IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Task Parallelism-Aware Deep Neural Network Scheduling on Multiple Hybrid Memory Cube-based Processing-in-Memory

YOUNG SIK LEE[1], (Student Member, IEEE), TAE HEE HAN, (Member, IEEE)
[1]Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, South Korea
[2]Department of Semiconductor Systems Engineering, Sungkyunkwan University, Suwon 16419, South Korea

Corresponding author: Tae Hee Han (e-mail: than@ skku.edu).

This work was supported in part by Ministry of Trade, Industry & Energy (MOTIE, Korea) (No. 20011074) and in part by the MOTIE and KEIT(20010560, Development of system level design and verification for in storage processing architecture based on phase change memory).

ABSTRACT Processing-in-memory (PIM) comprises computational logic in the memory domain. It is the most promising solution to alleviate the memory bandwidth problem in deep neural network (DNN) processing. The hybrid memory cube (HMC), a 3D stacked memory structure, can efficiently implement the PIM architecture by maximizing the existing legacy hardware. To accelerate DNN inference, multiple HMCs can be connected, and data-independent tasks can be assigned to processing elements (PEs) within each HMC. However, owing to the packet-switched network structure, inter-HMC interconnects exhibit variable and unpredictable latencies depending on the data transmission path and link contention. A well-designed task schedule using context switching can effectively hide communication latency and improve PE utilization. Nevertheless, as the number of HMC increases, the variability of a wide range of inter-HMC communication latencies causes frequent context switching, degrading overall performance. This paper proposes a DNN task scheduling that can effectively utilize task parallelism by reducing the communication latency variance owing to HMC interconnect characteristics. Task partitions are generated to exploit parallelism while providing inter-HMC traffic within the sustainable link bandwidth. Task-to-HMC mapping is performed to hide the average communication latency of intermediate DNN processing results. A task schedule is generated using retiming to accelerate DNN inference while maximizing resource utilization. The effectiveness of the proposed method was verified through simulations using various realistic DNN applications performed on a ZSim x86-64 simulator. The simulations revealed that DNN processing with the proposed scheduling improved the DNN processing speed by reducing the processing time by 18.19% over conventional methods where each HMC operated independently.

INDEX TERMS Processing-in-memory, hybrid memory cube, deep neural network, task scheduling, parallel computing

I. INTRODUCTION

Deep neural networks (DNNs) have achieved breakthroughs in solving a wide range of challenging computation problems, from image recognition to speech translation [1]. State-of-the-art DNNs rely on massive parameter count ($\geq 10^8$) and manually designed architectures (with 10^1–10^3 layers) to surpass human performance. Inferring these models causes the von Neumann bottleneck between the memory and the processor in existing architectures. Off-chip DRAM access consumes hundreds of times the power consumed by an ALU operation [2], [3]. The processing-in-memory (PIM) paradigm performs computations using computational logic added to memory and is the most promising approach for alleviating these shortcomings [4]. Past attempts to integrate high-density memory and computational logic on a single chip or die have been extremely complex and expensive.

3D-stacked memory architectures, including hybrid memory cubes (HMCs), offer the possibility of implementing

PIM platforms efficiently [5]–[7]. Their through-silicon via (TSV)-based connections dramatically reduce process complexity and yield degradation problems by enabling cost-effective integration of independently fabricated logic and memory. In an HMC, each grouping of memory partitions is combined with a corresponding section of the logic die, forming a parallelizable computing unit referred to as a vault. Although processing elements (PEs) in the HMC provide the benefit of computational efficiency, the system performance does not scale proportionally with the number of PEs [8]. This occurs because adding a PE also increases the amount of memory bandwidth required to support it. However, the memory bandwidth remains almost constant irrespective of the memory capacity owing to the limitation on the number of TSVs per HMC. As a result, to support a higher computing performance, a new HMC must be connected to increase the memory bandwidth and the number of PEs. Existing studies, including Tesseract [5] and GraphH [6], have focused on this problem and proposed an efficient connection structure of HMCs for PIM.

The most important objective is to maximize the overall performance by balancing the total computing throughput of PEs and the sustainable BW that inter-HMC links can provide. Data-level independent tasks can be allocated to different PEs to improve parallel processing. Throughput can be improved compared to a single HMC by utilizing PEs in multiple HMCs. However, the increased latency of communication traffic between HMCs may cause a bottleneck in throughput improvement.

In inter-HMC networks, a packet-switched protocol is used instead of the DDRx protocol [9], [10]. This allows parallel transmission between HMCs, but the communication latency between HMCs increases and becomes variable owing to the following reasons. First, the number of hops that the packets pass through varies depending on the source and destination of data transmission owing to the router-based connection. In intra-HMC communications, data can be transferred within a guaranteed time through a fully connected interconnect with sufficient bandwidth, but this is not possible in inter-HMC communications. Second, when the number of packets exceeds the sustainable link bandwidth, contention occurs, further increasing the latency. A well-designed task schedule that uses context switching is effective for hiding communication latency and improving the PE utilization. Nevertheless, latency hiding that mitigates the high latency of the interconnect between HMCs requires frequent context switching, thus generating additional delays. Therefore, a task scheduling technique is needed that considers the effect of the number of context switching on the worst-case latency.

This paper proposes a DNN task scheduling that fully utilizes data-level parallelism by reducing the communication latency variance owing to HMC interconnect characteristics. Tasks are allocated to the HMC to reduce link contention and long hop count, which are the factors that increase the worst-case latency. First, in task partitioning to utilize data-level parallelism, contention is prevented by maintaining the communication between HMCs as a sustainable link bandwidth. Next, task-to-HMC mapping is performed to hide the average communication latency of the intermediate DNN processing results. Finally, in task scheduling, the communication latency of each task becomes hidden, and PE utilization is improved. The primary contributions of this study are as follows. First, the DNN computation performance is improved by mitigating the worst-case communication latency between HMCs in task scheduling. This is solved by preventing traffic that exceeds the sustainable bandwidth in task partitioning and minimizing the hop count of communication in task-to-HMC mapping. Second, because the proposed task partitioning and scheduling are recursive, the schedule can be derived faster through dynamic programming.

