slicing and mapping method. As shown in the flowchart, the proposed slicing method is performed layer-by-layer and time-by-time to ensure that at least one layer is fully mapped to the NoC platform at a time.

Fig. 8 illustrates an example to map the *big neurons* of the *Clustered DNN* to a 2 × 2 mesh-based NoC platform. In this example, we apply the aforementioned clustering strategy to group the neurons in the *Flattened DNN* and generate its corresponding *Clustered DNN*, as shown in Fig. 6(b). Besides, the group size is three, which is aligned with the computing capability of the involved PE (*i.e.*, a PE can support at most three neuron operations). By following the rules of the proposed method, we first slice the *Clustered DNN* layer-wisely, as shown in Fig. 8(a). The slicing decision is made considering the total computing resources on the NoC platform. After slicing the *Clustered DNN*, we map each slice to the NoC platform. As shown in Fig. 8(b), at the first mapping iteration, Slice-1 (*i.e.*, the *big neurons* in the first two layers containing G1, G2, and G3) is mapped to the NoC platform. The PE4 is idle because its computing resources are not enough to support all computations in layer 3. After completing the computations in Slice-1, temporary results will be stored in the off-chip memory. At *Time-2* (*i.e.*, the

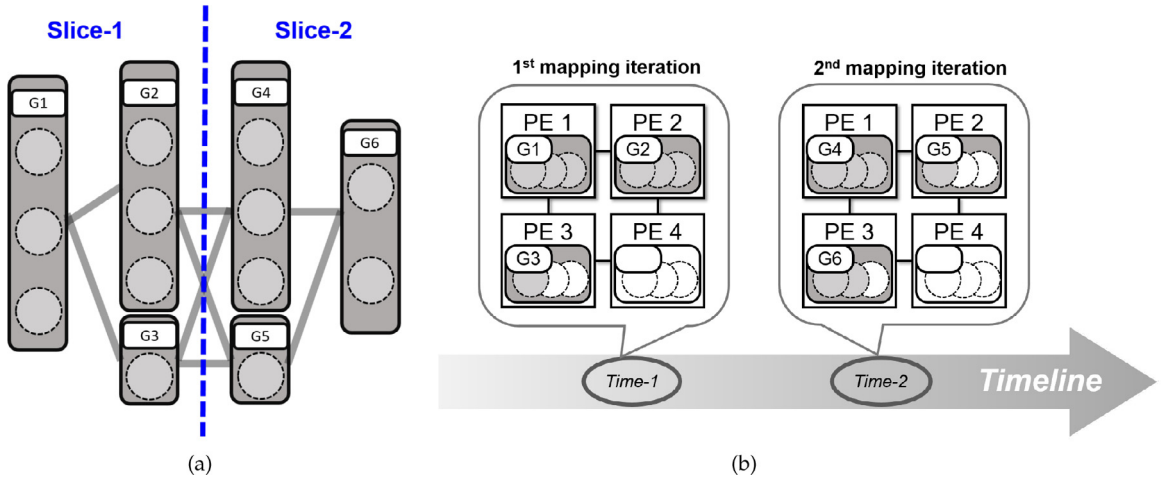

Fig. 8. (a) The *Clustered DNN* is sliced into four slices, and (b) map each slice to the small-sized NoC within two mapping iterations.

second mapping iteration), the *big neurons* in layer 3 and 4 (*i.e.*, G4, G5, and G6) are mapped to the platform. G4 and G5 require the temporary results of G2 and G3, and these results can be read from the off-chip memory. By this approach, a large-scale DNN can be mapped to a small-sized NoC platform.

Along with the dynamic DNN slicing method at runtime, a mapping algorithm should be applied to map each *big neuron* to a PE of the NoC platform. The simplest mapping way is to map each *big neuron* to the NoC platform along with the dimension order, as shown the mapping results in Fig. 8(b). Although this kinds of dimension-order mapping algorithms are easy to realize, the NoC platform may suffer from severe traffic congestion as the adjacent *big neurons* are mapped too densely [16]. To alleviate this problem and to keep simplicity, in this work, we apply the *NN-aware* mapping algorithm [9] where the traffic condition of the NoC-based DNN design can be guarantee along with a proper routing algorithm. Because of the similarity between the mesh and torus topology, all the mapping algorithms for the mesh-based NoC platform can also be applied to the torus-based NoC platform.

3.2.4. Generic PE computing flow and control mechanism

In the proposed DNNoC-Sim, a PE should be able to perform different types of computations in convolution, pooling, and fully-connected layers. Thereby, the PE computing flow should be designed in such a way that to support different computations and to control them at runtime. Fig. 9 shows the computing flow of a PE receiving all the necessary data from the *big neurons* in the previous layer. If the *big neuron* size is *N*, all active PEs perform *N* neuron computations in parallel (*i.e.*, all PEs follow the same computing flow). After processing the data, PEs package the data and deliver it to the next destination for further computing or producing the final output. Note that the destination assignment depends on the underlying mapping algorithm.

As mentioned before, each neuron computation requires specific weights. For example, the weights are equal to one when performing the pooling operation. The weights for the convolution operation depend on the adopted kernel. Finally, the weights for the fully-connected layer are obtained through the DNN training process. As shown in Fig. 10(a), we employ the SystemC model to distinguish different neuron computations and to control the behavior of the PE operation. The weight memory in Fig. 10 is used to store all the involved weights in the current computing iteration of this PE. In case of the DNN mapping with several iterations, the weight memory should be updated after each mapping iteration, and the

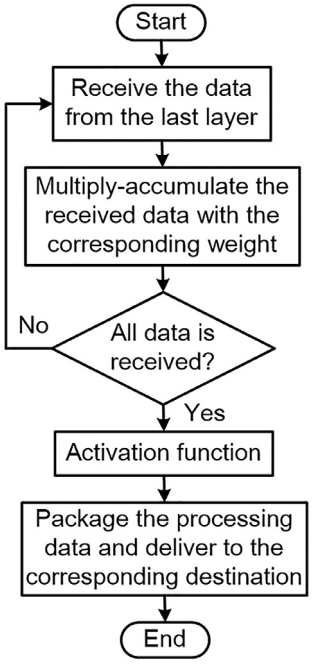

Fig. 9. The computing flow of the PE computing model in the proposed DNNoC-Sim.

proposed DNNoC-Sim also simulates the behavior of the weight memory updating in abstract level. On the other hand, to support different computing flow, the PE is controlled by a FSM control flow, as shown in Fig. 11. The the output control signal from the FSM will further control the involved multiplexers in Fig. 10. Obviously, the FSM control flow is determined by the *Op* and *DataTail* input signals, and the corresponding output control signal is shown in TABLE 1. *Op* indicates the current neuron operation (*i.e.*, convolution (Conv.), pooling (Pool), or fully-connected (FC) operations). On the other hand, the *DataTail* is triggered to one when all input data has been received (*i.e.*, all the necessary partial results from the *big neurons* in the previous layer). When *DataTail* is set to one, the result of the last MAC operation becomes the output of this PE. After applying the activation function, the output becomes the input of the next PE.

