Learn-to-Scale: Parallelizing Deep Learning Inference on Chip Multiprocessor Architecture

Kaiwei Zou†‡, Ying Wang†‡, Huawei Li†‡, Xiaowei Li†‡
†SKLCA, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
‡University of Chinese Academy of Sciences, Beijing 100049, China
Email: {zoukaiwei, wangying2009, lihuawei, lxw}@ict.ac.cn

***Abstract*—Accelerating deep neural networks on resource-constrained embedded devices is becoming increasingly important for real-time applications. However, in contrast to the intensive research works on specialized neural network inference architectures, there is a lack of study on the acceleration and parallelization of deep learning inference on embedded chip-multiprocessor architectures, which are favored by many real-time applications for superb energy-efficiency and scalability. In this work, we investigate the strategies of parallelizing single-pass deep neural network inference on embedded on-chip multi-core accelerators. These methods exploit the elasticity and noise-tolerance features of deep learning algorithms to circumvent the bottleneck of on-chip inter-core data moving and reduce the communication overhead aggravated as the core number scales up. The experimental results show that the communication-aware sparsified parallelization method improves the system performance by 1.6×-1.1× and achieves 4×-1.6× better interconnects energy efficiency for different neural networks.**

Keywords—parallelization, multi-core, inference, neural network, embedded devices

I. Introduction

Enabling real-time deep neural network inference is becoming an increasingly important requirement on embedded devices due to the popularity of edge visual recognition, robotics control and speech recognition. However, the large quantity of numerical operations and parameters induced by CNN inference poses a significant challenge to the resource-constrained devices [6]. As an alternative to conventional general-purpose CPU and GPU cores, customized neural network accelerators (NNA) like [2, 4] are gaining popularity for their efficiency. While the trend of general-purpose processor design has been shifting from single-core to on-chip multi-core architectures in the past decade, some of the recent neural accelerators are also scaling up to chip-multiprocessors (CMP) architectures to deal with the massive data-level parallelism in deep neural network inference [1, 3, 5, 8]. Accompanied by core-level gating, multi-core neural accelerators are also showing better elasticity and energy efficiency [4].

When more and more CMP-like neural accelerators begin connecting multiple separate PE arrays or clusters through on-chip networks for higher processing throughput [1, 5, 8], the discussion on how to parallelize single-pass neural network inference on such architectures is scarce. Most of the prior works on deep neural network parallelization confine themselves to the node-level or rack-level machine and generally concern more about the model training instead of network inference [6]. Recently, there are also many proposals studying the parallelism of deep learning inference in both specialized architectures like TPU [8] or conventional general-purpose processors [11]. However, they are more concerned with the datacenter applications and pursue high-throughput service by providing the ability of running many threads of network inference concurrently on the single chip to service many incoming queries or input data. DaDianNao [5] that connects all the tiles using a fat tree Network-on-Chip (NoC) is also oriented at data-level parallelism in server machines. These designs focus on the throughput boost by running multiple network models or multiple inputs concurrently. They are not geared towards the embedded or mobile systems that highlight the response speed of single-pass network invocation to satisfy the Quality-of-Service (QoS) constraint of a single task.

While running multiple network models or processing multiple inputs on different cores independently does not involve intensive inter-core communications [1], parallelizing single-pass network inference on CMP architectures has to handle the soaring overhead of inter-core communications. It is due to the data dependence and dense neuron connections if a single-pass network inference is partitioned and parallelized on different cores of CMP architec-ture. For example, the data communication may account for more than 30% inference latency in DaDianNao and grows up rapidly with the increase of system scale when mapping one single network on it [5].

To accelerate a single inference task on multi-core neural accelerators and reduce the communication overhead that could be prohibitively expensive, as illustrated in Fig. 1, we investigate and propose several inference parallelization techniques: (1) *Traditional parallelization*. For the target network, each core is in charge of a layer partition and routinely broadcasts its output neurons values to all the other cores to synchronize data for the process of the next layer, as indicated by dashed arrows in Fig.1 (a). This method induces intensive inter-core traffics and potenially causes performance penalty. (2) *Structure-level parallelization*. By exploiting the algorithmic resilience and redundancy in deep networks, we deliberately and slightly modify the original network without influencing the functionality of the model so that the cores do not broadcast the output neurons values in specific layers and no inter-core communications are induced consequently as Fig. 1(b) shows. (3) *Communication-aware sparsiied parallelization*. Our key observation is that the zero-value neural weights/neurons, which are deemed as sparsity in neural networks, do not affect the inference results and need not be transmitted across the cores in computation. Instead of forcing the network model to break up the intra-layer connections by design as in Fig.1 (b), this method leverages the sparsification technique to let the networks "learn" to converge on a both accurate and communication-overhead reduced structure by themselves in the training phase without modifying the network configurations as illustrated in Fig.1 (c). These techniques show some useful clues on how to speed-up deep neural network inference on embedded CMP architectures composed of either neural accelerators or even general-purpose cores like [7, 10].

