This is the accepted manuscript of a paper published in
Proc. 4th Int'l Symp. on Computing and Networking (CANDAR'16)
Copyright (C) 2016 IEEE

Evaluation of Task Mapping on Multicore Neural Network Accelerators

Satoshi SHINDO*, Momoka OHBA†, Tomoaki TSUMURA*, and Shinobu MIWA†
*Nagoya Institute of Technology, Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp
†The University of Electro-Communications, 1-5-1 Chofugaoka, Chofu, Tokyo, Japan

***Abstract*—Deep neural networks are widely used for many applications such as image classification, speech recognition and natural language processing because of their high recognition rate. Since general-purpose processors such as CPUs and GPUs are not energy efficient for such neural networks, application-specific hardware accelerators for neural networks (a.k.a. neural network accelerators or NNAs) have been proposed to improve the energy efficiency. There are many studies to increase the energy efficiency of NNAs, but few studies focus on task allocation on the accelerators. This paper provides the first exploration of task mapping to cores within NNAs for the increased performance. Intuitively, a well-tuned task mapping has less amount of communication between cores. To confirm this assumption, we tested two types of task mappings that generate different amount of communication between cores on an NNA. Our experimental results show that the number of communication between cores strongly affects the execution cycle of the NNA and the most effective task mapping differs depending on the size of neural networks.**

I. Introduction

Artificial neural networks are widely used for machine learning applications such as image classification, speech recognition and natural language processing. Recent machine learning applications use neural networks having many layers, *Deep Neural Networks (DNNs),* to achieve the high recognition rate. For example, a DNN composed of twenty layers showed the highest recognition rate in the image classification competition of ILSVRC-2014 (ImageNet Large Scale Visual Recognition Challenge) [1]. Machine learning for DNNs is called *Deep Learning*, and DNNs are growing deep and wide to improve the recognition rate. However, computing DNNs on general-purpose processors (e.g., CPUs and GPUs) is time-consuming and energy inefficient [2], [3].

For the increased performance and energy efficiency, hardware accelerators for neural networks (a.k.a. neural network accelerators or NNAs) have been proposed [4], [5], [6], [7], [8], [9]. NNAs are highly customized to compute neural networks and therefore show the significant improvement in the performance and energy when compared to general-purpose processors. For example, a state-of-the-art NNA achieves both 3x speedup and 100x energy efficiency when compared to a GPU [8]. Although many studies to increase the energy efficiency of NNAs have been conducted, few studies focus on tuning computational tasks on the accelerators.

This paper provides the first exploration of task mapping to cores within NNAs for the increased performance. Intuitively, a well-tuned task mapping has less amount of communication between cores. To confirm this assumption, we tested two types of task mappings with different communication patterns on an NNA and found that the number of communication between cores proportionally impacted on the execution cycle of the accelerator.

II. Research Background

In this section, we overview artificial neural networks and existing NNAs.

A. Neural Networks

Artificial neural networks are information processing models inspired by biological nervous systems. They are comprised of many computational units (*neurons*) and connections between them (*synapses*). The most general neural network is composed of multiple layers with each having some neurons, and neuron within a layer is connected to some neurons within the next layer.

Figure 1 illustrates an example of multi-layer neural networks. This multi-layer neural network has three layers: input, hidden, and output layers. In the multi-layer neural network, signals received at the input layer propagate toward the output layer through the synapses. A layer in which each neuron is connected from all neurons in the previous layer is called a fully-connected layer. Each synapse i has its own value called *weight* w_i $(i = 1, ..., n)$. A neuron's output y is defined as follows;

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \quad (1)$$

where x_i $(i = 1, ..., n)$ represents input signals for the neuron and b is a constant value called *bias*. The function f is called an *activation function*. ReLU [10] which is often used as an activation function in DNNs is defined as follows;

$$f(x) = max(0, x) \quad (2)$$

We define the computation for one neuron (i.e., the computation of y) as a *task*. More specifically, a task consists of a multiply-add operation with input signals and synaptic weights, and the computation for the activation function.

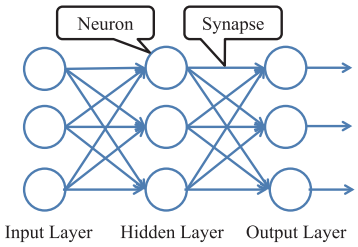


Fig. 1. A multi-layer neural network.