The remainder of this paper is organized as follows. Section II introduces HMC-based PIM studies related to the proposed method. Then, the advantages of the new method over the existing methods are presented. Section III presents the proposed technique in terms of task partitioning, task-to-HMC mapping, and schedule generation. In Section IV, the effectiveness of the proposed method is confirmed through a ZSim-HMC-based system-level simulation using various realistic DNN applications. Section V provides the conclusion.

II. RELATED WORK

A. PROCESSING-IN-MEMORY

PIM architecture can place computational resources inside or near the memory to enhance the processing speed [11]-[14]. Several solutions have been proposed to improve the performance of PIM architectures. To date, PIM solutions typically leverage the high internal bandwidth of DRAM dual in-line memory modules (DIMM) to accelerate computation by modifying the architecture or operation of DRAM chips to implement computations within the memory [15], [16]. Beyond DRAM, researchers have also demonstrated similar PIM capabilities by leveraging the unique properties of emerging non-volatile memory (NVM) technologies such as resistive RAM (ReRAM) and phase-change memory (PCM). The specific implementation approaches vary widely depending on the type of memory technology used; however, the modifications are generally minimal to preserve the original function of the memory unit and meet area constraints. For instance, certain DRAM-based solutions minimally change the DRAM cell architecture while relying heavily on altering memory commands from the memory controller to enable functions such as copying a row of cells to another, logical operations such as AND, OR, and NOT, and arithmetic operations such as addition and multiplication. DRAM and NVM-have demonstrated promising improvements for graph and database workloads by accelerating search and update operations directly where the data are stored.

The HMC is also a promising solution for implementing PIM by maximizing the legacy hardware without new pro-

cess technology [10]. The HMC memory stack contains 16 or 32 vaults. The memory and logic layers in each vault are connected to the TSVs. The HMC architecture places the PE close to the data in memory. Similar architectures have been applied to TETRIS [17] and Tesseract [5], which integrate a neural network engine on a logic die and utilize multiple DRAM dies on the HMC.

Azarkhish et al. proposed a smart memory cube that added high bandwidth, low latency, and eXtensible Interface-4.0 [18] compatible logic base (LoB) interconnect to meet the enormous bandwidth demand of HMC serial links [19]. The modified intra-HMC interconnect provides additional bandwidth to the PIM device embedded in the LoB. Dai et al. proposed GraphH which links 16 HMCs for large graph processing [6]. They dealt with random access and poor locality issues through an on-chip vertex buffer and reconfigurable double mesh topology. In addition, they solved the problem of unbalanced workloads and heavy conflicts through index mapping interval-block and round interval pairs.

Ahn et al. proposed a Tesseract architecture to maximize the available memory bandwidth in multiple HMC-based PIM structures for graph processing [5]. Tesseract is still widely used as the base architecture for HMC-based PIM, and the system architecture is presented in Fig. 1. In Tesseract, the network between the PEs of the HMC and the I/O links is configured in a fully connected topology. Each PE has an integrated in-order core for execution and a PF buffer (PFB) to prefetch communication traffic. Links between HMCs are connected in a dragonfly topology.

The proposed method focuses on the HMC architecture, such as Tesseract. It provides task scheduling that aims to exploit parallelism in a system connecting HMCs to improve PIM throughput and reduce the communication delay time between HMCs.

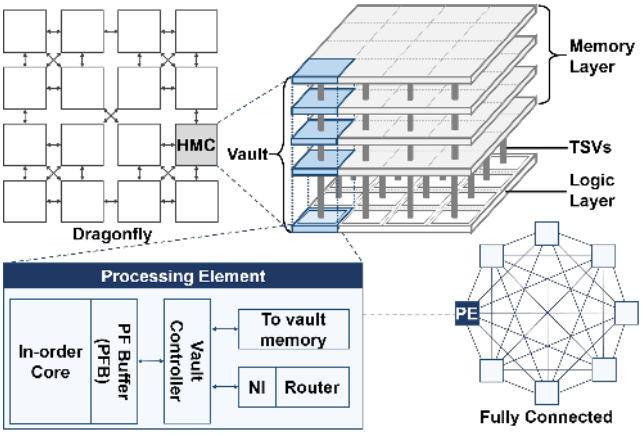


FIGURE 1. HMC system architecture for PIM processing

B. DEEP NEURAL NETWORK SCHEDULING

Numerous DNN acceleration techniques have been conducted to alleviate the complexity of DNN tasks that require tremendous computation and data access, considering the attributes of algorithms and computational structure. Related studies have mainly focused on minimizing data movements while incorporating compression and pruning to eliminate redundances in computation processes and data formats [18], [19]. By pruning weights and zero activations, the computational efficiency can be improved, and memory footprints can be effectively reduced. Chen et al. proposed the neural network accelerator Eyeriss, which uses zero-valued neurons to reduce power consumption [20].

A schedule that improves data-level parallelism under a given task size and memory access volume can also be combined with the aforementioned redundancy removal technology to accelerate DNN processing. Previously, loop blocking or auto-tuning has been used to improve the operation speed of DNNs [21], [22]. Donyanavard et al. proposed SPARTA, a throughput-aware task scheduling for many-core platforms [23]. SPARTA collects sensor data to characterize tasks and uses this information to prioritize these tasks while performing assignments. Gao et al. proposed Tetris, an HMC-based neural network accelerator, and a scheduling algorithm for it [17]. Tetris developed a hybrid partitioning scheme that parallelizes neural network calculations through multiple accelerators. Wang et al. jointly optimized the allocation of computation and memory resources on the 3-D-stacked PIM architecture to minimize schedule length by removing synchronization overhead [24]. In the depth of previous work, Wang et al. proposed a scheduling scheme, Para-Net, to completely utilize all the resources of a single HMC in a neural network application [7]. They maximized resource utilization by setting the standard for storing intermediate results in a cache with fast access or in a large-capacity DRAM and determining the phase through retiming-based task scheduling.

The proposed task scheduling differs from the aforementioned technology. We aim to accelerate DNN processing through task scheduling by reducing the variance of inter-HMC communication latency. None of the above tasks consider the data communication latency that changes according to task allocation in scheduling.