This paper is supported by National Natural Science Foundation of China under grant No. (61432017, 61532017, 61504153, 61876173, 61874124). The corresponding authors are Ying Wang and Huawei Li.

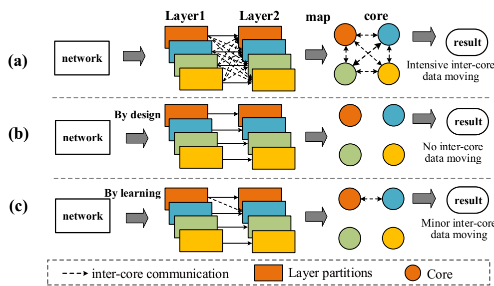

Fig. 1. (a). Traditional parallelization. (b). Structure-level parallelization. (c). Communication-aware sparsified parallelization mapped on four cores.

Specifically, this paper makes the following contributions:

- We study and propose several communication-aware approaches to parallelize the single-pass network inference on multi-core architectures.
- We propose the communication-aware network training method to reduce the inter-core data moving on multi-core neural accelerator. The proposed training method not only reduces the on-chip traffics, but also aims to reduce the communication activities between two distant cores, significantly contributing to communication cost reduction.
- We evaluate the deep neural network parallelization schemes on simulated CMP architectures integrated with specialized NNA cores, and show conspicuous performance and energy-efficiency speedup. In addition, we investigate their scalability on CMP architectures.

II. Related Work

A. Deep Learning on Multi-core Architectures

A lot of prior works are conducted in these years to accelerate Deep NNs on GPGPUs, high-end CPUs [9] and FPGAs[15]. There are also closely related researches utilizing the many-core architecture of CPUs to accelerate deep learning [11]. However, these works are conducted at the arithmetic-level, and only concern about how to accelerate the matrix multiplication operations on multi-processor architecture, but do not investigate the inference parallelization problem from the aspects of network model design as in this work.

There are also plenty of researches that parallelize the training phase of large-scale neural networks on distributed systems by exploiting the capability of multiple GPGPUs and CPUs [6]. Compared to distributed training on rack-scale machines, powerful single-chip solutions like TPU and other designs [4] are more related to our work. However, unlike our work targeted on embedded scenarios, they seek to pursue input-level parallelism by concurrently inferring many independent networks to process the incoming input data or requests on high-throughput chips.

B. Deep Learning Parallelization

The studies on neural network parallelization implementations can be categorized into two types [17] : (1) *Model parallelism.* Different workers train different parts of the model, which needs frequent communication across workers to synchronize data. (2) *Data parallelism*. Different workers train the same model on different data batches. These methods are widely applied to large distributed systems or GPUs for neural network training. For example, work [13, 14] focuses on data parallelism and work [6] concentrates on model parallelism while both techniques are adopted by work [17]. However, unlike the above model parallelization method that generally partitions the model by layers (inter-layer) and processes them in pipelined manner, our techniques partition one layer into several concurrent and independent parts, then map them to different cores, which is more viable on embedded multi-core architectures because pipelining layers with distinct hyper-parameters cause severe load-imbalance issue on cores. One important point worth mentioning is that the inter-core parallelization policies proposed in this work are orthogonal to the intra-core neural network parallelism extracted by the many PEs in single-core architectures as in [2, 4].