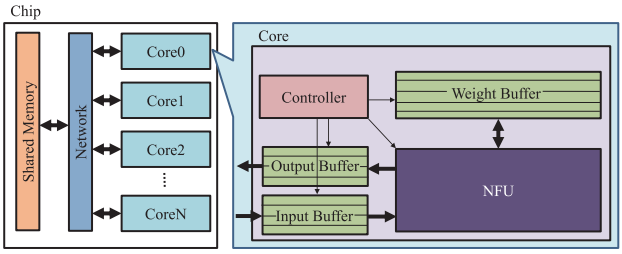


Fig. 2. A multicore neural network accelerator.

Computing output signals of neurons from their input signals is called *inference*, and adjusting synaptic weights to correct the output signals is called *training*.

B. Related Work

Various NNAs have been proposed, and they can be classified into two types. The first type of accelerators has many processing elements that are specialized to compute output signals of neurons. Each neuron in a neural network is mapped to its own processing element. For example, Merolla et al. [11] have proposed an accelerator having 256 processing elements, which can simulate a neural network composed of up to 256 neurons. Similarly, Temam et al. [12] have proposed an accelerator composed of many processing elements, which connect each other through some buffers to keep synaptic weights. In these accelerators, a processing element starts to compute the output signal of the corresponding neuron as soon as the input signals have arrived, and then sends the computed output signal to the neighboring processing elements. Because these operations are simple, the above accelerators need not have any control logic or instruction set. Besides, these accelerators enable a processing element to fast load the corresponding synaptic weights because they are stored in the buffers close to the processing element. Although these accelerators provide high-speed inference for some neural networks, they have no capability to simulate neural networks where the number of neurons exceeds the number of processing elements.

The second type of accelerators has a small number of processing elements or cores with each computing outputs of multiple neurons, and therefore has the capability to simulate various scales of neural networks. For example, Chen et al. [5] have proposed DianNao, which is a single-core NNA that executes programs composed of dedicated instructions. Each instruction computes outputs of multiple neurons in parallel. DianNao can simulate various scales of neural networks depending on a program running on the core. Chen et al. have also proposed DaDianNao [4], a multicore accelerator which is an enhanced model of DianNao. DaDianNao can simulate a larger scale of neural networks than DianNao because of the increased memories for keeping input/output signals and synaptic weights. Similar to general-purpose multicore processors, the performance of DaDianNao is expected to depend on how to map computational tasks to cores, but it has not been discussed in the paper [4].

III. Multicore Neural Network Accelerators

In this section, we describe multicore NNAs including DaDianNao in detail.

A. Architecture

Figure 2 illustrates the architecture of a multicore NNA including DaDianNao. The accelerator consists of a Shared Memory and multiple cores. The Shared Memory stores input and output signals of neurons. All cores can read and write data on the Shared Memory through the on-chip network (not shown in Fig. 2).

A core consists of a controller, an NFU (Neural Functional Unit), and three buffers: an Input Buffer, a Weight Buffer, and an Output Buffer. The Input and Output Buffers store input and output signals of neurons, respectively, and the Weight Buffer stores synaptic weights. The NFU is a processing unit for tasks, which consists of multiply-add and activation function logic. The controller manages three buffers and the NFU, depending on a dedicated instruction which is read from a per-core instruction memory (not shown in Fig. 2).

The Input and Output Buffers retain a part of the input and output signals on the Shared Memory, respectively. An input signal read from the Shared Memory is temporarily stored in the Input Buffer. An output signal computed on the NFU is temporarily stored in the Output Buffer and then written back to the Shared Memory. The NFU can reuse the input and output signals on two buffers for another task, and two buffers thus help to reduce the pressure of the Shared Memory.

The Shared Memory and three buffers deal with a bunch of data as a group, which is called a **chunk**. A core can simultaneously read or write the data within a chunk on the Shared Memory, and the NFU can execute the tasks for the data in parallel. In this paper, we define the number of input signals simultaneously readable from the Shared Memory as N_i, and the number of output signals simultaneously writable to the Shared Memory as N_o.

Here, we explain how to deal with input and output signals and synaptic weights in the multicore NNA. Figure 3 shows an example of data format used in the accelerator. This example shows an input vector including 8 input signals ($I_1 ... I_8$), a 12×8 weight matrix ($w_{ij} : 1 \leq i \leq 12, 1 \leq j \leq 8$), and an output vector including 12 output signals ($O_1 ... O_{12}$). The output vector is given by the matrix multiplication of the input vector and the weight matrix. For example, the input signals $I_1 ... I_8$ are multiplied by the weight vector w_{ij} $(j = 1, ..., 8)$,