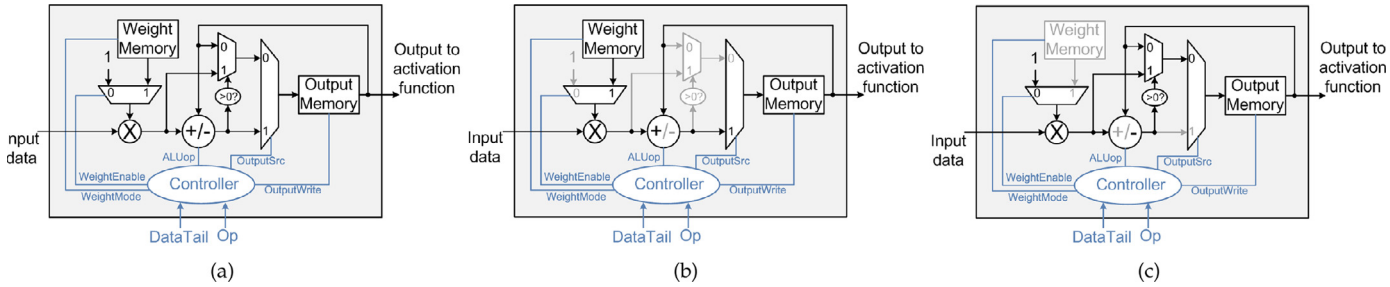

Fig. 10. (a) The PE model and its control signals; (b) data path of the convolution and fully-connected layers; (c) data path of the pooling layer.

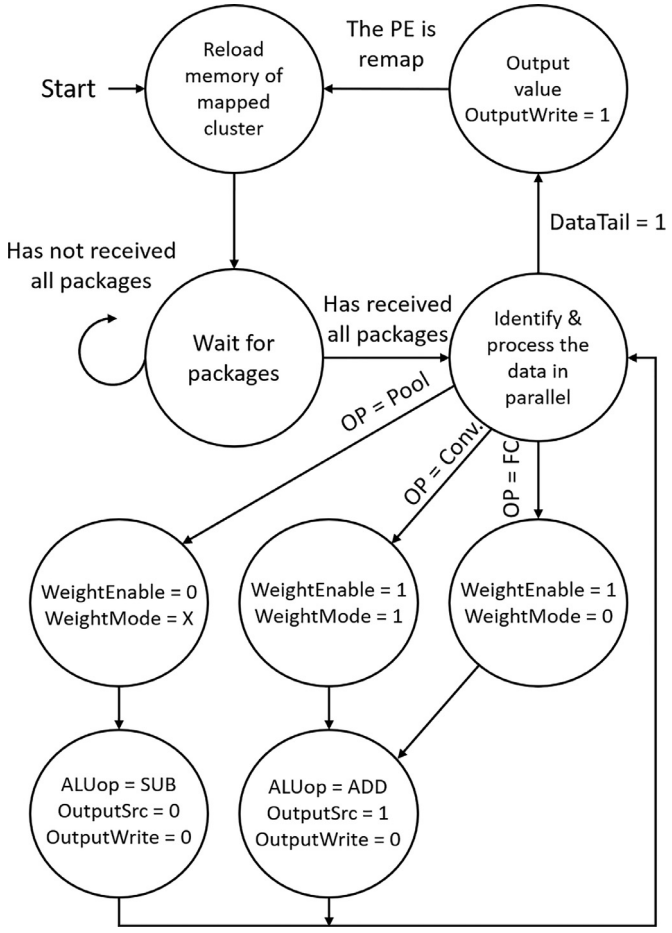

Fig. 11. The finite state machine of the PE computing model in the proposed DNNoC-Sim.

Table 1
The control signal assignment for the PE computing.

	DataTail = 0			*DataTail* = 1		
OP =	*Conv.*	*Pool*	*FC*	*Conv.*	*Pool*	*FC*
WeightEnable	1	0	1	x	x	x
WeightMode	CONV	x	FC	x	x	x
ALUop	ADD	SUB	ADD	x	x	x
OutputSrc	1	0	1	x	x	x
OutputWrite	0	0	0	1	1	1

The *ALUop* signal is used to select between the add or subtraction operation. Since the operations in the convolution and fully-connected layers are MAC-type operations, the *addition* will be selected by the *ALUop* signal. On the other hand, in the pooling operation, the input data should be compared with each other to find the maximum value. The comparison operation can be implemented using substation (*i.e.*, inverse and add operation). Therefore, in the pooling operation, *ALUop* is adjusted to select the *subtraction* operation. The *WeightEnable* and *Weightmode* signals select whether the weight should be 1 or read from weight memory. As was already discussed, weights are set to 1 in the pooling operation while for the operations in the convolution or fully-connected layers, the weights are fetched from weight memory. PE can produce accumulation and subtraction results. The *OutputSrc* signal is used to select which of them should be written into the output memory. Finally, the *OutputWrite* signal is used to latch the computing results until the final result is obtained.