III. Background and Motivation

A. Chip Multi-core Neural Accelerator Architecture

We assume the tiled architecture with Network-on-Chip (NoC) is adopted to implement the multi-core neural accelerator. This CMP-like architecture comprises a number of replicated tiles connected via the NoC. A tile typically incorporates a processing core which could be general-purpose CPU or specialized accelerator core, local buffers for weight and data, and network interfaces ported to a router. Fig. 2 depicts a CMP architecture that consists of 16 neural network accelerator cores [2] connected with a Mesh NoC. The NoC is responsible for transferring on-chip data between cores and off-chip data between memory controller and cores as well. Based on this CMP-like architecture, we split and map the single-pass of CNN inference to the distributed on-chip cores to exploit the computation parallelism.

B. Motivational Study

Fig. 3 shows an example of partitioning and mapping a layer of convolutional neural network to two cores. The first convolutional layer (Conv1) contains four filters (also known as kernels) to abstract the high-level features from the input maps. The kernels can be organized in $K_x \times K_y \times N_i$ tensors, where N_i equals the number of input feature maps. The four kernels are mapped to two cores to exploit the parallelism and the kernels shown in the same color are mapped to one core.

One kernel computes with the input data will generate an output feature map. After the computation of the first layer, the generated four output feature maps are in two different cores. Thus, the following data synchronization induces the communication overhead between cores. We estimate this traditional parallelization approach using several representative neural networks on 16-core CMP-like architecture. The data volume needed to be exchanged in NoC for different layers are shown in TABLE I. As the table shows, the data size is increasing as the size of the neural network model and the input image become larger. When larger and deeper networks like VGG19 and Resnet-incept are deployed on such architectures, the partitioning-induced traffics will be rocketing. In such cases, the data packets are injected in burst during layer transition and are prone to block each other due to the limited bandwidth of the on-chip network, and eventually cause communication congestion in the NoC communication, potentially degrading the system performance. When the processors become stronger, the relative

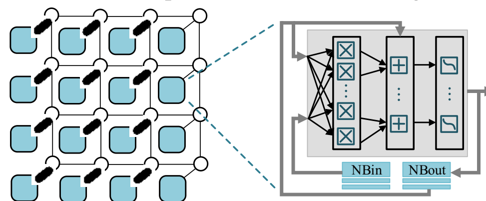

Fig. 2. Chip multi-core neural accelerator architecture.

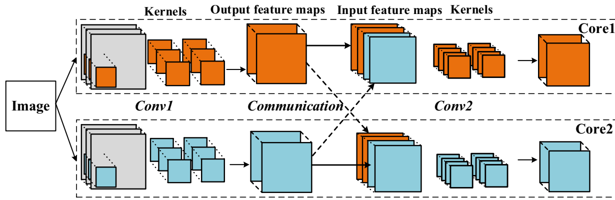

Fig. 3. Traditional parallelization of CNN inference on two cores. The dashed arrows mean data communication between cores.

TABLE I. DATA VOLUME TO TRANSMIT IN NOC AFTER LAYER PARTITIONING AND PARALLELIZATION

Net-works	Datasets	Data moving size [a] (Byte)					
		Conv2	*Conv3*	*Conv4*	*Conv5*	*Ip1*	*Ip2/3*
MLP	MNIST	-	-	-	-	28K	17K
LeNet	MNIST	225K	-	-	-	57K	29K
Convnet	Cifar-10	450K	113K	-	-	57K	-
AlexNet	ImageNet	2M	2.4M	1.8M	1.8M	450K	57K
VGG19	ImageNet	42M	22M	11M	5.4M	1.4M	57K

a. Both Conv2_1 and Conv2_2 of VGG19 are showing with Conv2.

performance penalty caused by these burst traffics tends to be more significant. According to our analysis, it costs about 23% time for AlexNet to communicate between cores during a single-pass inference on one embedded chip integrated with NNAs [2].

In conclusion, in order to accelerate the CNNs inference on latency-focused and energy-efficient mobile systems, parallelizing neural network layers on multiple cores is an effective solution to amortize the large-quantity operations, but the effectiveness is highly correlated to the inter-core communication penalty entailed by task partitioning and synchronization. In consideration of both computation and communication overhead, we evaluate several approaches that exploit the parallelism of neural network inferences in different manners, and discuss their implication on on-chip communication and system performance.

IV. CMP ORIENTED CNN INFERENCE PARALLELIZATION

In this section, we discuss three parallelization methods as briefly illustrated in Fig. 1, which are investigated and evaluated for their performance and scalability on CMP architectures.

