Pooling On-the-go for NoC-based Convolutional Neural Network Accelerator*

Wenyao Zhu[1 [0000−0002−4911−0257]], Yizhi Chen[1 [0000−0001−8488−3506]], and Zhonghai Lu[1 [0000−0003−0061−3475]]

KTH Royal Institute of Technology, Stockholm, Sweden
{wenyao,yizhic,zhonghai}@kth.se

Abstract. Due to the complexity and diversity of deep convolutional neural networks (CNNs), Network-on-chip (NoC) based CNN accelerators have grown in popularity to improve inference efficiency and flexibility. Current optimization approaches focus on computational-heavy layers. Therefore, pooling layers are often ignored and processed individually using general processing units. In this work, we explore the acceleration of pooling layers by in-network processing. We propose a pooling on-the-go method to do the pooling operations while transmitting its prior layer outputs. Consequently, we combine the pooling layer with its prior convolution layer to remove unnecessary data movements. We demonstrate our method on a cycle-accurate NoC-CNN accelerator simulator on two CNN models, LeNet and VGG16. The results show that the processing time of individual pooling layers is almost eliminated by around 99%. Compared with the pooling standalone baseline, we can achieve 1.09x speedup in the full LeNet model, and up to 1.16x speedup in the combined layers that our approach applies.

Keywords: CNN Accelerator · In-network Processing · Network-on-Chip · Pooling.

1 Introduction

Network-on-chip-based convolutional neural network accelerators (NoC-CNNs) have gained significant interest since their structure is flexible and enables large-scale parallelism for running myriad CNN models on the same chip [3, 13]. In current NoC-CNN designs, CNN models undergo layer-by-layer processing, with intermediate feature maps stored in memory [4, 8]. Pooling layers are processed like convolutional layers and handled by PEs in NoC-CNNs [5, 14].

Reducing pooling layer latency doesn't attract much attention as their computational cost is less significant than other layers. However, since pooling layers are treated independently, the memory access and communication latency are not negligible. This work is motivated by the in-network processing concept and explores the possibility of offloading the pooling operation to network devices.

* The research has been supported in part by Vetenskapsrådet (Swedish Research Council) through the LearnPower project (2020-03494).

We propose the pooling on-the-go approach for NoC-CNN to seamlessly integrate the pooling layer operations while transmitting the adjacent convolution layer outputs. In this case, we can remove the pooling layer latency with little overhead. We summarize our contributions as follows.

- We propose a new optimization method that combines the pooling layer with its prior layer for NoC-CNN to eliminate pooling layer latency.
- We design the pooling on-the-go structure in the network interface of NoC to offload pooling operations from PE.
- We validate our approach in a cycle-accurate NoC-CNN simulator on two CNN models and discuss the design trade-off of our approach.

2 Related Work

Recent works regarding NoC-CNNs employ novel NoC structure designs to reduce energy consumption in computational-heavy layers [8, 11]. Other researches on NoC-CNN improve hardware utilization for convolution layers [4, 7]. However, these state-of-the-art optimization techniques often bypass pooling layers, though they are essential in CNN models to reduce spatial dimensions.

Two common approaches exist for processing pooling layers. The first is to treat pooling as a standalone layer, as adopted by the aforementioned NoC-CNN architectures [4, 7, 8, 11]. They use local or global buffers to store intermediate layer outputs and conduct inference in a layer-completing order. This approach reduces design complexity and offers adaptability to various workloads. However, it introduces overhead in communication and memory access for distributing and aggregating data. The second way is to merge it with its prior convolution layer from the model side. Alwani *et al.* first explore the layer fusion technique by creating a computation pyramid across adjacent layers to reduce data transfer [2]. They pipeline the computation in PE to save bandwidth, which may increase the overall latency. Additionally, extra on-chip storage is required for overlapping pyramids. These challenges are intensified when implementing such techniques in NoC-CNN. Simba [14] presents a large-scale CNN inference system using a two-level mesh NoC. They assume the weights remain stationary in PEs and only the neuron outputs are transferred across PE. Then a post-processing unit (PPU) for pooling and non-linear activation is deployed in each PE for cross-layer fusion. This can improve data locality but requires large on-chip buffers. Zhu *et al.* propose a sparse skipping technique in NoC-CNN to reduce the memory overhead for CNN inference [17]. They make the PPU an independent core in NoC and link it with off-chip memory. It can mitigate the drawback in storage scale. However, the standout PPU creates a bottleneck where every convolution output will be sent to it for pooling.