Fig. 10 (b) and (c) illustrate two examples to demonstrate the data paths using the control signals in TABLE 1. As shown in Fig. 10(b), if *Op* is *Conv.* or *FC*, the MAC operations in the convolution or fully-connected layer will be performed. First, the *WeightMode* signal is adjusted to select the corresponding weight in the weight memory. Afterward, the MAC operation is performed by setting the *WeightEnable* to 1 and *ALUop* to ADD. As long as the PE receives the input data, the result of the MAC operation will be latched in the output memory and reused as one of the inputs of the next MAC operation (*i.e., OutputSrc* is set to 1 and *OutputWrite* is set to 0). As shown in Fig. 10(c), if the *Op* is set to Pool, the operation in the pooling layer will be performed (*i.e.*, maximal data determination). In this case, *WeightEnable* will be set to 0, and *WeightMode* becomes a donât-care signal. Since the comparison operation can be implemented using subtraction, the *ALUOp* will be set to SUB, and the temporary result is selected by setting the *OutputSrc* to 0. Finally, the *OutputWrite* will be set to 1 as long as the *DataTail* becomes 1 (*i.e.*, the current input data is the tail of the series input data).

In addition to a reliable flow control, the parameters required for operation also need to be ensured. In other words, the size of the memory that implemented in the PE must be able to accommodate at least larger than the group size. Besides, the accuracy of the application is also one of the factors that affect the memory size. The designer should give the proper memory resources based on the computing requirement after evaluation with the proposed simulator. Follow the FSM in Fig. 11, memory will reload the weight data after all output been transmitted. It is worth mentioning that all PEs in the NoC operate independently, which means that the weight update time of each PE is not synchronized. Staggering time of on-off-chip memory transmission and minimizing the use of transmission bandwidth is another benefit of NoC systems.

4. Evaluation results and analysis

4.1. Simulation setup

To validate the proposed DNNoC-Sim, we simulate small, medium, and large-scale DNN models, which are LeNet [29], Mo-

Table 2
Comparison between conventional and NoC-based designs regarding the number of off-chip memory accesses.

	Conventional [10]	NoC-based Design
LeNet [29]	884,736	47,938 (-94.6%)
MobileNet [30]	1,061,047,296	4,360,616 (-99.6%)
VGG-16 [31]	1,165,128,192	138,508,072 (-88.1%)

bileNet [30], and VGG-16 [31], respectively. Besides, the NN-aware mapping algorithm is employed in this work to map the target DNN models to the NoC platform because it has been proven as an efficient mapping algorithm to achieve near-optimal solution [9]. In addition to the mapping algorithm, the involved routing algorithm and the network topology may affect the NoC system performance significantly. To investigate the effect bringing from the employed routing algorithm and network topology, we select XY routing algorithm and west-first adaptive routing algorithm in the experiments under two different network topologies (*i.e.*, mesh and torus.) To ensure the correctness of the proposed DNNoC-Sim, we validate the results of classification precision from DNNoC-Sim with the one of the Keras framework [18] under 32-bit, 16-bit, and 8-bit width data. Therefore, the correctness of the proposed simulator can be ensured.

After validating the DNNoC-Sim, we have evaluated the NoC-based DNN platform. In the following evaluations, we have first reported the number of off-chip memory accesses in the conventional DNN design and the NoC-based DNN design. Afterward, we have analyzed the performance of the NoC-based DNN design under different design parameters. Considering the conventional DNN design, we have selected UNPU [10] as the baseline of this work. Compared with Eyeriss [28], UNPU can support various computations in the common DNN networks due to the flexible computing flow. For the fair comparison, in DNNoC-Sim, each PE runs at 86.4 GPOS peak performance under 200MHz,(*i.e.*, similar to the UNPU design.) Besides, the frequency of the router in DNNoC-Sim is 1GHz (*i.e.*, similar to the settings in Noxim [22].)

4.2. Analysis of memory accesses

In conventional DNN accelerator, the system performance is dominated by the massive off-chip memory accesses [12]. Because the total off-chip memory latency depends on the number of memory accesses, in this section, we first compare the number of memory accesses in the conventional ASIC-based DNN design with the NoC-based designs. We consider two different NoC-based DNN design implementations: with and without dynamic DNN slicing. The former assumes that all parameters of the DNN model can fit the NoC platform at once (*i.e.*, the off-chip memory is accessed only at the beginning of the simulation). This assumption is aligned with some other NoC-based architectures, such as Intelâs Skylake [32] where the local memory size in each PE is enlarged, and the global memory size is decreased. However, this assumption may not seem realistic as in the large DNN networks, millions of parameters should fit the local on-chip memory blocks. Hence, in the latter case, dynamic DNN slicing is applied where part of the DNN model is mapped to the NoC platform at a time. It should be noted that in all designs, we have assumed that the memory bandwidth is enough to access one data within one cycle for the high-level analysis. For different memory bandwidths, the memory access latency can be changed proportionally.

Table 2 shows the number of off-chip memory accesses in the conventional DNN accelerator and the NoC-based design without DNN slicing. The number of local memory accesses is not counted in the table as the latency of on-chip memory access is much lower than an off-chip memory access. Besides, the number of on-chip memory access by using the two kinds of design paradigms are identical. The reason is that the involved PE model in the proposed simulation tool is the same as the UNPU PE model. Therefore, the computing behavior is the same as the UNPU PE as well. The NoC-based DNN design read from off-chip memory only at the beginning of the DNN computation to load all parameters of the DNN model. Then, all partial results are propagated by packets between PEs, and no more off-chip memory access is needed. In the conventional DNN design, however, regular accesses to the off-chip memory are needed to read and write partial results during DNN computation. The table shows that the new NoC-based design paradigm can reduce off-chip memory accesses by 88.1% to 99.6% under different DNN models.

As mentioned before, in practice, we cannot map a large-scale DNN model to a small-sized NoC platform. To address this issue, we proposed a dynamic DNN slicing method where the number of mapping iterations depends on the *big neuron* size as well as the NoC size. In each mapping iteration, one or more layers are mapped to the NoC platform at a time. Partial results are stored in the off-chip memory and reloaded to the NoC platform in the next mapping iteration.

Fig. 12 (a), (b), and (c) show the number of off-chip memory accesses in the LeNet, MobileNet, and VGG-16 model, respectively. Each network is evaluated under five different NoC sizes (*i.e.*, 4×4, 6×6, 8×8, 10×10, and 12×12) and three *big neuron* sizes. In parallel with each configuration, we also report the necessary number of DNN slicing (*i.e.*, mapping iteration.) Now, let us consider the LeNet model in Fig. 12(a) where the network size is 4×4 and the maximum computational capacity of a PE is 846 neuron computations. As shown in this figure, the number of memory accesses increases when the computation capacity of a PE reduces (*i.e.*, each PE handles a lower number of neurons). This is due to the fact that with lowering the PE computation capacity, a large-scale DNN model may not be mapped to the NoC platform at once. Thereby, the DNN model should be sliced and mapped to the platform, which demands to access the off-chip memory for reading/writing the partial results. As shown in Fig. 12(a), two mapping iterations are needed when the network size is 4×4, and the *big neuron* sizes are 384 or 216. In other cases, the whole DNN network can be mapped at once.