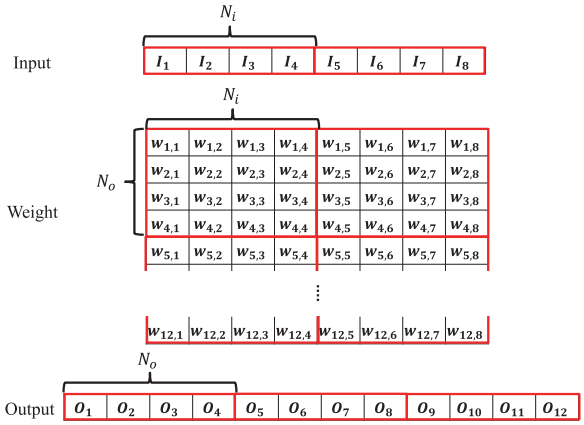
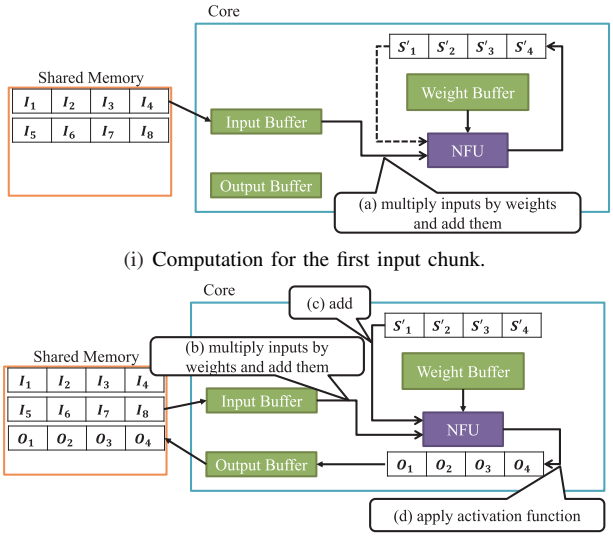


Fig. 3. Data format in the multicore neural network accelerator.



(i) Computation for the first input chunk.

(ii) Computation for the second input chunk.

Fig. 4. Computation of one output chunk with two input chunks.

and this results in the output signal O_i. The input and output vectors, and weight matrix are partitioned into multiple chunks in the accelerator. If $N_i = N_o = 4$, four input and output signals, and 4×4 synaptic weights, which are surrounded by the red lines in the figure, are grouped, respectively.

The Input and Output Buffers have the limited number of entries (e.g., 16 entries) with each holding one chunk. As mentioned earlier, the core can reuse a chunk on the Input and Output Buffer for another task. It means that the core does not need to read a chunk from the Shared Memory if it resides in either Input or Output Buffer. A buffer has to discard an existing chunk when the number of incoming chunks exceeds the number of entries. If the core requires the chunk discarded from the Input Buffer, it has to be re-read from the Shared Memory.

B. Operations

This section describes how cores execute tasks in the multicore NNA. Figure 4 illustrates an example where the accelerator computes the tasks shown in Fig. 3. As mentioned earlier, we assume $N_i = N_o = 4$, so that four input and output signals are grouped in the Shared Memory. For example, the eight input signals $I_1 ... I_8$ are partitioned into two chunks ($I_1 ... I_4$ and $I_5 ... I_8$) on the Shared Memory. The figure shows the case where the core computes the four output signals $O_1 ... O_4$ in parallel. Since the eight input signals are partitioned into two chunks, the core needs to read these chunks from the Shared Memory sequentially. Also, the NFU conducts the multiply-add operation for the above chunks sequentially. The top figure (i) illustrates the computation for the first input chunk, while the bottom figure (ii) illustrates the computation for the second input chunk.

The core first reads the input signals $I_1 ... I_4$ from the Shared Memory and then stores them in the Input Buffer. Next, the NFU reads the input signals from the Input Buffer. At the same time, the NFU reads the corresponding weight matrix ($w_{ij} : 1 \leq i \leq 4, 1 \leq j \leq 4$) from the Weight Buffer. After that, the NFU conducts the matrix multiply-add operation with the input signals and the corresponding weight matrix (a), and it results in the partial sums which are labeled as $S'_1 ... S'_4$.

Next, the core reads the next input signals $I_5 ... I_8$ from the Shared Memory and then sends them to the NFU via the Input Buffer. The NFU conducts the matrix multiply-add operation with the partial input signals and the corresponding weight matrix (b), and then adds the result to the partial sums $S'_1 ... S'_4$ to get the final sums (c). Next, the NFU applies the activation functions to the computational results (d), and this results in the output signals $O_1 ... O_4$. Finally, the core stores the output signals to the Shared Memory via the Output Buffer. The other output chunks (i.e., $O_5 ... O_8$ and $O_9 ... O_{12}$) can be computed in the similar manner.