III. TASK PARALLELISM-AWARE SCHEDULING

By scaling up the number of HMCs, the computing capacity can be expanded; however, the variation in communication latencies in the inter-HMC communications also increases, which exacerbates the worst-case performance. The proposed task scheduling exploits parallelism while preventing longer communication latencies between HMCs. First, task partitions are generated to exploit parallelism while providing inter-HMC traffic within the sustainable link bandwidth. Second, a task-to-HMC mapping is performed to hide the average communication latency of intermediate DNN processing results. Third, a task schedule is generated using retiming to accelerate the DNN processing while maximizing resource utilization. For the formal expression of each step, **Definition**

1 and the notation list of Table 1 are presented.

Definition 1. A DNN model can be expressed as a directed graph, $G(\mathcal{V}, \mathcal{E})$. $v_{i,j} \in V_i$ corresponds to the j-th node of the i-th layer, and V_i is a subset of $\mathcal{V}$. $e^i_{j,k} \in E$ is the directed edge from $v_{i,j}$ on the upstream layer V_i to $v_{i+1,k}$ on the downstream layer V_{i+1}. Each edge represents a delivery of processing results from the upstream node to the downstream node.

TABLE 1. Notations used in the proposed scheduling

Notation	Definition
$\mathcal{V}$,$\mathcal{E}$,$\mathcal{C}$	The set of task nodes, edges, and processors, respectively
$L^i_{j,k}$	The transmission latency of intermediate result on edge $e^i_{j,k}$
$\mathbb{P}$	The set of task partitions
$\mathcal{P}^n$	The set of task nodes in the nth task partition
P^n_i	The set of task nodes in the nth layer on the ith task partition
$es(v_{i,j})/ef(v_{i,j})$	Execution start/finish time of task $v_{i,j}$
$cs(e^i_{j,k})/cf(e^i_{j,k})$	Communication start/finish time of $e^i_{j,k}$
$E_{in}(\mathcal{P}^n)$	The number of internal edges of task partition $\mathcal{P}^n$
$E_{ext}(\mathcal{P}^n)$	The number of external edges of task partition $\mathcal{P}^n$
$E(\mathcal{P}^n, \mathcal{P}^m)$	The number of edges from task partition $\mathcal{P}^n$ to $\mathcal{P}^m$
D_{in}	The data transmission latency in intra-HMC communication
D_{out}	The data transmission latency in inter-HMC links
T	The iteration period
N_{PE}	The number of PEs in an HMC
N_C	The number of HMCs in PIM architecture
N_P	The number of task partitions in the DNN graph
BW_{MAX}	The bandwidth of inter-HMC links
r_{MAX}	The maximum retiming value of the schedule
$\mathbb{R}[S][m]$	The maximum retiming value for the subset of m intermediate results according to the available PFB capacity S

A. DNN TASK PARTITIONING

Prior to task scheduling, set of vertices $\mathcal{V}$ on the DNN graph is partitioned into subgraphs $\mathcal{P}^n$ to be allocated to each HMC. The finer the granularity of task partitions, the more the PEs that can operate concurrently. However, this concept causes frequent data transmission on inter-HMC communication and consequently leads to diminishing return. Therefore, the size of the task partition should be determined by considering the link bandwidth and delay time between HMCs.

If data-dependent tasks are assigned to different PEs, the transmission of intermediate results between the PEs is inevitable. Since the inter-HMC communication latency is longer than that of intra-HMC communication, data-dependent tasks can be allocated within the same HMC to reduce the latency. By contrast, even though data-independent tasks are allocated to different PEs, there is no communication traffic. By assigning them to different HMCs, parallel processing can be improved without communication latency.

Fig. 2(a) presents an example of graph processing through a single HMC in a DNN graph. We assume a DNN graph with four layers and 12 vertices and an HMC structure

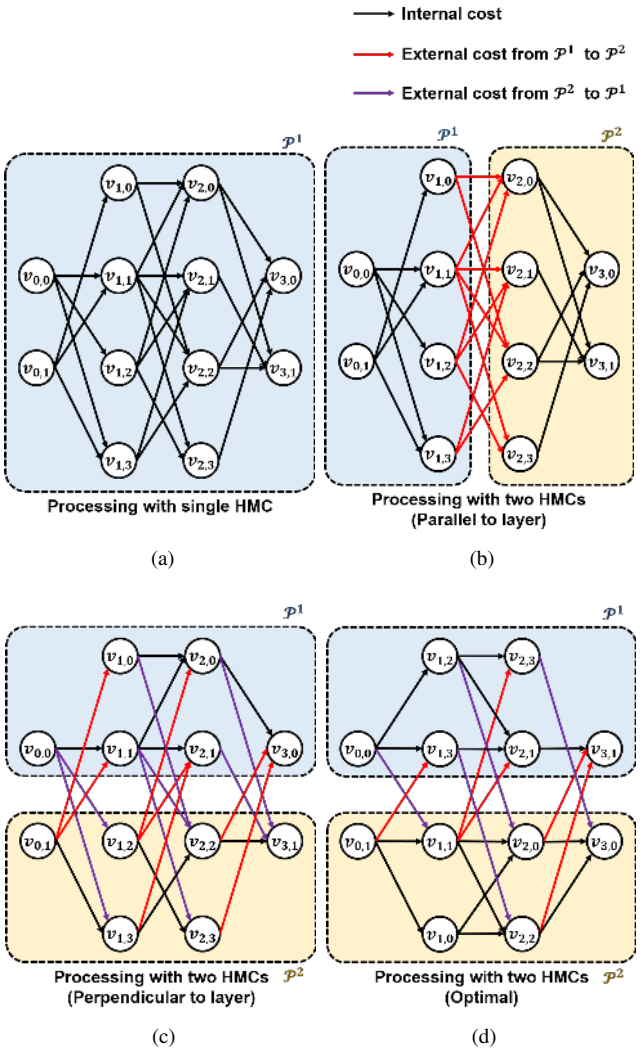


FIGURE 2. Example of partitioning a DNN task graph (a) Operation on a single HMC, (b) Operation on two HMCs with partition parallel to the layer, (c) Operation on two HMCs with partition perpendicular to the layer, and (d) Operation on two HMCs with partition perpendicular to the layer minimizing data movement between partitions

with two vaults for simplicity. If all tasks in the DNN are performed in a single HMC, the results will be stored in the cache or DRAM of the vault where the next task will be processed. In DNN processing using two HMCs, tasks can be partitioned parallel to the DNN layers as depicted in Fig. 2(b) or perpendicular to the DNN layers as depicted in Fig. 2(c). If task partitions are generated parallel to the DNN layers, there is a dependency between all the tasks between partitions. Consequently, computational speedup cannot be achieved. In contrast, if task partitions are generated perpendicular to the DNN layers, parallel computation is possible between vertices in each layer in different partitions.