Similarly, in the VGG-16 model, it is impossible to map the whole model to a 4×4 NoC platform when the group size is 200,704. In this case, six mapping iterations are needed to execute the whole VGG-16 model, which leads to a large number of off-chip memory accesses (*i.e.*, 156,370,728). However, the number of off-chip memory accesses are still much lower than 1,165,128,192 accesses in the conventional DNN design. The number of mapping iterations can be reduced by employing larger group sizes (*i.e.*, increasing the computational capacity of the PEs) or larger NoC size. In the best case, the whole DNN model can fit the platform at once and thereby, the minimum number of off-chip memory accesses is obtained (*i.e.*, Table 2). The larger size of cluster or NoC size results in higher hardware cost, so it is a design trade-off in the NoC-based DNN accelerator design.

4.3. Performance analysis under different design parameters

In this section, we evaluate the performance of the NoC-based DNN accelerator under different design parameters such as NoC size, NoC topology, routing algorithm and group size. We analyze the PE computational latency, the NoC data delivery latency, and the overall latency (*i.e.*, the sum of both). Because all PEs work in parallel, the PE computational latency dominates by the PE, which spends the longest time to compute. Hence, the total PE computing latency is calculated by summing up the longest PE computation time in each mapping iteration. On the other hand, the NoC

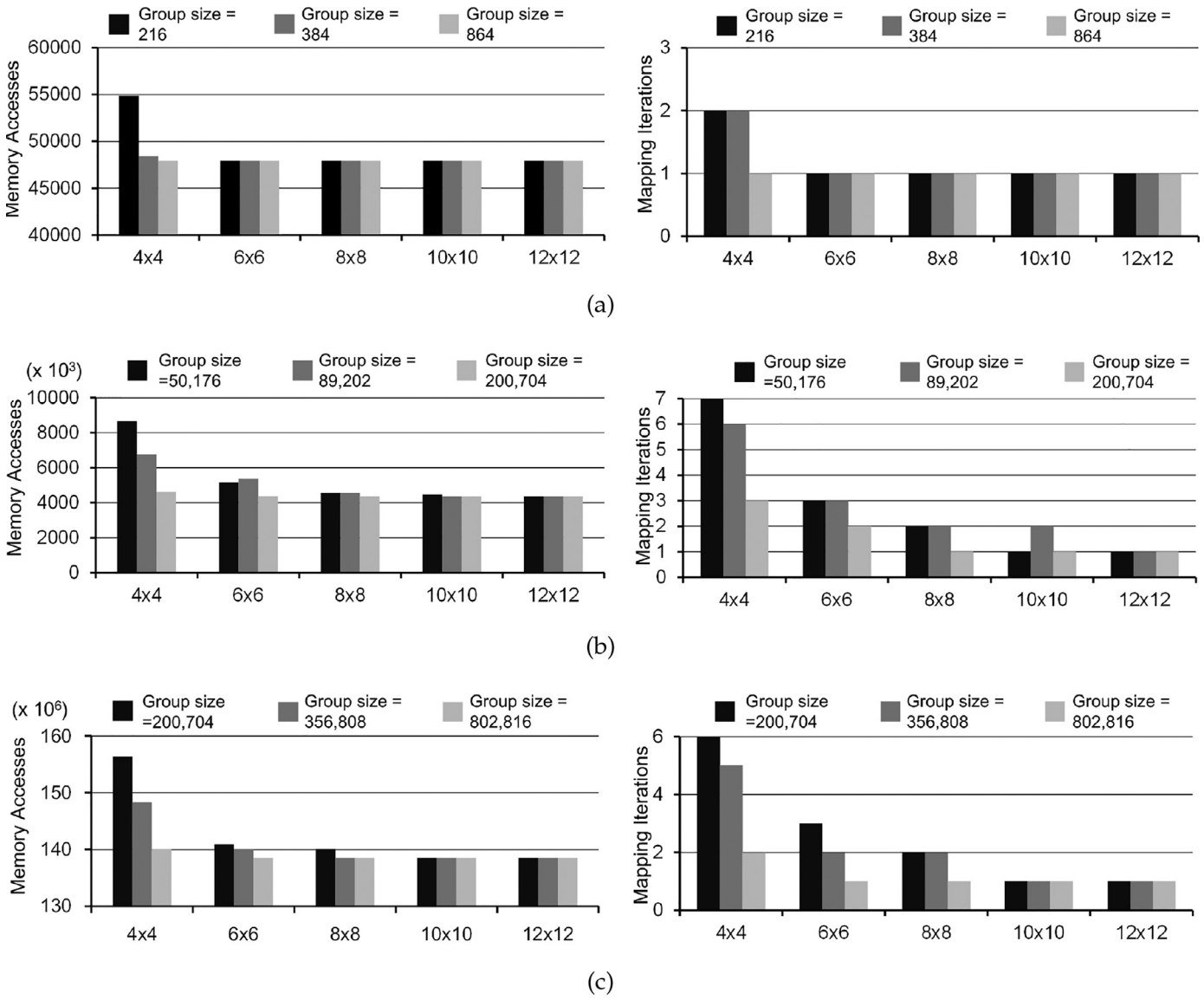

Fig. 12. The off-chip memory accesses and the corresponding number of mapping iterations under (a) LeNet (b) MobileNet, and (c) VGG-16 models.

data delivery latency reflects the data transmission time on the NoC platform. Note that, the delay of off-chip memory accesses is not considered in this set of analyses. For the evaluations, we consider three DNN models, five different NoC sizes, and three various neuron group sizes. The evaluation is performed on two different NoC topologies: mesh (Fig. 13) and torus (Fig. 14).