The tasks within a layer can be computed in parallel because there are only read-after-read memory dependencies between them. The outputs of the tasks are used as the inputs for the tasks within the next layer. Therefore, all cores store the outputs of the tasks in the Shared Memory. A core reads these outputs from the Shared Memory when it executes the tasks for the next layer.

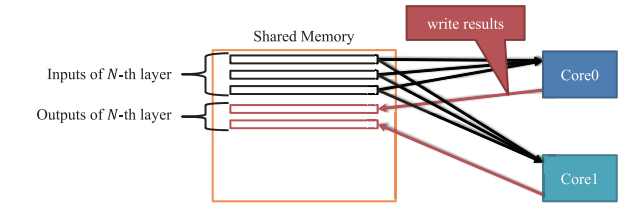
Figure 5 illustrates an example of communication between cores with the Shared Memory. Figure 5 (i) represents the case where two cores execute the tasks for the N-th layer, while (ii) represents the case where both cores execute the tasks for the $(N+1)$-th layer. First, when computing the tasks for the N-th layer, each core writes the computational result into the Shared Memory (i). Next, each core reads the result from the Shared Memory before computing the tasks for the $(N+1)$-th layer (ii). This enables cores to share outputs of tasks in the multicore NNA.

IV. Task Mapping for Multicore NNAs

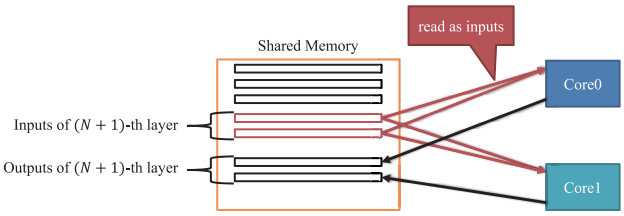
In this section, we describe two task mapping methods in multicore NNAs.

A. Overview

As explained in Section III, data transfer from a core to another core via the Shared Memory is needed if the latter



(i) Execution of tasks for the N th layer.



(ii) Execution of tasks for the $(N+1)$-th layer.

Fig. 5. Inter-core communication via the Shared Memory

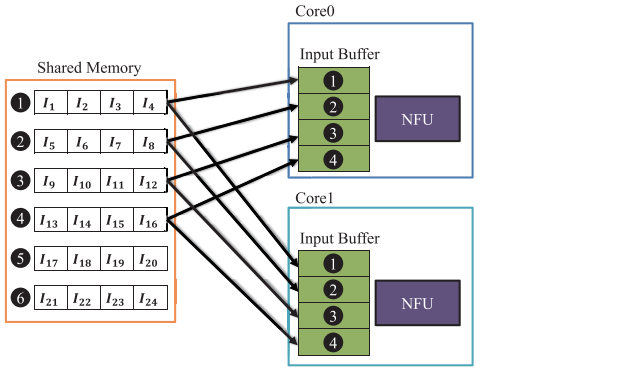
core requires the results computed by the former core. This data transfer can be regarded as inter-core communication. Since the number of inter-core communications can affect the performance of a multicore NNA, this paper investigates its impact on the performance by using two task mappings with different communication patterns. The first task mapping is to group output signals of neurons in a layer and assign all tasks within a group to a core. The second task mapping is to group input signals of neurons in a layer and assign all computations with a group to a core. We explain these two mappings in the following two subsections.

B. Task Mapping Based on Output Signals

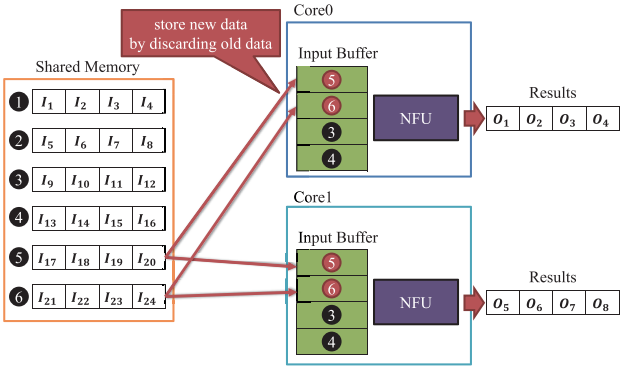
First, we explain task mapping based on output signals of neurons. In this task mapping, many tasks are mapped into one core. Some cores may need to read the same input signals to compute the assigned tasks, but all tasks assigned to a core require no additional calculation or communication.