A. Traditional Parallelization

Fig. 3 illustrates the traditional parallelization process on two cores. Suppose there are four convolution kernels divided into two groups for Conv1 layer and each group with two kernels is mapped to one core. All the cores have the same input image. When the input image is convolved with the according kernels, each core then outputs two feature maps. After that, the computation of the next layer is invoked, and the related core needs the data of the rest feature maps on the other core to start the computation of current layer. Thus, every core needs to broadcast the output feature maps of the previous layer to the other core through the NoC and receive the response data sent by the other core as well to synchronize the data, which lead to communication overhead in NoC. This method will produce the same output result as the non-parallelized network in single-core implementation. However, as the computing power and the core number scale up, the performance of NoC is likely to be the bottleneck of the whole system because of the intensive inter-core traffics.

This type of parallelism technique loyally maps the network to multi-core architectures and thus is adopted by many works on neural network acceleration [17] and state-of-the-art mathematic library like MKL [12] for speed-up in multi-core CPUs or GPGPUs. Therefore, the traditional parallelization technique is also regarded as the baseline to be optimized in this work.

B. Structure-level Parallelization

One effective method to resolve the communication problem is to seek the structure-level parallelization as shown in Fig. 4. Each core generates two output feature maps as the traditional parallelization does. However, the cores do not broadcast the feature maps to other cores for certain convolutional layers. In other words, the output neurons of the next layer are connected only to those neurons on the same core generated by the previous layer, to avoid the communication between cores. If we intentionally transform the neural networks into this partly-connected structure, there will be no necessity to transfer the activation data to certain cores. In Fig. 4, the kernels in Conv2 layer take input only from those Conv1 feature maps that reside on the same core. This design intuition is borrowed from the classic structure of AlexNet [9] that partitions some of the middle convolutional layers (the second convolutional layer for instance) into two non-interactive groups to reduce the parameters updates in training on GPUs, which is also called "grouping". Grouping is originally proposed to reduce the memory space demand in training. However, we observe that this structure modification can also be extended to reduce inter-core data moving in inference with little impact on model functionality if we partition the dense convolutional layer into independent sub-groups. However, this grouping technique is not randomly applied to all layers for accuracy sake. In general, we choose to "split" the convolutional layers with high-dimension kernels because these layers induce higher energy and penalty to transmit the intermediate computing results according to TABLE I.

For large-scale neural networks, structure-level parallelization will significantly reduce the communication and the computation complexity as well. This is because that the input data of the next layer is incomplete and the kernel size is reduced correspondingly, as a consequence, the operation amount is decreased significantly. However, this method may cause accuracy loss sometimes. A compensational solution is to replace the model with a stronger but more complex one to recompense the loss. The details are discussed in Section V.

C. Communication-aware Sparsified Parallelization

The structure-level parallelization makes it convenient to parallelize CNNs at a lower cost, and it needs experience to find the best grouping strategy about which and how many layers are split in a net. If not properly configured, it may cause accuracy loss due to inadequate feature learning and extraction. Therefore, instead of splitting the network directly, we also propose to utilize the inherent resilient training characteristic of neural networks and *let themselves "learn" the network architecture and parameters that are most suitable to be parallelized on CMPs*. We observe that neural network sparsification techniques can be used to learn this type of CMP-friendly neural networks with reduced communication between cores.

1) Overview: Pruning the over-parameterized neural networks is an efficient approach to reduce network complexity and

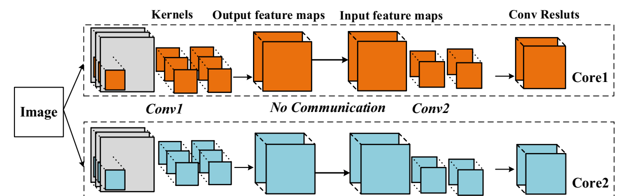

Fig. 4. Structure-level parallelization on two cores.

also the chances of over-fitting by removing redundant connections and nullifying the according neural parameters. Therefore, this work utilizes the neural network sparsification technique to obtain the desired models without influencing their functionality. For details, as Fig. 5 shows, the convolutional kernel size (N_i) is consistent with the number of the input feature maps and each feature map is convolved by the corresponding part of the kernel. As a result, when the corresponding parameters in that kernel are purposely sparsified in training to be all zero (white parts), the result of the according output feature map will eventually be zero whatever the data values of the input feature map are (shown in gray). Thus, it is unnecessary to transfer the activations that will be eventually multiplied by zeros from core to core. For example, to obtain the input data for Conv2 layer computation, the cores need to communicate with each other to obtain the output results from Conv1 layer according to traditional manner. However, if the weight kernels of Conv2 layer are so sparsified that the to-be-transmitted input feature maps are destined to generate zero-value outputs after convolution operation, then there would be unnecessary for the other core to send the previous layer results.