Because the number of edges between task partitions represents the amount of communication traffic between HMCs, task partitions should be generated to reduce the number of edges between the partitions. In Fig. 2(c), $E_{in}(\mathcal{P}^1)$ is

five, $E_{in}(\mathcal{P}^2)$ is four, $E(\mathcal{P}^1, \mathcal{P}^2)$ is seven, and $E(\mathcal{P}^2, \mathcal{P}^1)$ is seven. Because these task partitions have more external edges than internal edges, if these partitions are assigned to different HMCs, they have more inter-HMC communications than intra-HMC communications. These allocations do not properly utilize the low-latency, high-bandwidth interconnects inside the HMC.

Fig. 2(d) shows the partitioning result that minimizes the number of edges between subgraphs. Here, $E_{in}(\mathcal{P}^1)$ is six, $E_{in}(\mathcal{P}^2)$ is eight, $E(\mathcal{P}^1, \mathcal{P}^2)$ is four, and $E(\mathcal{P}^2, \mathcal{P}^1)$ is five. The result has more internal edges and fewer external edges than that shown in Fig. 2(c); thus faster intra-HMC links can be utilized as much as possible compared to slower inter-HMC links. To reduce the worst-case communication latency caused by setting communication traffic to less than the sustainable bandwidth, partitions with reduced inter-HMC communication are needed, as shown in Fig. 2(d).

Algorithm 1 presents the task partitioning to reduce the number of edges between partitions First, it is necessary to check whether the performance gain can be observed when the target DNN graph is separated. There are situations in which it is difficult to expect performance improvement when partitions are divided. These cases occur when task parallelism cannot be improved even by partitioning or the inter-HMC communication delay time is long, offsetting the gain of parallel processing. Therefore, a partition is generated only when the following three conditions are satisfied.

Condition 1) In the layer with the most vertices, the number of vertices per partition is greater than the number of PEs in a single HMC.

$$\forall V_i \subset \mathcal{V}, Max(n(V_i)) > N_{PE} \tag{1}$$

Condition 2) The number of internal edges of a partition is greater than D_{out}/D_{in} times the number of external edges.

$$\forall i, E_{in}(\mathcal{P}^i) > D_{out}/D_{in} \times max(\sum_{\forall j} E(\mathcal{P}^i, \mathcal{P}^j), \sum_{\forall j} E(\mathcal{P}^j, \mathcal{P}^i)) \tag{2}$$

Condition 3) The number of external edges between partitions is less than the bandwidth requirement.

$$\forall i, \forall j, E(\mathcal{P}^i, \mathcal{P}^j) < BW_{MAX} \tag{3}$$

When **Conditions 1** to **3** are satisfied, the DNN graph is partitioned based on the Kernighan-Lin algorithm. The Kernighan-Lin algorithm is a heuristic method for partitioning arbitrary graphs which is both effective in determining optimal partitions, and fast enough to be practical in solving large problems [25]. Because this algorithm solves the partitioning problem of undirected graphs, it must be modified to apply to DNN graphs whose edges have orientation. Because HMC links operate in full-duplex mode, the edges must be reduced by dividing the edges according to the direction of the edges. Therefore, when dividing the DNN graph G into two subgraphs $\mathcal{P}^a$ and $\mathcal{P}^b$, the target to reduce should be as follows.

$$max(E(\mathcal{P}^a, \mathcal{P}^b), E(\mathcal{P}^b, \mathcal{P}^a)) \tag{4}$$

The Kernighan-Lin algorithm is modified considering the oriented edge of this DNN graph and the full-duplex communication link between HMCs.

First, initial partitions $\mathcal{P}^a$ and $\mathcal{P}^b$ are generated by randomly dividing the vertices on each layer in half. For the vertices belonging to each partition, $E_{in}(v_{i,j})$ for $v_{i,j} \in P^a$ is the internal cost of $v_{i,j}$ for $\mathcal{P}^a$, that is, the sum of the number of edges between $v_{i,j}$ and other vertices in $\mathcal{P}^a$. $E(v_{i,j}, \mathcal{P}^b)$ is the external cost of $v_{i,j}$ for $\mathcal{P}^b$, that is, the sum of the number of edges with vertices in $\mathcal{P}^b$. Let $D(v_{i,j})$ be the difference between the external and internal costs of $v_{i,j}$ in $\mathcal{P}^a$:

$$D(v_{i,j}) = E(v_{i,j}, \mathcal{P}^b) - E_{in}(v_{i,j}) \tag{5}$$

When $v_{i,j}$ in $\mathcal{P}^a$ and $v_{i,k}$ in $\mathcal{P}^b$ are interchanged, the reduction in cost g is

$$g = max(E(v_{i,j}, \mathcal{P}^b), E(v_{i,k}, \mathcal{P}^a)) - E_{in}(v_{i,j}) - E_{in}(v_{i,k}) \tag{6}$$

The goal of the proposed partitioning is to determine the optimal series of interchanges between the components of $\mathcal{P}^a$ and $\mathcal{P}^b$ that maximize g. The algorithm randomly classifies the pairs of vertices in each layer of the DNN graph and calculates the cost reduction according to their exchange. For each layer, an interchange that can maximize this cost reduction is explored.