Figure 6 shows an example of task mapping based on output signals of neurons. The figure represents the case where two cores compute eight output signals of neurons ($O_1...O_8$) with each receiving twenty-four input signals ($I_1...I_{24}$). We assume $N_o = N_i = 4$ in this example, so that the input signals $I_1...I_{24}$ are partitioned into six chunks and stored in the Shared Memory. Additionally, the eight output signals are divided into two groups (i.e., $O_1...O_4$ and $O_5...O_8$), and the tasks for $O_1...O_4$ and $O_5...O_8$ are allocated to Core0 and Core1, respectively.

The tasks are computed as follows. First, both cores read the same input chunk of $I_1...I_4$ from the Shared Memory, and then compute the partial results for the assigned output signals in parallel. Next, both read the next input chunk of $I_5...I_8$ and then compute the new partial sums. Then, the newly-computed partial sums are added to the first partial sums in each core. Two cores repeat such operations for the remaining four input chunks of $I_9...I_{24}$, and this results in the matrix multiply-add with all input signals and corresponding synaptic weights.



(i) Computation for the first four input chunks.



(ii) Computation for the remaining two input chunks.

Fig. 6. Task mapping based on output signals.

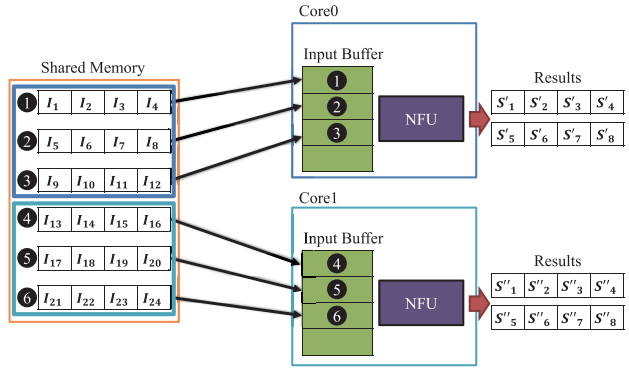
Finally, each core computes the own output signals ($O_1...O_4$ or $O_5...O_8$) with the result and activation function, and then writes the output signals into the Shared Memory.

In summary, this task mapping needs each core to read all input signals from the Shared Memory, but not to write any partial sums into the Shared Memory.

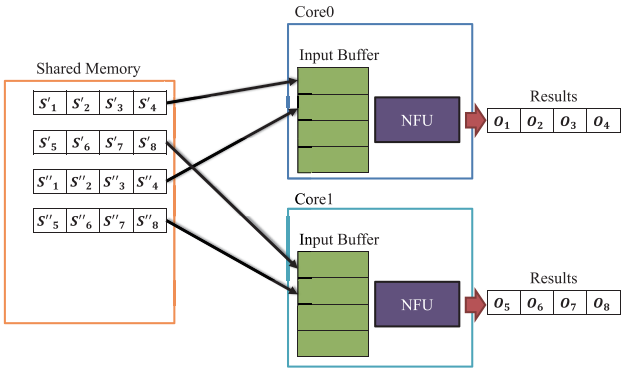
This task mapping has one problem that it is difficult for a core to reuse the input chunks in the Input Buffer. This is because the core sequentially reads all input chunks in the Shared Memory. The example shown in Fig. 6 assumes that each Input Buffer has four entries, so that all entries of the buffer are filled up after the first four input chunks are transferred from the Shared Memory (i). When a core tries to read the fifth and the sixth input chunks from the Shared Memory (ii), the Input Buffer has no empty entry so that two input chunks (e.g., the first and second input chunks) in the buffer have to be discarded. Thus, if the number of the input chunks of a neuron is bigger than the number of entries of the Input Buffer, a core is difficult to reuse the input chunks in the Input Buffer under this task mapping.

C. Task Mapping Based on Input Signals

Second, we explain task mapping based on input signals of neurons. This mapping needs cores to compute tasks in two steps. In the first step, the input signals in the Shared Memory are divided into some groups, and a core then computes partial results of multiply-add with one of the groups. The



(i) First step: computation of partial results with assigned input signals.



(ii) Second step: integration of partial results.

Fig. 7. Task mapping based on input signals.

core temporarily stores the result in the Shared Memory. In the second step, cores compute output signals with the stored partial sums and activate function.

Figure 7 shows an example of task mapping based on input signals. Note that the tasks and cores used in the figure are the same as those used in Fig. 6.