Based on this observation, we employ a structured sparsification method to train the networks. Compared to early non-structured sparse networks, which are pruned independently and without a specific constraint and the trained models are possessing randomly distributed zero-value weights, structured sparsity approaches purposely distribute the non-zero weights at pre-assigned locations, so that the zero-value neural weights or activations could be more hardware-friendly for acceleration [16]. This work uses structured sparsification technique, specifically group Lasso, to obtain a communication-reduced network model with regularized weights distribution (e.g., the kernels of Conv2 layer in Fig. 5) for embedded multi-core accelerators.

In addition, since the communication overhead between two arbitrary cores depends on their Hamming Distance from each other in NoC with Mesh topology, the system performance will be influenced by the distribution of non-zero parameters in a kernel because they determine which node in this NoC a certain feature map will be sent to. Thus, we further propose communication-aware sparsified parallelization technique that takes the Hamming Distances between cores into consideration to train the CNN models that have the best zero-weight distribution for CMP-like architecture performance.

2) Group Lasso Regularization: Specifically, we use group Lasso regularization to learn structured sparsification for CNNs models [16]. The corresponding optimization target in training can be formulated as:

$$L(W) = L_D(W) + \lambda \cdot R(W) + \lambda_g \cdot \sum_{l=1}^{l} R_g(W^l) \tag{1}$$

here, W is the collection of all weights in the DNN and $L_D(W)$represents the loss on data as specified by the prediction task. $R(\cdot)$ is the generic regularization term applied to every weight that are non-structured, such as l_2-norm. $R_g(\cdot)$ is the structured sparsity regularization term on each layer and the group Lasso regularization method makes it possible that any specified group of weights in the whole network are more likely to be zero than weights in other locations. Then the weights distribution is

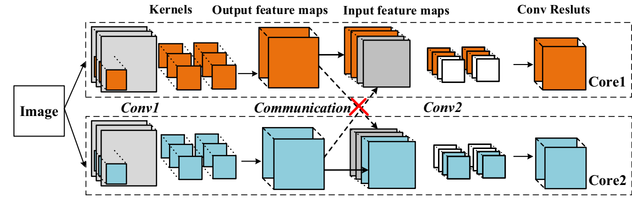

Fig. 5. Communication-aware sparsified parallelization on two cores, kernels and feature maps in different color indicate they are mapped to different cores while kernel parts in white are automatically zeroed in training.

changed and re-structured at group-level. The regularization of group Lasso on a group of weights can be represented as:

$$R_g(W) = \sum_{g=1}^{G} \| w^g \|_g \tag{2}$$

where $\|\cdot\|_g$ is the group Lasso, w^g is a group of weights in w and G is the total number of groups. We formulate $\|\cdot\|_g$ as:

$$\| w^g \|_g = \sqrt{\sum_{i=1}^{|w^g|} (w_i^g)^2} \tag{3}$$

where $| w^g |$ is the number of weights in w^g .

3) Detailed Methodology: We firstly partition the weight matrix into several groups of the same number as the square of the core number. Then, we use the distances between cores as a factor to influence the learning process by assigning different sparsity strength[1] to the weights according to their involved communication cost. Therefore, we obtain a model with few data moving activities between two distant cores in NoC, which is measured with data volume and core distance (decided by routing algorithm).

Suppose the chip has 16 cores as shown in Fig. 2, and the distance between two adjacent cores is 1. The distances of the first four cores are as Fig. 6 (a) shows, given that dimensional-ordered routing is adopted. We use this 16×16 distance matrix as the factor mask to sparsify the weight groups during training, so the groups of parameters will be sparsified with the priority as specified in the factor matrix. Consequently, the parameters that induce higher communication overhead will be the first to be pruned. For example, the weights on the diagonal groups will not cause any communication between any two cores because the corresponding data are on their own cores. Therefore, we assign lower sparsity strength to these groups to keep their values while pruning those groups that cause high communication overhead because of long distance.