The Kernighan-Lin algorithm divides a graph into two subgraphs with half the number of vertices. To generate additional partitions, the algorithm should be recursively run with $\mathcal{P}^a$ and $\mathcal{P}^b$ as inputs. The algorithm is repeated until a specific partition does not satisfy the aforementioned conditions or until up to seven task partitions are generated. The partitions generated in this manner are combined with the dragonfly topology between HMCs to generate a task schedule that hides latency. The details are described in Section 3.B.

B. TASK-TO-HMC MAPPING

After task partitioning, they are mapped to the HMC to perform DNN operations. Task mapping aims to generate an advantageous structure to hide latency while reducing communication between task partitions. The size of task

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3077294, IEEE Access

Algorithm 1 DNN graph partitioning, $TPart(G(\mathcal{V},\mathcal{E}),N_P)$

Input 1) DNN graph $G(\mathcal{V},\mathcal{E})$,
2) the number of task partitions N_P

Output 1) Graph partitions $\{\mathcal{P}^0, \mathcal{P}^1, ..., \mathcal{P}^{N_P-1}\}$

if G violates **Condition 1** do
  return $\{G\}$
end if
determine a balanced initial partition of the nodes into $\mathcal{P}^a$ and $\mathcal{P}^b$
do
  compute $D(v_{i,j})$ values for all $v_{i,j} \in \mathcal{V}$
  generate gv, av, and bv as empty lists
  for i:=0 **to** $N_L - 1$ **do**
    for n:=1 **to** $n(V_i)/2$ **do**
      find $v_{i,j}$ from $\mathcal{P}^a$ and $v_{i,k}$ from $\mathcal{P}^b$ such that $g = max(E(v_{i,j},\mathcal{P}^b), E(v_{i,k},\mathcal{P}^a))$ $E_{in}(v_{i,j}) - E_{in}(v_{i,k})$ is maximal
      remove $v_{i,j}$ and $v_{i,k}$ from further consideration in this pass
      add g to gv, a to av, and b to bv
      update D values for the elements of $\mathcal{P}^a$ and $\mathcal{P}^b$
    end for
  end for
  find k which maximizes g_{max}, the sum from $gv[0]$ to $gv[k]$
  if $g_{max} > 0$ **then**
    exchange $av[0], av[1], \ldots, av[k]$ with $bv[0], bv[1], \ldots, bv[k]$
  end if
until $g_{max} \leq 0$
if $\mathcal{P}^a$ and $\mathcal{P}^b$ violates **Condition 2** and **Condition 3 then**
  return $\{G\}$
end if
N_P++
if $N_P < 5$ **then**
  $\mathcal{P}^a = Partition(\mathcal{P}^a, N_P)$
  $\mathcal{P}^b = Partition(\mathcal{P}^b, N_P)$
else if $N_P = 6$ **then**
  $\mathcal{P}^a = Partition(\mathcal{P}^a, N_P + 1)$
end if
return $\{\mathcal{P}^a, \mathcal{P}^b\}$

partitions obtained using **Algorithm 1** may be uniform or non-uniform. For example, if $G(\mathcal{V},\mathcal{E})$ is divided into five task partitions, partitions $\mathcal{P}^0$, $\mathcal{P}^1$, and $\mathcal{P}^2$ with $n(\mathcal{V})/4$ number of vertices and $\mathcal{P}^3$ and $\mathcal{P}^4$ with $n(\mathcal{V})/8$ number of vertices are composed. When performing operations on these in each HMC, each layer operation in $\mathcal{P}^3$ and $\mathcal{P}^4$ consumes half the computation time compared to $\mathcal{P}^0$, $\mathcal{P}^1$, and $\mathcal{P}^2$. HMCs assigned with $\mathcal{P}^3$ and $\mathcal{P}^4$ have half the number of computation tasks as the HMCs assigned $\mathcal{P}^0$, $\mathcal{P}^1$, and $\mathcal{P}^2$; therefore, they have an idle time corresponding to the difference in processing time. In task scheduling, the frequency of context switching for latency hiding is reduced by utilizing the idle time caused by the uneven workload between these partitions.

The size of the task partition is classified into one or two types. Depending on the value of N_P, the number of greater partitions N_{PB} and the number of smaller partitions N_{PL} are determined as shown in Table 2. When F is defined as:

$$F = 2^{floor(\log_2 N_P)}. \tag{7}$$

The value of N_{PB} and N_{PL} can be expressed as:

$$N_{PB} = \begin{cases} N_P, & (F = N_P), \\ 2F - N_P, & (F \neq N_P), \end{cases} \tag{8}$$

$$N_{PL} = 2N_P - 2F \tag{9}$$

TABLE 2. Value of N_{PB} and N_{PL} according to the number of partitions N_P

N_P	N_{PB}	N_{PL}
1	1	0
2	2	0
3	1	2
4	4	0
5	3	2
6	2	4
7	1	6

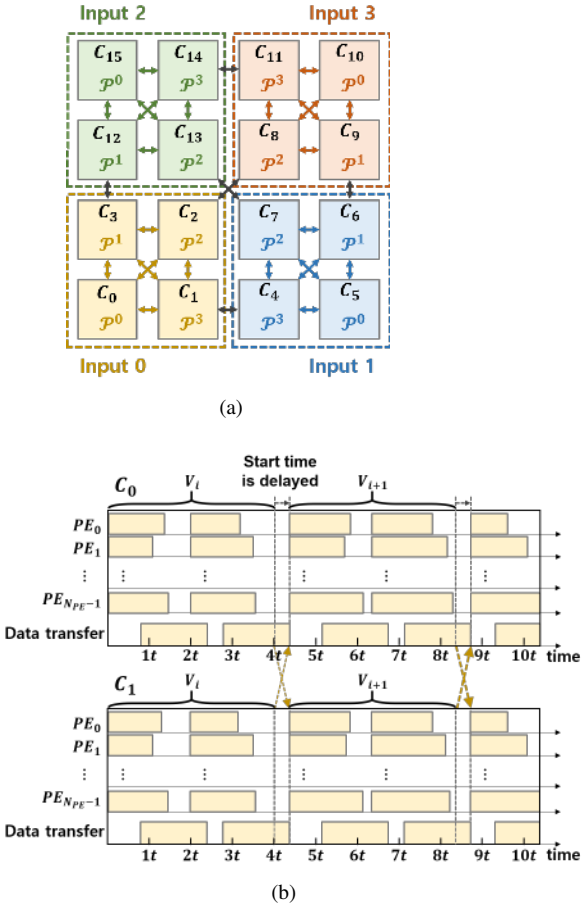
The dragonfly topology between HMCs enables one-hop communication for every four HMCs. If the maximum value of N_P is limited to seven, then N_{PB} has a value of four or less. By assigning N_{PB} large partitions to these HMCs, they can minimize latency through one-hop communication.