In the first step, the six input chunks on the Shared Memory are divided into two groups and each group is then assigned to a core (i). In this example, the first group (i.e., $I_1...I_{12}$) is assigned to Core0, while the second group (i.e., $I_{13}...I_{24}$) is assigned to Core1. Each core sequentially reads the assigned input chunks and then computes all partilal results of multiply-add with reading the input chunks. For example, Core0 computes the partial results of multiply-add, which are labeled as $S'_1...S'_8$, with the input signals $I_1...I_{12}$ and the weight matrix. Similar to this, Core1 computes the partial results of multiply-add, which are labeled as $S''_1...S''_8$, with the input signals $I_{13}...I_{24}$. These partial results are temporarily stored in the Shared Memory.

Note that each core can reuse the input chunks on the Input Buffer with this task mapping. This is because the Input Buffer composed of four entries can retain all three input chunks in a group. For example, Core0 need not read the input signals $I_1...I_{12}$ from the Shared Memory to compute $S'_5...S'_8$ because these input signals are already stored in the Input Buffer at the computation of $S'_1...S'_4$.

In the second step, both cores compute output signals with these partial results in parallel (ii). In this example, Core0 computes the output signals $O_1...O_4$, while Core1 computes the output signals $O_5...O_8$. Each core reads the corresponding partial results from the Shared Memory, and then sum them up. After that, the core computes the assigned output signals with the result and activate function, and then writes the output signals into the Shared Memory.

In summary, this task mapping enables cores to reduce the number of input-chunk reads from the Shared Memory to the number of input chunks, at the cost of some reads and writes for the partial results. It can be achieved if the number of input chunks in a group is smaller than the number of entries in the Input Buffer.

V. Evaluation

In this section, we show the experimental result of two task mappings on a multicore NNA, and discuss their impact on the performance.

A. Experimental Setup

The two task mapping methods described in Section IV are evaluated with our in-house cycle-level simulator, which carefully models multicore NNAs including DaDianNao. We confirmed that our simulator had a good ability to reproduce the results presented in the paper [4], and this experimental result is shown in the paper [13].

The detailed configuration of the simulated accelerator is shown in TABLE I. The number of cores is 16. The parameters including N_i, N_o, and the number of entries of Input and Output Buffers are modeled on DaDianNao [4].

We evaluated execution cycles to process one fully connected layer in multi-layer neural networks. In order to investigate the impact of layer size, we used three layers: FC1 (512 neurons, 512 input signals), FC2 (1280, 1280), and FC3 (2560, 2560).

Since our accelerator has 16 cores, task mapping based on output signals divides the neurons in each layer into 16 groups, with each being assigned to a core. Similar to this, task mapping based on input signals divides the input signals in each layer into 16 groups, with each being assigned to a core. Therefore, tasks assigned to cores are balanced regardless of the task mapping methods.

To investigate the impact of data reuse in Input Buffers, we measured execution cycles in two cases where the data reuse function is validated and invalidated, respectively. All input signals are stored in the Shared Memory before the simulation starts. It means that our experiment ignores the time to generate input signals.

B. Evaluation Result

We tested the following five configurations for FC1...FC3.

(B) all tasks assigned to a single core (baseline)

(T1) task mapping based on output signals when invalidating the data reuse in Input Buffers (run on 16 cores)

TABLE I
CONFIGURATION OF THE ACCELERATOR

Chip		
Number of cores	16	cores
Clock frequency	606	MHz
NFU latency	1	cycle
On-chip network latency	1	cycle
On-chip network bandwidth	100	GB/s
N_i	16	neurons
N_o	16	neurons
Core		
Input Buffer	16	entries
Output Buffer	16	entries
Shared Memory		
Latency	10	cycles
Number of ports	16	ports

TABLE II
THE NUMBER OF PER-CORE SHARED MEMORY ACCESSES

	FC1		FC2		FC3	
	Read	Write	Read	Write	Read	Write
(B)	1056	32	6480	80	25760	160
(T1)	66	2	405	5	1610	10
(TR1)	66	2	405	5	1610	10
(T2)	98	34	485	85	1770	170
(TR2)	36	34	90	85	180	170

(TR1) task mapping based on output signals when validating the data reuse in Input Buffers (run on 16 cores)
(T2) task mapping based on input signals when invalidating the data reuse in Input Buffers (run on 16 cores)
(TR2) task mapping based on input signals when validating the data reuse in Input Buffers (run on 16 cores)

Figure 8 shows the number of execution cycles, while TABLE II summarizes the number of per-core Shared Memory accesses. Note that all cores have the same number of Shared Memory accesses because tasks are equally allocated to cores.