We evaluated various NoC sizes from 2 × 2 to 12 × 12 and reported the results for five different sizes, as shown in Fig. 13 and Fig. 14. Although the neural network scale seems different in LeNet, MobileNet, and VGG-16, they follow the same setting in the experiments of (Fig. 13 and Fig. 14. The first group size setting is chosen based on the assumption that the largest layer of the *Flattened DNN* can be mapped to the 2 × 2 NoC size within one mapping iteration. Similarly, the second and third group size settings are selected by the assumption that the largest layer can be mapped to the 3 × 3 and 4 × 4 NoC size, respectively. We then adopted these group size settings for larger NoC sizes. For example, in the LetNet model, the largest neuron layer can be mapped to the 2 × 2 NoC in one mapping iteration when a group contains 846 neurons (or less). It may take several mapping iterations to compute the whole DNN model. On the other hand, the largest neuron layer can be mapped to the 3 × 3 and 4 × 4 NoC when the group size is 384 and 216, respectively. Let us first analyze the performance of the mesh-based platform. The first observation is that the PE computational latency increases as the neuron group enlarges. This is because of the heavier PE computational load. In turn, as was discussed, a larger neuron group size leads to a lower number of off-chip memory accesses due to lower mapping iterations, as shown in Fig. 12.

The NoC data delivery is another important metric to evaluate the performance of the NoC-based DNN accelerator. A smaller neuron group size means a lower PE computation load and a shorter packet length. Furthermore, since the whole DNN model may not fit the NoC platform at once, several mapping iterations should take place. Each mapping iteration means offloading the traffic from NoC and transferring data through access to the off-chip memory. By this assumption, the traffic load on the NoC is low, and the latency is mostly dominated by the packet length. Therefore, the NoC data delivery latency is generally shorter when employing smaller group sizes. In contrary, the NoC data delivery latency becomes longer concerning the larger neuron group sizes. According to the performance analysis with different routing algo-

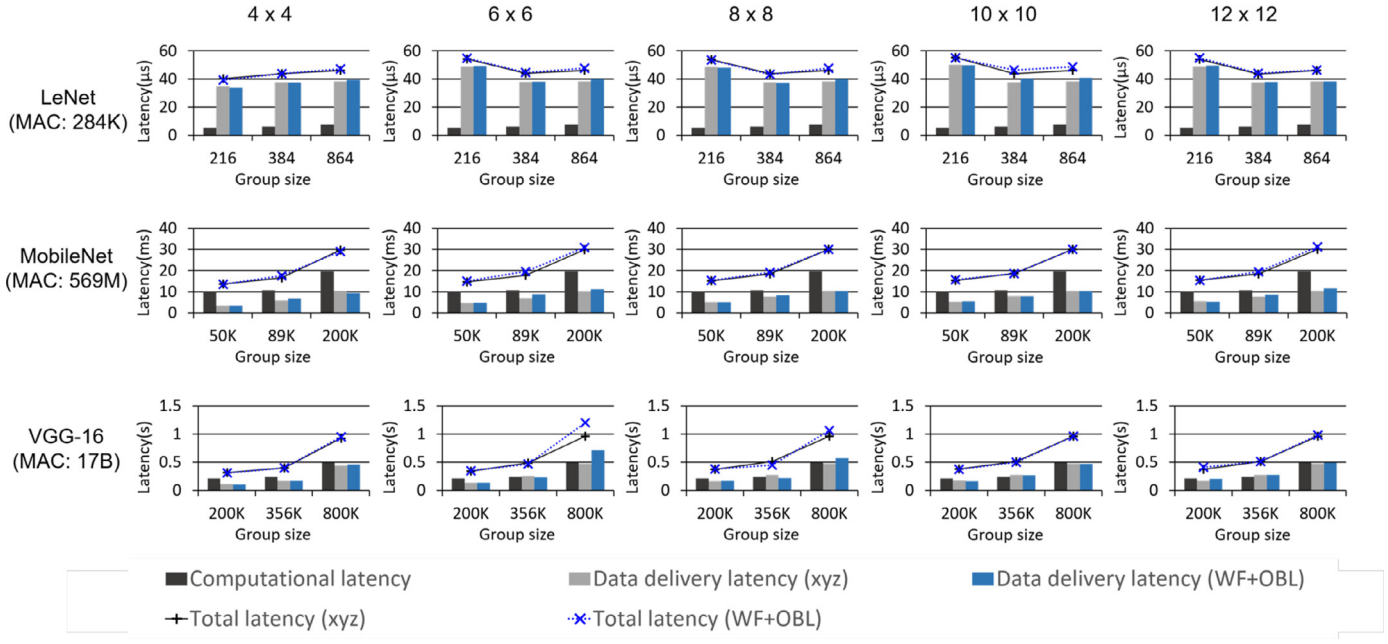

Fig. 13. The performance evaluation of three DNN models under different Mesh-based NoC and neuron group sizes.