Fig. 3(a) presents an example of a mapping for a DNN graph divided into four task partitions. Uniform sizes $\mathcal{P}^0$, $\mathcal{P}^1$, $\mathcal{P}^2$, and $\mathcal{P}^3$ are allocated to C_0, C_1, C_2, and C_3, respectively. Fig. 3(b) shows an example of the corresponding DNN processing schedule. When the operation on V_i is completed in each HMC, intermediate results are moved to prepare for the operation on V_{i+1}. The latency that occurs during this process slows the operation start time of V_{i+1} and increases the DNN processing time.

Fig. 4(a) shows an example of mapping for a DNN graph divided into five task partitions. For operation on input 0 of DNN processing, task partitions $\mathcal{P}^0$, $\mathcal{P}^1$, and $\mathcal{P}^2$ are assigned to C_0, C_1, and C_2, respectively, to enable one-hop communication. $\mathcal{P}^3$ and $\mathcal{P}^4$ can be assigned to C_3 and C_{12} to minimize the hop count of communication with other assigned task partitions. To reduce communication latency variation, a partition with a larger number of external edges among $\mathcal{P}^3$ and $\mathcal{P}^4$ is allocated to C_1. Both C_1 and C_4 are assigned tasks of size $n(\mathcal{V})/8$ and have idle time to perform tasks of size $n(\mathcal{V})/8$ for other inputs. Accordingly, $\mathcal{P}^3$ and $\mathcal{P}^4$ for another DNN processing input 1 may be allocated to C_4 and C_1, respectively.

Fig. 4(b) illustrates an example of the corresponding DNN processing schedule. The operation of the V_{i+1} layer of $\mathcal{P}^0$ for DNN input 0 mapped to C_0 takes $4t$.However, the operation of the V_{i+1} layer of $\mathcal{P}^3$ mapped to C_1 takes $2t$. If the V_i layer operation of $\mathcal{P}^3$ starts at $5t$, this operation is completed at $7t$. Because the V_i layer of $\mathcal{P}^0$ is completed at the time of $4t$, there is a spare time of $1t$ for transferring

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3077294, IEEE Access



(a)

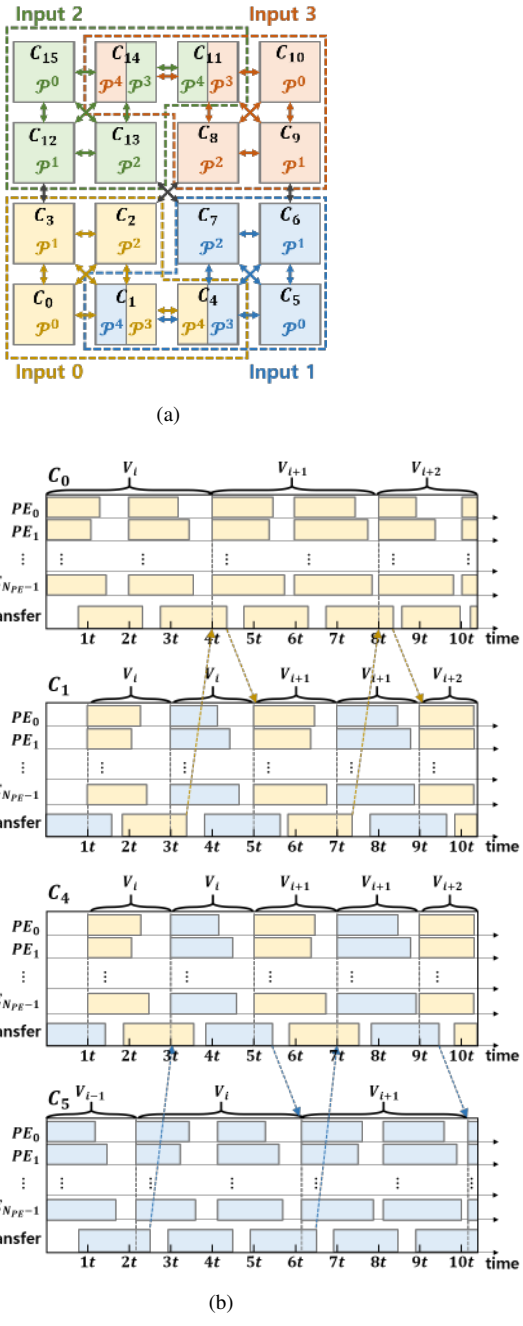
(b)

FIGURE 3. (a) Mapping example for a DNN graph divided into four task partitions and (b) Corresponding DNN processing schedule in C_0 and C_1

the intermediate result to C_1. Similarly, because the V_{i+2} layer operation of $\mathcal{P}^0$ is completed at the time of $8t$, there is a spare time of 1t to transfer the intermediate result from C_1. Thus, $\mathcal{P}^0$ and $\mathcal{P}^3$ have an extra interval of 1t between operations in each layer, and this interval can be used to hide the transmission latency of the intermediate results.

This method reduces the worst-case communication latency between each HMC and maintains the traffic limited to less than sustainable bandwidth using **Condition 3** in the task partitioning step. In Fig. 3(a), the amount of communication of the link between C_1 and C_2 is the same as that in Fig. 4(a). This is because this link also includes communication between C_4 and C_2.