The figure shows two important aspects for task mapping on the multicore NNA. First, the 16-core execution (T1), (TR1), (T2), and (TR2) drastically improves the performance of the accelerator for all layers when compared to the single-core execution (B). Therefore, task mapping is very important to maximize the performance of the multicore NNA. Second, the configuration (TR1) is faster than the configurations (T2) and (TR2) for FC1, while the latter configurations are faster than the former configuration for FC2 and FC3. This means that the most effective task mapping differs depending on the number of neurons and inputs.

C. Discussion

First, we focus on (T1) and (TR1). Figure 8 indicates that (T1) and (T2) shows the same number of the execution cycles for all layers. In addition, the number of per-core Shared Memory accesses in (T1) is also equal to that in (TR1). This is because (TR1) has no chance to reuse the data in Input Buffers. The layer FC1, which has the smallest number of inputs among three layers, has $512/N_i = 32$ input chunks, so that the number of the input chunks exceeds the number of entries of an Input Buffer. As a result, no input chunks in

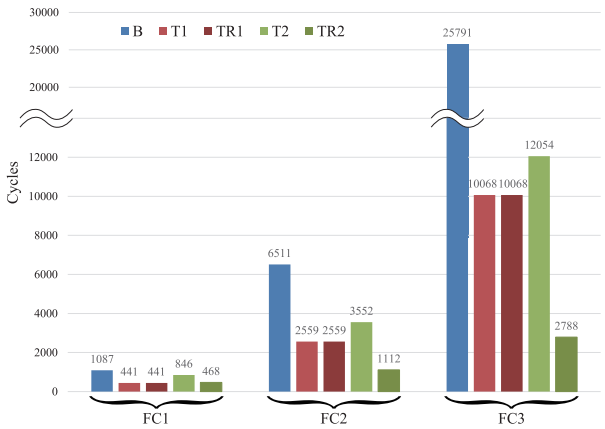


Fig. 8. Execution cycles.

Input Buffers can be reused in (TR1). All input chunks have to be read from the Shared Memory.

Next, we focus on (T2) and (TR2). Our experimental result shows that both execution cycles and numbers of per-core Shared Memory reads in (TR2) are smaller than those in (T2). This is because each Input Buffer can keep the assigned input chunks during the computation of partial results. For example, FC3, which has the largest inputs among three layers, has $2560/N_i = 160$ input chunks. These input chunks are partitioned into 16 groups under task mapping based on input signals, so that each core computes partial results with $160/N_i = 10$ input chunks. Each Input Buffer can retain the 10 input chunks because the number of the input chunks is smaller than the number of entries of the buffer. Thus, each input chunk needs to be read only once from the Shared Memory in (TR2).

Furthermore, we discuss our experimental results per layer. Figure 8 presents that the execution cycles in (TR2) is larger than that in (TR1) for FC1. This is because cores spend lots of time for the second step of task mapping based on input signals (i.e., integration of partial results). For example, the number of writes to the Shared Memory in (TR1) is only 2, while that in (TR2) is 34. This implies that the increase in writes to the Shared Memory cancels out the reduction in the execution cycles caused by the data reuse in (TR2).

In contrast, the execution cycles in (TR2) are smaller than those in (TR1) for FC2 and FC3. This is because task mapping based on input signals enables cores to reuse a number of data in Input Buffers. For example, for FC2, the numbers of writes to the Shared Memory in (TR1) and (TR2) are 5 and 85, respectively, while the numbers of reads from the Shared Memory in (TR1) and (TR2) are 405 and 90, respectively. Since the total number of Shared Memory accesses in (TR2) is much smaller than that in (TR1), (TR2) shows the higher performance than (TR1).

To clarify the relationship between numbers of Shared Memory accesses and execution cycles, we drew Fig. 9 with the results shown in Fig. 8 and TABLE II. This is the main result of this paper. We can see that the number of execution

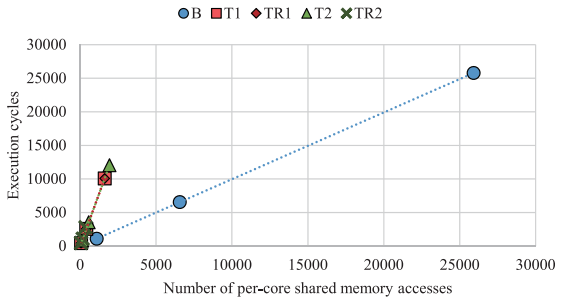


Fig. 9. Relationship between numbers of per-core shared memory accesses and execution cycles.

cycles of the multicore NNA is proportional to the number of per-core Shared Memory accesses. Note that the configuration (B) has the different gradient. This is because the throughput of Shared Memory accesses is restricted by the throughput of the single core in (B), while it is restricted by the throughput of the Shared Memory or the on-chip network in the other configurations. We can learn an important lesson from this graph: *the reduction in the number of inter-core communications is quite effective for the improved performance of multicore NNAs.*