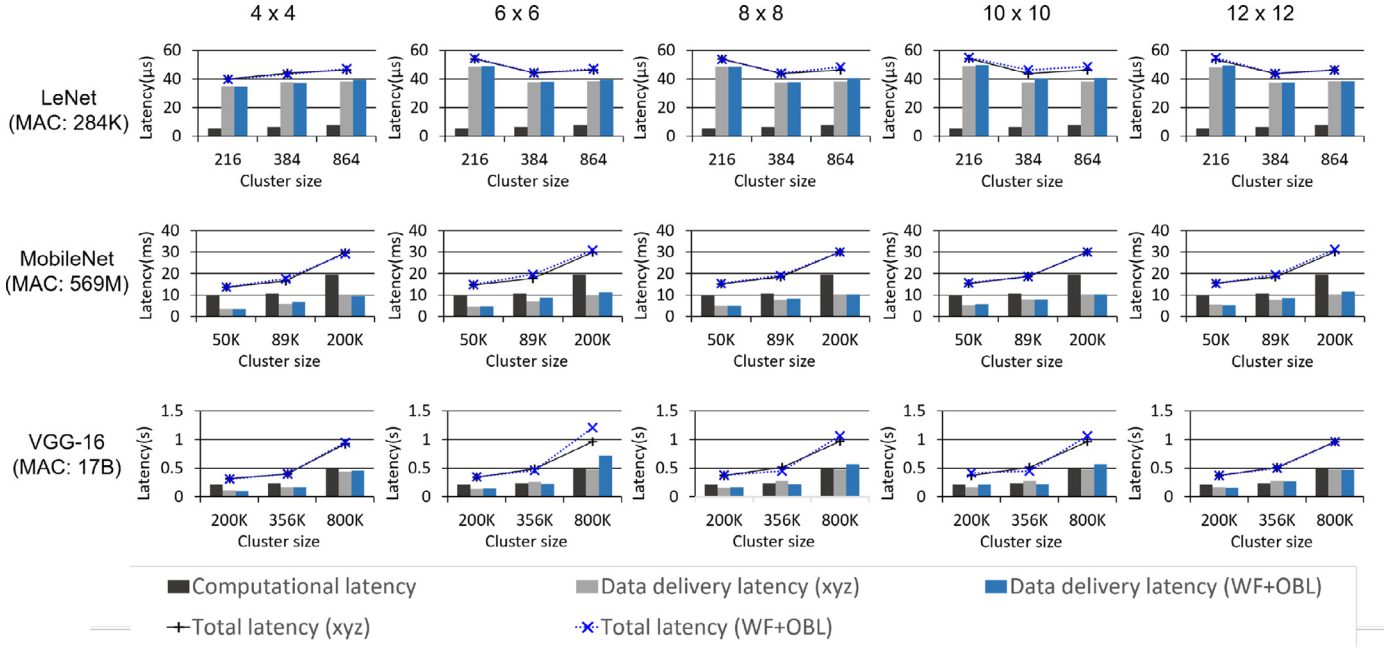

Fig. 14. The performance evaluation of three DNN models under different Torus-based NoC and neuron group sizes.

rithms, the adaptive routing algorithm does not improve the performance significantly. The reason is that the DNN operation is a kind of feed-forwarding layer-wise operation (*i.e.*, the neuron operations are performed layer-by-layer.) Therefore, the traffic load on the NoC platform is usually lite, which reduces the benefit bringing from the adaptive routing. It can also be observed that the overall latency is dominated by the NoC communication latency. Results show that the performance under the torus topology is similar to that of the mesh-based platform. This is because the larger path diversity in the torus topology is diminished by employing XY-routing and westfirst routing with OBL. In a short summary, the design parameters of neuron group size and the NoC scale are more important than the one of routing algorithm selection for the NoC-based DNN computing efficiency.

4.4. Analysis of design trade-offs between design parameters

According to Fig. 12, Fig. 13, and Fig. 14, we observe that the larger neuron group size and larger NoC size lead to lower off-chip memory accesses but longer NoC data delivery latency and PE computational latency. In the contrary, although the smaller group size and smaller NoC size lead to lower NoC data delivery latency, off-chip memory accesses are increased due to more remapping iterations. Thereby we investigate a trade-off between different design parameters to reach a high-performance NoC-based DNN accelerator.

As mentioned before, the bandwidth limitation between the off-chip memory and processing elements affects the overall performance. To simplify the problem, we assume that the 64-bit

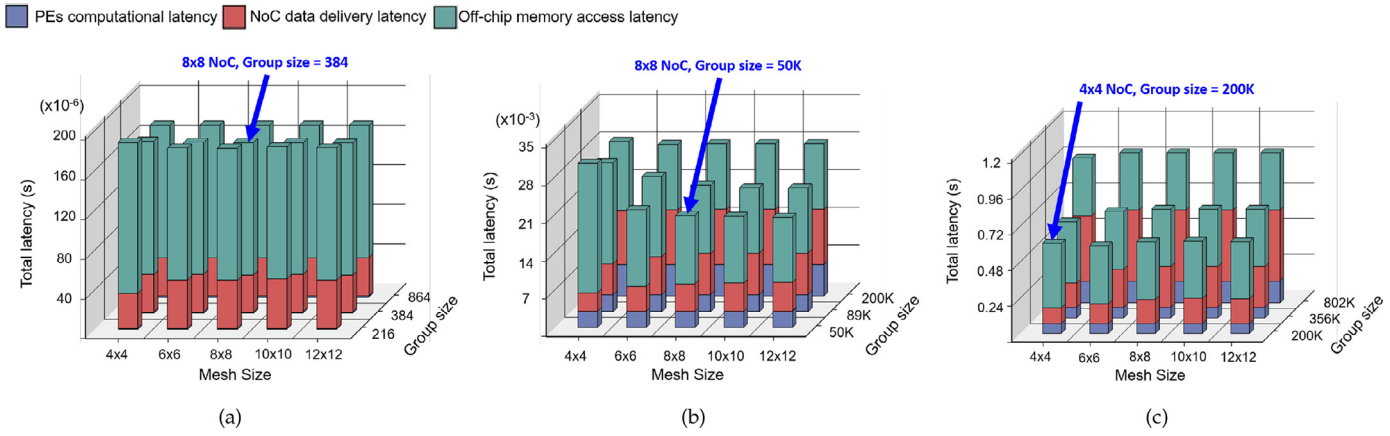

Fig. 15. The design trade-off between group size and Mesh NoC size under(a) LeNet (b) MobileNet, and (c) VGG-16 models.

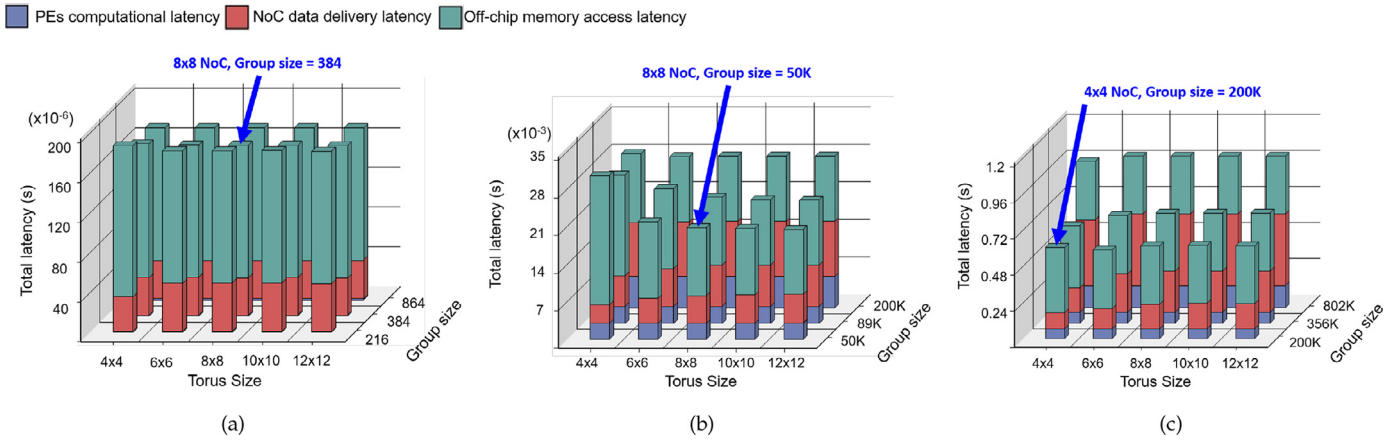

Fig. 16. The design trade-off between group size and Torus NoC size under(a) LeNet (b) MobileNet, and (c) VGG-16 models.