Algorithm 2 describes the task-to-HMC mapping that reduces the communication hop count between HMCs. First, N_{PB} and N_{PL} values are obtained using (8) and (9). Next, the N_{PB} partitions are mapped in ascending order starting from C_0 with the smallest number of linked links. Next, the remaining N_{PL} partitions are mapped from $C_{N_{PB}-1}$ and $C_{4(N_{PB}-1)}$. Consequently, the mapping result shortens the communication path of the intermediate resulting in detailed scheduling, thereby reducing latency variation.



(a)

(b)

FIGURE 4. (a) A mapping example for DNN graph divided into five task partitions and (b) Corresponding DNN processing schedule in C_0, C_1, C_4, and C_5

C. TASK SCHEDULING

After the task-to-HMC mapping, DNN task scheduling is performed, which maximizes resource utilization between HMCs and hides communication latency between HMCs. The retiming concept of Para-Net [7] can improve the utilization of PEs while effectively hiding latency for intermediate results. However, these were limited to a single HMC research; hence, a schedule must be generated that considers the communication delay time between HMCs.

Algorithm 2 Task-to HMC mapping, $TMap(\mathbb{P}, \mathcal{C})$

Input 1) Task partitions $\mathcal{P}_0, \mathcal{P}_1, ..., \mathcal{P}_{N_P-1} \subset \mathbb{P}$,
2) HMC set $C_0, C_1, ..., C_{N_C-1} \subset \mathcal{C}$

Output 1) Task partition mapping on HMCs $\mathcal{M}_{\mathcal{P}^i \to \mathcal{C}_j}$, $\mathcal{P}_i \subset \mathbb{P}, C_j \subset \mathcal{C}$

find F from (7)
if $F == N_p$ **then**
  $N_{PB} = N_p$
else
  $N_{PB} = 2F - N_p$
end if
$N_{PL} = 2N_P - 2F$
for $i := 0$ **to** $N_{PB} - 1$ **do**
$Map(\mathcal{P}^i \to \mathcal{C}_i)$
end for
for$j := 0$ **to** $N_{PB} + (N_{PL}/2)$ **do**
  if $E_{ext}(\mathcal{P}^{2j}) > E_{ext}(\mathcal{P}^{2j+1})$ **then**
    $Map(\mathcal{P}^{2j} \to \mathcal{C}_j)$
    $Map(\mathcal{P}^{2j+1} \to \mathcal{C}_4 j)$
  else then
    $Map(\mathcal{P}^{2j+1} \to \mathcal{C}_j)$
    $Map(\mathcal{P}^{2j} \to \mathcal{C}_4 j)$
  end if
end for

The intermediate result transfer schedule between data-dependent tasks is affected by the end time of the upstream task and the start time of the downstream task. Intermediate results can be stored in PFB or large-capacity DRAM presented in Fig. 1, varying access latency accordingly. This latency variance is intensified by the packet-based communication structure in the multi-HMC based PIM system. The concept of retiming is applied to solve the problem of context switching and latency hiding accordingly. First, the terms iteration, prologue, and retiming to introduce the proposed task scheduling is described.

Definition 2 (Iteration). Given a DNN partition $G(\mathcal{V}, \mathcal{E})$, iteration is a set of processes that repeat a set of vertices $v_{i,j} \in \mathcal{V}$ that are concurrently processed by a number of PEs.

Definition 3 (Prologue). Given a DNN partition $G(\mathcal{V}, \mathcal{E})$, a prologue is an opening to a schedule that contains vertices $v_{i,j} \in \mathcal{V}$. After prologue, the iteration is repeated until the DNN processing is completed.

Definition 4 (Retiming). Given a DNN partition $G(\mathcal{V}, \mathcal{E})$, retiming R of G is a function that $v_{i,j} \in \mathcal{V}$ to an integer $R(i, j)$. Initially $R(i, j) = 0$. By retiming $v_{i,j}$ once, if it is legal, $R(i, j) = R(i, j)+1$, and one iteration of a convolution operation $v_{i,j}$ is re-allocated into the prologue.

The retiming value $R(i, j, k)$ for intermediate result $L^i_{j,k}$ transferred from each $v_{i,j}$ to $v_{i+1,k}$ represents the number of iterations of $e^i_{j,k}$ re-allocated in the prologue. When two data-dependent tasks are retimed as prologues, the following relation exists between the two tasks.

Theorem 1. For a pair of convolution operations $e^i_{j,k} \in E$ in iteration l, after $v_{i,j}$ is retimed $R(i, j)$ times and $v_{i+1,k}$ is retimed $R(i+1, k)$-times, the associated intermediate result, $L^i_{j,k}$, with a transmission latency less than mT can always be scheduled if $v_{i,j}$ that has been retimed $(l - R(i, j))$ times is re-allocated at least $(m + 1)$ more iterations ahead of the $v_{i+1,k}$ retimed $(l - R(i + 1, k))$-times.

Proof. By retiming $v_{i,j}$ two more iterations ahead of $v_{i+1,k}$ in the iteration $(l-R(j))$, $v_{i,j}$ will be re-allocated in iteration $(l - R(j) - 2)$. In each iteration, $es(v_{i,j}) < T$. Let $L^i_{j,k}$ in iteration $(l-R(i, j)-1)$ be the associated intermediate result of $v_{i,j}$ retimed $(l - R(i, j) - 2)$-times and $v_{i+1,k}$ retimed $(l-R(i+1, k))$-times. As $es(v_{i,j})+ef(v_{i,j})$ in $(l-R(j)-2)$-th iteration is less than $es(e^i_{j,k})$ in $(l - R(j) - 1)$-th iteration and $es(v_{i,j})+ef(v_{i,j})$ in $(l-R(j)-1)$-th iteration is less than $es(e^i_{j,k})$ in $(l-R(j))$-th iteration, the associated intermediate result $L^i_{j,k}$ is always schedulable during that time span.