VI. Future Work

We list the future work of our task mapping study. Although we assumed that only the data in Input Buffers were able to be reused in this evaluation, we need to examine the impact of the data reuse in Output Buffers on the performance of multicore NNAs. For example, a core has an opportunity to reuse the data in the Output Buffer when it starts to compute neurons in the next layer, because some neurons in the next layer receives some output signals computed by the core. In addition, a core can reuse the partial results of multiply-add on the Output Buffer when it starts the second step of task mapping based on input signals, because some partial results are needed to compute the final outputs on the core. The data reuse in Output Buffers helps to reduce the number of Shared Memory accesses and the execution cycles of a multicore NNA.

This paper mentions task mapping for a single layer, but task mapping for multiple layers should be also considered. Many DNNs have convolution layers in which neurons are partially connected from some neurons in the previous layer [14]. Additionally, many studies of pruning synapses to accelerate the computation of neural networks have been conducted [15], [16]. Since a core needs the limited number of output signals to compute the neurons in the next layer for these neural networks, the Output Buffer may have a chance to keep all output signals required to compute the next layer. This means that the core does not need to read input signals from the Shared Memory at the computation of the next layer. Therefore, task mapping considering the connections between layers will be more effective for partially-connected layers than fully-connected layers.

VII. Conclusion

This paper provides the first exploration of task mapping on multicore NNAs to improve the performance. Intuitively, a well-tuned task mapping has less amount of communication between cores. To confirm this assumption, we tested two types of task mappings that generate different amount of communication between cores on an NNA. Our experimental results show that the number of communication between cores strongly affects the execution cycles of the NNA and the most effective task mapping differs depending on the size of neural networks.

Our task mapping methods assume that only the data in Input Buffers are reusable, but the data in Output Buffers are also reusable on multicore NNAs. Data reuse in Output Buffers will lead to the further reduction in the number of inter-core communications, so one of our future work is to test task mapping methods considering data reuse in Output Buffers.

References

[1] C. Szegedy *et al.*, "Going Deeper with Convolutions," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[2] R. Preissl *et al.*, "Compass: A Scalable Simulator for an Architecture for Cognitive Computing," in *Proc. Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC'12)*, no. 54, 2012.

[3] Q. V. Le *et al.*, "Building High-level Features Using Large Scale Unsupervised Learning," in *Proc. 29th Int'l Conf. on Machine Learning (ICML-12)*, 2012, pp. 81–88.

[4] Y. Chen *et al.*, "DaDianNao: A Machine-Learning Supercomputer," in *Proc. 47th Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-47)*, 2014, pp. 609–622.

[5] T. Chen *et al.*, "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, 2014, pp. 269–284.

[6] J. Albericio *et al.*, "Cnvlutin: Ineffectual-Neuron-Free Deep Convolutional Neural Network Computing," in *Proc. 43rd Annual Int'l Symp. on Computer Architecture (ISCA'16)*, 2016, pp. 1–13.

[7] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *Proc. 43rd Annual Int'l Symp. on Computer Architecture (ISCA'16)*, 2016, pp. 367–379.

[8] S. Liu *et al.*, "Cambricon: An Instruction Set Architecture for Neural Networks," in *Proc. 43rd Annual Int'l Symp. on Computer Architecture (ISCA'16)*, 2016, pp. 393–405.

[9] P. A. Merolla *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, pp. 668–673, Aug. 2014.

[10] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proc. 27th Int'l Conf. on Machine Learning (ICML-10)*, 2010, pp. 807–814.

[11] P. Merolla *et al.*, "A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm," in *Proc. Custom Integrated Circuits Conf. (CICC)*, 2011.

[12] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *Proc. 39th Annual Int'l Symp. on Computer Architecture (ISCA'12)*, 2012, pp. 356–367.

[13] M. Ohba, S. Miwa, S. Shindo, T. Tsumura, H. Yamaki, and H. Honda, "Initial study of reconfigurable neural network accelerators," in *Proc. 7th Int'l Workshop on Advances in Networking and Computing (WANC'16)*, Nov. 2016.

[14] Y. LeCun *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov. 1998.

[15] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both Weights and Connections for Efficient Neural Network," in *Advances in Neural Information Processing Systems 28 (NIPS 2015)*, 2015, pp. 1135–1143.

[16] A. Polyak and L. Wolf, "Channel-Level Acceleration of Deep Face Representations," *IEEE Access*, vol. 3, pp. 2163–2175, 2015.