Finally, we obtain a structured and optimized communication-aware network model, which is more energy-efficient than the baseline in terms of on-chip parallelization cost. The theory behind this technique is that in network training there are various choices of weight values that all make the model generate an accurate prediction, which is also called redundancy in network. Fig. 6 (b) shows an example of the final grouped weights matrix obtained in experiments.

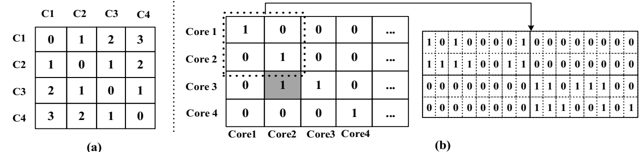

Fig. 6. (a). Distance for 4 cores. (b). Final weights matrix in group-level.

[1] The strength to sparsify the weights, the higher, the weights are more likely to be pruned down.

Each line is mapped to a core which has two convolutional kernels sized as 2×2×32. The two kernels in one core are further divided into 16 groups (Fig. 6 (b) only shows the first four groups). The number 1 represents that the value is not zero. The shaded area means the group sparsification is unnecessary and the communication cost caused by this block of weights is marginal because the two cores are adjacent to each other (only one hop from core3 to core2), so that the training procedure will freely parameterize groups like this to preserve accuracy. The figure shows that all the groups of weights that induce long-distance communication are pruned away as an ideal case.

V. Experimental Evaluation

The experimental platform is a simulated embedded CMP with 16 cores connected with a mesh NoC as shown in TABLE II. Two variants of ConvNet are used to evaluate the effectiveness of the structure-level parallelization by varying the kernel numbers of the convolutional layers on ImageNet10 (images containing ten object classes of ILSVRC 2012) as shown in TABLE III. In addition, we choose several representative neural network models, including MLP and LeNet on MNIST, ConvNet on Cifar-10, and AlexNet on ImageNet as the benchmarking networks for communication-aware sparsified parallelization. MLP has three fully-connected layers with the neuron number of 512/304/10 respectively. The inference performances of networks that are parallelized with the traditional manner are used as the baselines for comparison. The neural cores are simulated with an in-house simulator that could faithfully simulate the design of Diannao [2]. BookSim2 and DSENT are used to simulate the NoC communication process. We also test the sensitivity of the proposals to core number in experiments to investigate their scalability on CMP-like architectures.

A. Performance and Energy Evaluation

1) Structure-level parallelization: As TABLE III shows, Parallel#1 and Parallel#2 are two different implementations of the same ConvNet variant, and they are parallelized using different grouping methods. To evaluate this technique, we divide the convolutional kernels in Conv2 and Conv3 layers of these two neural networks into *n* groups so that each group could be mapped to one core for inference. When n equals 1 as in Parallel#1, it indicates that the network is parallelized in a traditional way, which is used as the baseline.

In Fig. 7, the metric of performance speedup gauges both the speedup of computation and the NoC transmission in CNN inference caused by network grouping. The normalized communication performance speedup measures the ratio between the computation-blocking communication cycles of the baseline and that of the proposed structure-level parallelization. Fig. 7 shows that the structure-level parallelization accelerates both computation and communication for the reduction of the computing complexity and communication (Conv2 and Conv3 layers do not need data transmission according to Fig. 4). However, this method eventually results in 2.8% accuracy loss. By adding the kernel number, we obtain a more accurate model, which is Parallel#3. Although Parallel#3 entails more operations due to the increased convolutional filters, the experimental results show that the system performance speedup is still higher than the baseline result (4.6×) with 1.6% increased accuracy. The overall energy consumption reductions are 91% and 88% for Parallel#2 and Parallel#3 respectively.

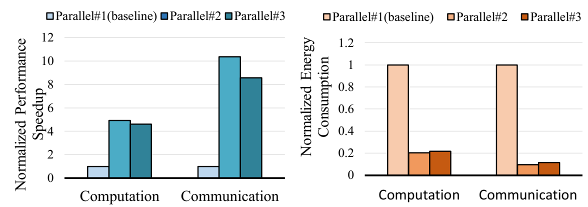

Fig. 7. System performance speedup (left) and communication energy reduction (right) for structure-level parallelization.