These layer fusion techniques are beneficial for reducing data movement, but they are also possible to increase end-to-end latency in NoC-CNNs. Inspired by the concept of in-network processing [12], we can offload the pooling function in PPU from PE to the network for NoC-CNN. Thereby, the pooling operation is combined with the communication in its prior layer to reduce the layer latency.

3 Pooling On-the-go Design for NoC-CNN

A common NoC-CNN architecture [4] is depicted in **Fig.** 1. In the mesh network, two types of cores are connected to the network interface (NI) and router. The PE core contains the arithmetic unit and local buffers for neuron computations. The memory controller (MC) core is the portal to the on-chip buffer that transmits data to and from other PE cores through NoC. The on-chip buffer hosts the input feature map (IFMap), weight table, and output feature map (OFMap) for a layer. It updates the buffered weights after completing the current layer from off-chip memory, and OFMap is reused as the next IFMap.

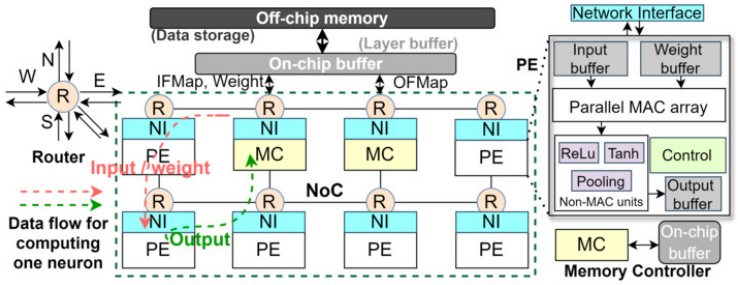

Fig. 1: NoC-based CNN accelerator schematic.

To accommodate various layer types, we utilize the unfolding technique in an output-oriented manner similar to the assumption in [4]. Each layer is represented by a batch of output neurons that equals the size of OFMap. NoC-CNN maps these neurons to PEs during CNN model loading. Assuming a layer's OFMap size is M, and NoC has k PEs, with one PE computing one neuron per round, it necessitates $R = \lceil M/k \rceil$ rounds to complete this layer.

3.1 Pooling On-the-go Approach

In the pooling standalone NoC-CNN design, the pooling operations are processed in PE. For example, in a DNNoC-like [4] workflow, PE will save the convolution layer (Conv.) outputs to off-chip memory and load it again for pooling operations. This costs extra data moving overhead and lowers the utilization of PE. Therefore, we propose a pooling on-the-go approach to combine the pooling layer computation during transmission, minimizing redundant data movements. **Fig.** 2 illustrated the standalone and on-the-go pooling processes, both starting at T_0. For standalone pooling, the Conv. OFMap is completed at T_1. Then it serves as the pooling layer IFMap and finishes at T_2. Alternatively, in our approach, the pooling operation is done during the transmission of Conv. OFMap from PE to MC. Then MC receives the pooling OFMap at T_2'. The latency improvement of the combined layer is $T_2 - T_2'$.

To realize pooling on-the-go in NoC-CNN, there are three possible positions other than PE, including router, NI, and MC. In flit-based virtual channel (VC) routers, data as payload are unknown to existing routing logic. Therefore, extra

decoding units are required for each port in case the pooling happens in the router, introducing a large area and latency overhead. NI and MC are equivalent to pooling block implementation as they are near the end of the data path. Since MC is directly connected to the global buffer, it runs in a slower clock domain than NI. Hence NI is a more suitable position, and only NI connected to MC needs modification. **Fig.** 2 also compares the data path of two pooling designs, where the pooling IFMap transfer is removed in our pooling on-the-go approach.

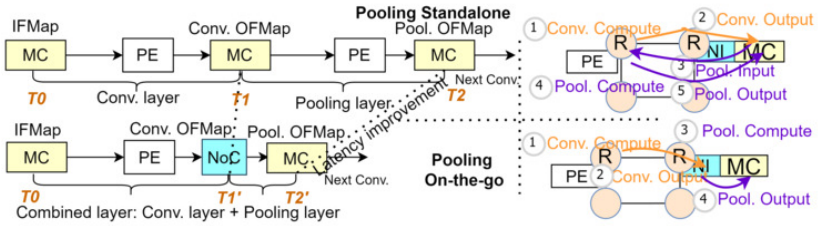

Fig. 2: Timeline (left) and data path (right) of standalone and on-the-go pooling.