bandwidth that max-transmission rate is 6.4 GiB per cycle as [10] (DDR3 SDRAM memory). Fig. 15 and Fig. 16 show the design trade-off between different design parameters. Please note that in this set of analyses, the total latency is the sum of the PEs computing latency, NoC data delivery latency, and the data transfer latency to/from the off-chip memory. As illustrated in this figure, the NoC data delivery latency and the off-chip memory access latency dominate the total latency of the NoC-based DNN design. As an example, let us look at MobileNet execution on the mesh-based platform. The lowest total latency is obtained under the NoC size of 8×8 and the neuron group size of 50k. Furthermore, we can find that better performance can be achieved under small group size as well as small or medium NoC size to map the medium or large-scale DNN models (*e.g.*, MobileNet or VGG-16). The reason is that the large NoC size or large group size may lead to longer NoC data delivery latency. On the other hand, if the target DNN model is small (such as LeNet), it is cost-efficient to employ larger NoC size or larger group size to map whole DNN model to the NoC platform. In summary, it is not that the larger the PE number or the memory resources, the better the performance will be.For NoC-based DNN accelerator, each application scale has its best hardware configuration. Designers must evaluate carefully before they can create a highly efficient and cost-effective NoC-based hardware AI design, which is also our biggest intention to the proposed simulator.

5. Conclusion

The NoC-based DNN design paradigm can mitigate the DNN accelerator design complexity significantly. In this paper, we explained a complete evaluation flow to execute a DNN model on the cycle-accurate NoC-based simulation platform, called DNNoC-Sim. We first propose a DNN flattening method to convert various DNN operations into MAC-based operations. The flattening method improves the computing flexibility and enable mapping of various DNN models to the NoC platform. Furthermore, we proposed a dynamic DNN slicing and mapping algorithm to map a large-scale DNN model to a small-sized NoC platform. With the proposed simulation platform simulator, we compare the number of off-chip memory accesses between the conventional DNN accelerator and the NoC-based DNN accelerator. Afterward, we investigated the performance of the NoC-based DNN accelerator under different design parameters. The experiments showed that the NoC-based DNN accelerator could reduce 87% to 99% off-chip memory accesses compared with the conventional DNN design. It was also proven that the size of neuron group size and the NoC size are two critical design parameters to keep the system performance high in the NoC-based DNN accelerators.

Declaration of Competing Interest

The authors declare that they do not have any financial or non-financial conflict of interests.

Acknowledgment

This work was supported by the Ministry of Science and Technology under the grant MOST 108-2218-E-110-010, TAIWAN; the STINT and VR projects, SWEDEN.