2) Communication-aware sparsified parallelization: We train four different neural networks for performance and energy evaluation. It is noted that because this method is deployed in training phase and the trained CMP-friendly neural network model is already prepared when enabling inference, therefore, no extra overhead is induced. The experimental results shown in TABLE IV demonstrate that our proposed method is very effective for the parallelization of both fully-connected layers (MLP) and large-scale convolutional layers (CaffeNet, Caffe-provided version of AlexNet with minor changes). The scheme named SS are also structured-sparsified with group Lasso and the neural connections in the same layer of SS method share the same sparsity strength factor without considering the distance between cores in training phase. The baselines are the same network implementations without structured sparsification. Compared to SS, SS_Mask is fully aware of the inter-core communication cost in NoC by using the sparsity mask matrices that reflect the locations of cores in Mesh. These mask matrices are used to impose different levels of sparsity strength factor on different groups of weights according to the potential communication overhead decided by data moving distance.

TABLE II. System Configurations

Component	Parameters
Specialized accelerator core [2]	Each core has：16×16 PEs, one 128KB weight buffer per core, two 32KB data buffers per core, 16-bit fixed-point integer operation
Network-on-chip	512-bit flit, 20-flit packet, 2D mesh topology, 3 stages, 2 physical channels, dimensional-ordered routing, 3 VCs
Main memory	1 channel, 1 rank, LPDDR3, 1GB, 4-bank

TABLE III. Performance of Structure-level Parallelization

ConvNet	Conv kernel number [a]	Group num. (*n*)	Accu.	Speedup
Parallel#1	64 – 128 – 256	1	0.726	1
Parallel#2	64 – 128 – 256	16	0.698	**4.9×**
Parallel#3	64 – 160 – 320	16	0.742	**4.6×**

[a] In the order of conv1-conv2-conv3.

TABLE IV. Performance and Energy Reduction of Communication-aware Sparsified Parallelization

Networks	Type	Accu.	NoC traffic rate	System speedup	Energy Reduction
MLP	Baseline	98.36%	100%	1×	0%
	SS	98.38%	30%	1.40×	59%
	SS_Mask	**98.36%**	**11%**	**1.59×**	**81%**
LeNet	Baseline	99.17%	100%	1×	0%
	SS	98.98%	82%	1.20×	15%
	SS_Mask	**98.60%**	**23%**	**1.51×**	**89%**
ConvNet	Baseline	78.75%	100%	1×	0%
	SS	80.15%	46%	1.19×	25%
	SS_Mask	**79.61%**	**35%**	**1.32×**	**55%**
CaffeNet	Baseline	55.19%	100%	1×	0%
	SS	55.02%	98%	1.02×	17%
	SS_Mask	**54.21%**	**57%**	**1.10×**	**38%**

TABLE IV shows that our proposed parallelization method significantly reduces the size of NoC traffics. For example, we reduce the size of transmitted packets to only 11% on average for MLP without any accuracy loss. This method significantly speeds up the system performance and reduces the NoC energy consumption due to the large amount of NoC traffics reduction. The SS_Mask scheme performs better than SS because the NoC traffics in SS_Mask are constrained to be between only close cores one or two hops away from each other.

B. Sensitivity to Core Number

All the above experiments are conducted on a 16-core chip. We test the sensitivity of the proposals to core number in experiments.

1) Structure-level parallelization: We evaluate the system performance and energy consumption of Parallel#3 in TABLE III on chips with 4, 8 ,32 cores respectively. The corresponding networks are listed in TABLE V. As Fig. 8 shows, the system performance speedup and energy efficiency for computation increase with the core number because of the declining computing complexity on each core. The system performance improvements and energy efficiency for communication remain steady. The reason is that the overall volume of moving data keeps stable, but both the average core-to-core distance and the bi-sectional bandwidth of the on-chip network increase at the same time. The latter two factors bring the NoC performance to different directions but finally make a balance point as core number increases. However, relative to computation performance, the communication cost becomes more pronounced and the proposed parallelization techniques become more useful.

2) Communication-aware sparsified parallelization: We use LeNet to test the communication-aware sparsified parallelization on 8 cores and 32 cores respectively. The models as well as the corresponding system performance speedups and communication energy reductions are listed in TABLE VI. It shows that the system performance speedups and energy reductions increase as the core number scales up. This is partly because that as the core number increased, the size of every kernel group assigned to each core becomes smaller and can be easily pruned out at a lower risk of accuracy loss. Another reason for the relative speedup is that the NoC bandwidth issue becomes relatively more severe as the diameter and node number of the NoC increase, which leaves a larger space of performance improvement for our method to exploit.