3.2 Pooling On-the-go Implementation

A canonical NI consists of two symmetric ports as the interface between core and router [6]. The original *outport* design is shown at the top of **Fig.** 3, where core messages are turned into flits and injected into the router. We develop a pooling on-the-go block to offload the pooling layer operations from PE and integrate it in our new *inport* design. The pooling block structure is depicted in the lower half of **Fig.** 3. We have a temporary buffer to store intermediate pooling values, a pooling function block for calculation, and a controller to manage the whole process. Two multiplexers are used to enable or bypass the new pooling block.

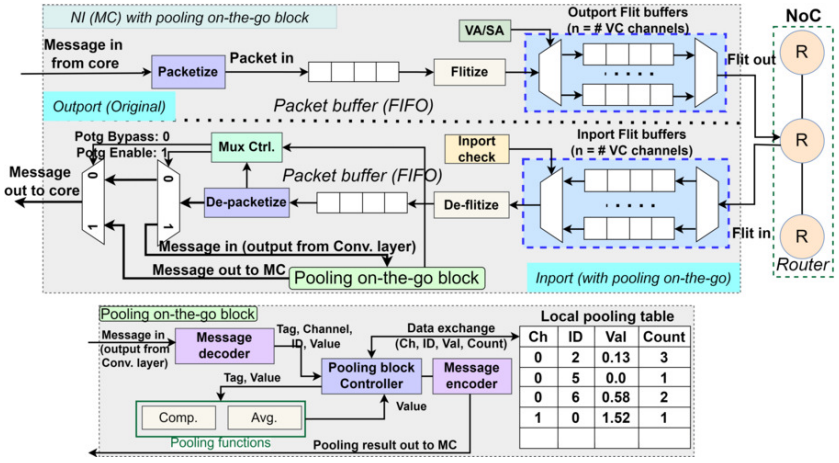

Fig. 3: Network interface with proposed pooling block and the block design.

Message Header A 1-bit **tag** is added to the message header to judge whether a message requires the new pooling service and control the multiplexer. Note that

only a combination of a pooling layer after a convolution layer can be optimized from our approach. To correctly perform pooling operations, the message header also includes a 1-bit signal that tells the pooling function, e.g. max pooling and average pooling. Then the channel and ID are given to identify the location in the pooling layer OFMap instead of the initial Conv. OFMap. The remaining bits are reserved for different pooling kernel sizes.

Local Pooling Table This is a key component to store the temporary results for pooling, which is organized as a buffer table that can access an entire row at a time. In each row, there are four elements, representing channel, ID, value, and count. "Count" indicates the number of messages received for the same channel and ID. We don't have to allocate the full size of pooling OFMap in the pooling table at once. Since there are multiple MCs in NoC, each will communicate with a fixed number of PEs. Then in each computation round, the number of messages that NI receives is also fixed. So the pooling table only needs the space to hold these messages in one round simultaneously. For example, if in the NoC-CNN, five PEs are communicating with this NI to reach MC, then we only need five rows in the pooling table.

Pooling Block Controller **Fig.** 4 shows the workflow of the pooling on-the-go controller. The input message from the packet buffer in *inport* is sent to the pooling block when the valid **tag** is given. Then it performs the following steps:

1. The decoder extracts function, channel, ID, and value from the message.
2. Controller checks the pooling table for index comparison on channel and ID.
3. If no matching entry, data is written to an empty row with Count set to 1.
4. If a matching entry exists, we calculate the pooling result of the input value and the stored value. The Count of this row increases by one.
5. If the Count is smaller than the pooling kernel size, the result is written back. Otherwise, a new message is created to deliver the final pooling result.

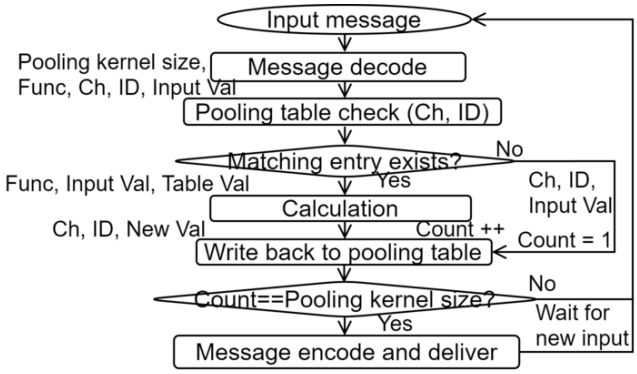

Fig. 4: Pooling block controller workflow.