References

[1] V. Sze, Y. Chen, T. Yang, J.S. Emer, Efficient processing of deep neural networks: a tutorial and survey, Proc. IEEE 105 (12) (2017) 2295–2329, doi:10.1109/JPROC.2017.2761740.
[2] B. Wolford, T. Speier, D. Bhandarkar, Qualcomm centriq 2400 processor, Hot Chips: A Symposium on High Performance Chips, HC29, 2017.
[3] J. Jeffers, J. Reinders, A. Sodani, Intel xeon phi processor high performance programming, 1st, Morgan Kaufmann, Cambridge, MA, USA, 2016.
[4] Q. Wang, N. Li, L. Shen, Z. Wang, A statistic approach for power analysis of integrated gpu, Soft Comput. 23 (3) (2019) 827–836.
[5] F. Abuzaid, S. Hadjis, C. Zhang, C. Ré, Caffe con troll: shallow ideas to speed up deep learning, CoRR (2015). abs/ 1504.04343.
[6] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, X. Ji, High-performance fpga-based cnn accelerator with block-floating-point arithmetic, IEEE Trans. Very Large Scale Integr. VLSI Syst. (2019) 1–12, doi:10.1109/TVLSI.2019.2913958.
[7] J. Kim, D. Park, C. Nicopoulos, N. Vijaykrishnan, C.R. Das, Design and analysis of an noc architecture from performance, reliability and energy perspective, in: 2005 Symposium on Architectures for Networking and Communications Systems (ANCS), 2005, pp. 173–182.
[8] Y. Chen, T. Yang, J. Emer, V. Sze, Eyeriss v2: a flexible accelerator for emerging deep neural networks on mobile devices, IEEE J. Emerging Sel. Top. Circuits Syst. 9 (2) (2019) 292–308, doi:10.1109/JETCAS.2019.2910232.
[9] X. Liu, W. Wen, X. Qian, H. Li, 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, doi:10.1109/ASPDAC.2018.8297296.
[10] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, H. Yoo, Unpu: an energy-efficient deep neural network accelerator with fully variable weight bit precision, IEEE J. Solid-State Circuits 54 (1) (2019) 173–185, doi:10.1109/JSSC.2018.2865489.
[11] R. Hojabr, M. Modarressi, M. Daneshtalab, A. Yasoubi, A. Khonsari, Customizing clos network-on-chip for neural networks, IEEE Trans. Comput. 66 (11) (2017) 1865–1877, doi:10.1109/TC.2017.2715158.
[12] H. Kwon, A. Samajdar, T. Krishna, Rethinking nocs for spatial neural network accelerators, in: Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip, in: NOCS '17, ACM, New York, NY, USA, 2017, pp. 19:1–19:8, doi:10.1145/3130218.3130230.
[13] A. Firuzan, M. Modarressi, M. Daneshtalab, M. Reshadi, Reconfigurable network-on-chip for 3d neural network accelerators, in: 2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS), 2018, pp. 1–8, doi:10.1109/NOCS.2018.8512170.
[14] P.C. Holanda, C.R.W. Reinbrecht, G. Bontorin, V.V. Bandeira, R.A.L. Reis, Dhyana: a noc-based neural network hardware architecture, in: 2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2016, pp. 177–180, doi:10.1109/ICECS.2016.7841161.
[15] J. Liu, J. Harkin, L.P. Maguire, L.J. McDaid, J.J. Wade, G. Martin, Scalable networks-on-chip interconnected architecture for astrocyte-neuron networks, IEEE Trans. Circuits Syst. I Regul. Pap. 63 (12) (2016) 2290–2303, doi:10.1109/TCSI.2016.2615051.
[16] K.J. Chen, T. Wang, Nn-noxim: High-level cycle-accurate noc-based neural networks simulator, in: 2018 11th International Workshop on Network on Chip Architectures (NoCArc), 2018, pp. 1–5, doi:10.1109/NOCARC.2018.8541173.
[17] K.-C.J. Chen, T.-Y.G. Wang, Y.-C.A. Yang, Cycle-accurate noc-based convolutional neural network simulator, in: Proceedings of the International Conference on Omni-Layer Intelligent Systems, in: COINS '19, ACM, New York, NY, USA, 2019, pp. 199–204, doi:10.1145/3312614.3312655.
[18] N. Ketkar, Introduction to Keras, in: Deep Learning with Python, Springer, 2017, pp. 97–111.
[19] D. Smilkov, S. Carter, D. Sculley, F.B. Viégas, M. Wattenberg, Direct-manipulation visualization of deep networks, CoRR (2017). abs/ 1708.03788.
[20] A.K. Jain, J. Mao, K. Mohiuddin, Artificial neural networks: a tutorial, Computer (Long Beach Calif) (3) (1996) 31–44.
[21] W. Nan-lan, P. Xiang-gao, Application of matlab/nntool in neural network system, Comput. Modern. 12 (2012) 125–128.
[22] V. Catania, A. Mineo, S. Monteleone, M. Palesi, D. Patti, Cycle-accurate network on chip simulation with noxim, ACM Trans. Model. Comput. Simul. 27 (1) (2016) 4:1–4:25, doi:10.1145/2953878.
[23] A.V. de Mello, Atlas-an environment for noc generation and evaluation, 2011.
[24] J. Wang, Y. Huang, M. Ebrahimi, L. Huang, Q. Li, A. Jantsch, 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, in: MES '16, ACM, New York, NY, USA, 2016, pp. 18–25, doi:10.1145/2934495.2949544.
[25] N. Jiang, G. Michelogiannakis, D. Becker, B. Towles, W.J. Dally, Booksim 2.0 Users Guide, Standford University, 2010.
[26] Y.-L. Boureau, J. Ponce, Y. LeCun, A theoretical analysis of feature pooling in visual recognition, in: Proceedings of the 27th International Conference on Machine Learning (ICML-10), 2010, pp. 111–118.
[27] D. Scherer, A. Müller, S. Behnke, Evaluation of pooling operations in convolutional architectures for object recognition, in: International Conference on Artificial Neural Networks, Springer, 2010, pp. 92–101.
[28] Y. Chen, T. Krishna, J.S. Emer, V. Sze, Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks, IEEE. Solid-State Circuits 52 (1) (2017) 127–138, doi:10.1109/JSSC.2016.2616357.
[29] Y. LeCun, B.E. Boser, J.S. Denker, D. Henderson, R.E. Howard, W.E. Hubbard, L.D. Jackel, Handwritten digit recognition with a back-propagation network, in: Advances in Neural Information Processing Systems, 1990, pp. 396–404.
[30] A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, Mobilenets: efficient convolutional neural networks for mobile vision applications, CoRR (2017). abs/ 1704.04861.
[31] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, CoRR (2014). abs/ 1409.1556.
[32] J. Doweck, W. Kao, A.K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, A. Yoaz, Inside 6th-generation intel core: new microarchitecture codenamed skylake, IEEE Micro 37 (2) (2017) 52–62, doi:10.1109/MM.2017.38.

Kun Chih (Jimmy) Chen (IEEE S'10 M'14) received his B.S. degree from National Taiwan Ocean University, Taiwan, in Computer Science and Engineering in 2007. He received the M.S. degree from National Sun Yat Sen University, Taiwan, in Computer Science and Engineering in 2009. He received the PhD degree from Nation Taiwan University, Taiwan, in Graduate Institute of Electronics Engineering in 2013. From October 2014 to January 2015, he served as a postdoctoral fellow in Intel NTU Connected Context Computing Center working on the development of Gree n Sensing Platform for Internet of Things (IoTs), Reliable Thermoelectric Converter, and Power aware Software Defined Network (SDN). From February 2015 to July 2016, Dr. Chen joined the faculty of Electronic Engineering Department of Feng Chia University. He is currently an Assistance 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, a nd VLSI/CAD design. Dr. Chen served as Technical Program Committee (TPC) Chair of the International Workshop on Network on Chip Architectures (NoCArc 2018), General Chair of the International Workshop on Network on Chip Architectures (NoCArc 2019) and Gues t Editor of Journal of Systems Architecture. Besides, he also served as the technical program committee of some major IEEE international conferences, such as ISCAS and SOCC. Dr. Chen received the Best Paper Award of International Symposium on VLSI Design, Automation and Test (VLSI DAT 2014), the Best Paper Award of International Joint Conference on Convergence (IJCC 2016), and the Best PhD Dissertation Award of IEEE Taipei Section in 2014. He is a member of IEEE

Masoumeh (Azin) Ebrahimi received a PhD degree with honors from University of Turku, Finland in 2013 and MBA in 2015. She is currently a senior researcher at KTH Royal Institute of Technology, Sweden and an Adjunct professor (Docent) at 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.

Ting Yi Wang received his B.S. degree from National Sun Yat sen University, Taiwan, in Computer Science and Engineering in 2017. 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.

Yueh Chi Yang received his B.S. degree from National Sun Yat sen University, Taiwan, in Computer Science and Engineering in 2018. 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 NoC system design.

Yuan Hao Liao received his B.S. degree from National Changhua University of Education, Taiwan, in Computer Science and Information Engineering in 2018. Currently, he is pursuing his M.S. degree at Department of Computer Science and Engineering, National Sun Yat sen Univ ersity, Taiwan. His research fields interest in the thermal aware multicore system design.