TABLE V. PERFORMANCE OF STRUCTURE-LEVEL PARALLELIZATION FOR PARALLEL#3 ON VARIOUS NUMBER OF CORES

Core number	*n*	Accu.	Speedup
4	4	0.694	2.7×
8	8	0.718	4.6×
16	16	0.742	6.0×
32	32	0.722	6.9×

TABLE VI. PERFORMANCE AND ENERGY REDUCTION OF COMMUNICATION-AWARE SPARSIFIED PARALLELIZATION OF LENET FOR 8 AND 32 CORES

LeNet	Type	Accu.	NoC traffic rate	System speedup	Energy Reduction
8core-1	Baseline	99.1%	100%	1×	0%
8core-2	SS	98.9%	80%	1.20×	10%
8core-3	SS_Mask	98.9%	68%	1.22×	32%
32core-1	Baseline	99.1%	100%	1×	0%
32core-2	SS	98.7%	32%	1.49×	34%
32core-3	SS_Mask	98.6%	18%	1.58×	56%

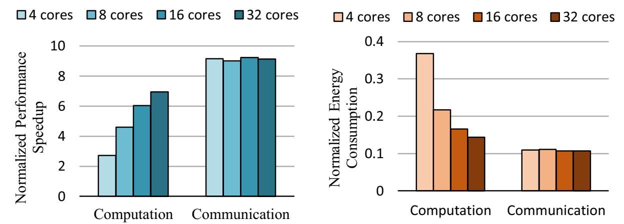

Figure 8: System performance speedup (left) and communication energy consumption (right) of various cores for structure-level parallelization.

VI. CONCLUSION

In this work, we evaluate several unorthodox parallelization schemes for single-pass neural network inference on-chip multi-core architectures. Among them, the communication-aware sparsified parallelization technique exploits the elasticity and noise-tolerance features of deep learning algorithms to enable the neural network to learn a configuration that is very suitable to be parallelized on CMP architectures. The experimental results show that this method improves the system performance by 1.6×-1.1× and achieves 4×-1.6× better on-chip communication energy efficiency for different neural networks.

REFERENCES

[1] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in ACM SIGARCH Computer Architecture News, 2016, vol. 44, no. 3, pp. 1-13: IEEE Press.

[2] T. Chen, *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in ACM Sigplan Notices, 2014, pp. 269-284: ACM.

[3] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks," arXiv preprint arXiv:1807.07928, 2018.

[4] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," IEEE Journal of Solid-State Circuits, pp. 127-138, 2017.

[5] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, and N. Sun, "Dadiannao: A machine-learning supercomputer," in Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014, pp. 609-622: IEEE Computer Society.

[6] A. Coates, *et al.*, "Deep learning with COTS HPC systems," in International Conference on Machine Learning, 2013, pp. 1337-1345.

[7] L. Gwennap, "Adapteva: More flops, less watts," Microprocessor Report, vol. 6, no. 13, pp. 11-02, 2011.

[8] N. P. Jouppi, *et al.*, "In-datacenter performance analysis of a tensor processing unit," arXiv:1704.04760, 2017.

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

[10] Y. Lee, *et al.*, "A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators," in European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th, 2014, pp. 199-202: IEEE.

[11] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in Proceedings of the 27th international ACM conference on International conference on supercomputing, 2013, pp. 273-282: ACM.

[12] MKL, pp. Math Kernel Library. https://software.intel.com/en-us/mkl.

[13] F. Seide, *et al.*, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns," in Fifteenth Annual Conference of the International Speech Communication Association, 2014.

[14] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li, "C-brain: a deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization," in Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE, 2016, pp. 1-6: IEEE.

[15] Y. Wang, *et al.*, "DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family," in Proceedings of the 53rd Annual Design Automation Conference, 2016, p. 110: ACM.

[16] W. Wen, *et al.* "Learning structured sparsity in deep neural networks," in Advances in Neural Information Processing Systems, 2016, pp. 2074-2082.

[17] F. Yan, O. Ruwase, Y. He, and T. Chilimbi, "Performance modeling and scalability optimization of distributed deep learning systems," in Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2015, pp. 1355-1364: ACM.