The controller iteratively processes the valid messages until the whole pooling layer OFMap is completed and stored in the on-chip buffer. Our design ensures that the pooling operation is finished in NI during the transmission of its prior convolution outputs.

4 Evaluation and Results

4.1 Experimental Setup

We build a cycle-accurate NoC-CNN simulator based on the NoC simulator in [16] to evaluate our approach. The VC NoC architecture is adopted from Gem5-Garnet [1]. We implement the pooling standalone baseline. Then we deploy our pooling on-the-go functions in the simulator for tests. The inference result and layer latency when running the same CNN model on baseline NoC-CNN and our approach are recorded to validate the correctness and evaluate the latency improvement. Based on **Fig.** 2, we define the latency reduction P of the pooling layer and the speedup ρ of the combined layer using our approach as

$$P = \frac{T_2 - T_2'}{T_2 - T_1} \times 100\%, \quad \rho = \frac{T_2}{T_2'}. \tag{1}$$

Simulation System Configuration The default NoC-CNN configurations are summarized in **Tab.** 1. We arrange cores in a row-major order, for example, cores with ID 0 to 7 are in the first row. Then we separate the whole network into four identical 4×4 subareas. MCs are positioned at each subarea's center with ID 17, 18, 21, 22, 41, 42, 45, and 46. We simulate an ASIC NoC-CNN in which the NoC components (router, NI) can run at 2 GHz [14, 16]. PE and MC are usually slower than routers since they have more complex combinational circuits for calculation. The default PE frequency is 200 MHz as the UNPU in [10]. We also change the PE frequency to 500 MHz and 1 GHz to simulate different hardware designs [14]. The latency results are in the unit of NoC cycles.

Table 1: NoC-CNN default configuration

Item	Amount	Configuration
NoC	64 nodes	8×8 mesh network. 2 GHz. X-Y routing. 256-bit link bandwidth. Bidirectional link.
Router	64 routers	4 VCs per port, 4-flit buffers per VC.
Core	64 cores	56 PEs and 8 MCs. 25 MACs per PE cycle. 200 MHz.

We select two representative CNN models, the small LeNet model [9] and the large VGG16 model [15], for evaluation. They both contain the combination of a pooling layer after a convolution layer that our approach applies. The input size is $32 \times 32 \times 1$ for LeNet and $224 \times 224 \times 3$ for VGG16.

LeNet comprises two combined layers that our approach applies. **Tab.** 2a gives the parameters for these layers: L1 (Conv1+MaxP1), L2 (Conv2+MaxP2). ϕ shows the proportion for computation (#OP) of pooling over the total combined layer. VGG16 has 13 convolution layers, five of which are followed by a max pooling layer that can use the pooling on-the-go method. **Tab.** 2b summarizes the parameters and the proportion of pooling computation of these five layers. For ease of simulation, we sample VGG16 with the same kernels but smaller feature maps, then scale them up to get the latency for the full model.

Table 2: Layer parameters for pooling on-the-go

(a) LeNet

Layer		Neurons	Kernel size	#OP	ϕ (%)
L1	Conv1	4704	5×5	(1×5×5)×4704	3.846
	MaxP1	1176	2×2	(2×2)×1176	
L2	Conv2	1600	5×5	(6×5×5)×1600	0.662
	MaxP2	400	2×2	(2×2)×400	

(b) VGG16

Layer		Neurons	Kernel size	#OP	ϕ (%)
L1	Conv2	3211264	3×3	(64×3×3)×3211264	0.173
	MaxP1	802816	2×2	(2×2)×802816	
L2	Conv4	1605632	3×3	(128×3×3)×1605632	0.087
	MaxP2	401408	2×2	(2×2)×401408	
L3	Conv7	802816	3×3	(256×3×3)×802816	0.043
	MaxP3	200704	2×2	(2×2)×200704	
L4	Conv10	401408	3×3	(512×3×3)×401408	0.022
	MaxP4	100352	2×2	(2×2)×100352	
L5	Conv13	100352	3×3	(512×3×3)×100352	0.022
	MaxP5	25088	2×2	(2×2)×25088	

4.2 Experimental Results

The LeNet latency on baseline NoC-CNN and pooling on-the-go approach (with label P) are plotted in **Fig.** 5. In three evaluated situations, the average latency reductions for MaxP1 and MaxP2 are 98.98% and 97.39%. Speedup ρ for the combined layers L1 and L2 are 1.148x and 1.054x. The more significant improvement in L1 is primarily due to the higher pooling proportion ϕ compared to L2. The core frequency determines the computation latency. Thus a shorter T_1 and T_2 in **Eq.** (1) are expected for faster PE, while $(T_2' - T_1)$ remains the same. Consequently, the best speedup appears for the 1 GHz setup, which is 1.09x for the full model, 1.16x for L1, and 1.06x for L2. Our observation validates that the proposed approach can achieve a better speedup when PE frequency is higher.

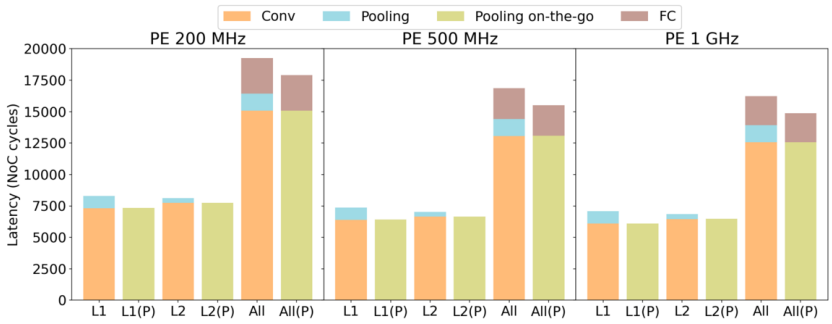

Fig. 5: Inference latency of LeNet.

Fig. 6 presents the inference latency of L1 to L5 in VGG16. The full VGG16 model needs 392 million NoC cycles on the baseline NoC-CNN, so we don't plot it here. For five pooling layers MaxP1 to MaxP5, the latency reductions are all over 99.9%. The average speedup of the combined layers from L1 to L5 ranges from 1.016x to 1.0015x. The overall speedup on VGG16 are 1.0036x, 1.0035x, and 1.003x when the PE frequency is 1 GHz, 500 MHz, and 200 MHz, respectively. We can see that the pooling layer latency is almost eliminated using our approach as we expected. However, the pooling operation only occupies a very small proportion of VGG16. For example in the 1 GHz PE setup, it takes around 1.2 million cycles, which is 0.359% of the full model. Therefore, the speedup of our approach on VGG16 is not obvious.

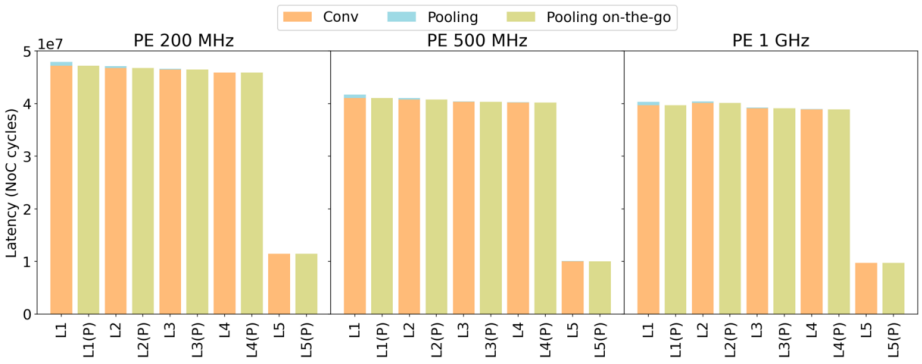

Fig. 6: Inference latency of VGG16.

4.3 Discussion

From the evaluation results, the pooling layer latency is almost eliminated when our approach is applied. The overall speedup depends on the workload proportion of the pooling layer in the CNN model. In LeNet, this proportion is way higher than in VGG16. Thus our method can achieve much better speedup results in LeNet compared with VGG16. Moreover, our approach replaces the standalone pooling units in PE with pooling on-the-go blocks in NI connected to MC. MC constitutes a small fraction of total cores in NoC-CNN, and the area for the extra pooling table and simple controller in the block is not significant. So the proposed approach can maintain or even save area for NoC-CNN. Also, the reduced memory accesses for the standalone pooling layer can improve energy efficiency. Regarding the communication overhead, since the Conv. output packet only contains one output neuron value, the added message header will not affect the packet size as it remains a single-flit packet.

Therefore, the proposed pooling on-the-go approach can improve the performance and efficiency of NoC-CNNs, especially when the target CNNs have lightweight convolution layers followed by normal pooling layers.

5 Conclusion

In this paper, we present a pooling on-the-go approach with a unique pooling block design in the network interface of NoC-CNN. Our method aims to do pooling operations during the transmission of its prior layer outputs, thereby minimizing the pooling layer processing latency. Compared with common standalone pooling, our approach reduces superfluous data transfers to optimize efficiency. We evaluate our method on two CNN models, LeNet and VGG16. The simulation results show that our approach can almost eliminate the pooling layer latency. We achieve 1.09x speedup for the full LeNet model. For the speedup of one combined layer, we reach up to 1.16x for LeNet and 1.016x for VGG16.

This work opens the opportunity for cross-layer optimization using in-network processing in NoC-CNN designs. The RTL implementation of our approach and tradeoffs between timing and power/area will be investigated in future work.

References

1. Agarwal, N., Krishna, T., Peh, L.S., Jha, N.K.: GARNET: A detailed on-chip network model inside a full-system simulator. In: IEEE International Symposium on Performance Analysis of Systems and Software. pp. 33–42. IEEE (2009)
2. Alwani, M., Chen, H., Ferdman, M., Milder, P.: Fused-layer CNN accelerators. In: 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 1–12. IEEE (2016)
3. Chen, K.C., Ebrahimi, M., Wang, T.Y., Yang, Y.C.: NoC-based DNN accelerator: A future design paradigm. In: Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip. pp. 1–8 (2019)
4. Chen, K.C., Ebrahimi, M., Wang, T.Y., Yang, Y.C., Liao, Y.H.: A NoC-based simulator for design and evaluation of deep neural networks. Microprocessors and Microsystems **77**, 103145 (2020)
5. Chen, Y.H., Yang, T.J., Emer, J., Sze, V.: Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. IEEE Journal on Emerging and Selected Topics in Circuits and Systems **9**(2), 292–308 (2019)
6. Dally, W.J., Towles, B.P.: Principles and practices of interconnection networks. Elsevier (2004)
7. Hu, X., Huang, H., Li, X., Zheng, X., Ren, Q., He, J., Xiong, X.: High-performance reconfigurable DNN accelerator on a bandwidth-limited embedded system. ACM Transactions on Embedded Computing Systems (2022)
8. Kwon, H., Samajdar, A., Krishna, T.: MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. ACM SIGPLAN Notices **53**(2), 461–475 (2018)
9. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)
10. Lee, J., Kim, C., Kang, S., Shin, D., Kim, S., Yoo, H.J.: UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision. IEEE Journal of Solid-State Circuits **54**(1), 173–185 (2018)
11. Liu, X., Wen, W., Qian, X., Li, H., Chen, Y.: Neu-NoC: A high-efficient interconnection network for accelerated neuromorphic systems. In: 23rd Asia and South Pacific Design Automation Conference (ASP-DAC). pp. 141–146. IEEE (2018)
12. Lu, Z.: PiN: Processing in Network-on-Chip. IEEE Design & Test (2023)
13. Nabavinejad, S.M., Baharloo, M., Chen, K.C., Palesi, M., Kogel, T., Ebrahimi, M.: An overview of efficient interconnection networks for deep neural network accelerators. IEEE Journal on Emerging and Selected Topics in Circuits and Systems **10**(3), 268–282 (2020)
14. Shao, Y.S., Clemons, J., Venkatesan, R., Zimmer, B., Fojtik, M., Jiang, N., Keller, B., Klinefelter, A., Pinckney, N., Raina, P., et al.: Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 14–27 (2019)
15. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: 3rd International Conference on Learning Representations, ICLR (2015)
16. Wang, B., Lu, Z.: Flexible and efficient QoS provisioning in AXI4-based network-on-chip architecture. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **41**(5), 1523–1536 (2021)
17. Zhu, L., Fan, W., Dai, C., Zhou, S., Xue, Y., Lu, Z., Li, L., Fu, Y.: A NoC-based spatial DNN inference accelerator with memory-friendly dataflow. IEEE Design & Test (2023)