The application of **Theorem 1** is illustrated in Fig. 5. **Theorem 1** represents the upper limit of the maximum relative retiming value for each pair of operations. Based on the definition of this retiming value, the dependency for each task pair places at most two $v_{i,j}$ iterations in the prologue.

$$r_{max} = \{max(R(i, j), v_{i,j} \in \mathcal{V})\} \tag{10}$$

For period T of iteration, the length of the prologue is determined as follows.

$$prologue\ \ time = r_{max} \times T \tag{11}$$

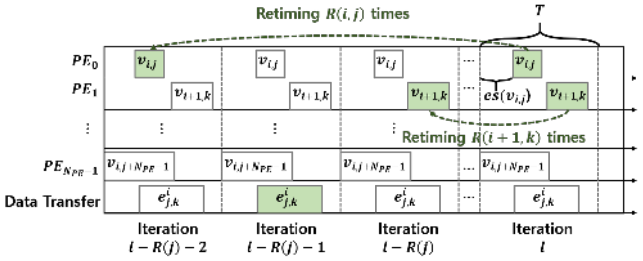


FIGURE 5. Exemplary allocation for DNN tasks with data dependency relationships.

Task scheduling has two main steps to completely utilize the parallelism of DNN applications. The first step is to generate an initial schedule that guarantees the threshold PE utilization ratio. The PE utilization ratio of an HMC C_m is defined as the non-idle time ratio of PEs in the C_m. Its value U_m can be expressed as:

$$U_m = \frac{\sum_{\forall v_{i,j} \to C_m} [ef(v_{i,j}) - es(v_{i,j})]}{T \cdot N_{PE}} \tag{12}$$

Algorithm 3 Initial DNN schedule genration, $TInit(\mathcal{M}_{\mathcal{P}^i \to \mathcal{C}_j})$

Input 1) Task partition mapping on HMC: $\mathcal{M}_{\mathcal{P}^i \to \mathcal{C}_j}$

Output 1) Initial task schedule sch_{init}

```
U_m = U_req + 1
while U_m < U_req do
    for m := 0 to N_C − 1 do
        for i := 0 to N_L − 1 do
            for n := 0 to N_P − 1 do
                if Map(P^i → C_m) then
                    Assign all v_{i,j}∈P^n to a PE in C_m
                    with the earliest available time
                end if
            end for
        end for
        find U_m using (12)
    end for
    Find the average PE utilization ratio U_avg
end while
```

Algorithm 3 provides the generation of the initial schedule. The proposed scheme calculates U_{avg}, the average value of U_m, and compares it with a predefined threshold utilization ratio U_{req}. If the PE utilization ratio is less than the threshold value, the proposed scheme allocates a DNN task set corresponding to other inputs to increase it. This process is repeated until the value of U_{avg} exceeds the value of U_{req}.

After the initial schedule generation, context switching between tasks is determined based on the communication latency between PEs and the PFB capacity. First, n intermediate results are sorted. In the sorted order, intermediate result I_m is the m-th task allocated to the PFB or DRAM. I_m is the m-th intermediate result to be allocated to the PFB or DRAM. Considering the allocation of the intermediate result I_m within the cache capacity I_m, it is possible to determine a method to allocate $m-1$ tasks with PFB capacity S. I_m allocation is determined by $m-1$ task allocation that minimizes the maximum retiming value and the remaining PFB capacity.

For intermediate result I_m, $\mathbb{R}[S][m]$ is defined as the maximum retiming value for the subset of m intermediate results I_1,I_2,...,I_m according to PFB capacity S. This gain represents a decrease in the maximum retiming value using the PFB capacity S. It is assumed that cr_m is the space required to allocate I_m to the PFB. As a result, I_m is placed in the PFB, and $\triangle R(m)$ is defined as a reduced retiming value. At this time, the optimal subproblem is defined as follows.

$$\mathbb{R}[S][m] = \begin{cases} 0, & (m = 0 \ \ or \ \ S = 0), \\ 0, & (m = 1 \ \ and \ \ cr_1 > S), \\ \triangle R(1), & (m = 1 \ \ and \ \ cr_1 \leq S), \\ \mathbb{R}[S][m-1], & (m \geq 2 \ \ and \ \ cr_1 > S), \\ max(\mathbb{R}[S][m-1], \mathbb{R}[S - cr_m][m-1] \\ \quad + max(\triangle R(m) - \triangle R(r_{max}), 0)), \\ & (m \geq 2 \ \ and \ \ cr_1 \geq S) \end{cases} \tag{13}$$

If m or PFB free space S is zero, $\mathbb{R}[S][m]$ becomes zero. This means that the intermediate result is not required for scheduling, or the PFB capacity is zero. As a result of the first intermediate processing, if the PFB free space cr_1 required to allocate I_1 is larger than S, then I_1 cannot be allocated in the PFB. If the space requirement of I_1 is less than or equal to S, the profit is initialized to the reduced retiming value $\triangle R(1)$. If the number of intermediate results is greater than one and the required PFB capacity is greater than S, this intermediate result cannot be allocated, and the retiming value cannot be reduced. In this case, $\mathbb{R}[S][m]$ is initialized to the maximum retiming value of the previous $m-1$ intermediate results.

Algorithm 4 presents the task allocation method reflecting this. The communication of the intermediate result whose $\triangle R(m)$ value is zero is not retimed; therefore, it does not affect the prologue time. Therefore, these intermediate results are allocated to DRAM to create a PFB free space for other tasks. The intermediate result with the highest maximum retiming value is selected and allocated to the PFB.

The intermediate result is inserted into queue Q. This queue defines the order of the allocation of the intermediate results. The proposed scheduling first sorts tasks by deadline. For tasks with the same a deadline, scheduling sorts m intermediate results in descending order according to $\triangle R(m)/cr_m$. intermediate results with low PFB capacity requirements and large $\triangle R(m)$ values have priority to be reserved in the PFB.

Algorithm 4 determines the schedule of each intermediate result by using the dynamic programming model in comparing $\mathbb{R}[S][m-1]$ and $\mathbb{R}[S - cr_m][m-1] + max(\triangle R(m) - \triangle R(r_{max}), 0)$. All intermediate results assigned to PFB have a relative retiming value $\triangle R(m)$. $\triangle R(r_{max})$ is the maximum relative retiming value among these. When $\mathbb{R}[S - cr_m][m-1] + max(\triangle R(m) - \triangle R(r_{max}), 0)$ is greater than $\mathbb{R}[S][m-1]$, it indicates that storing I_m in the PFB can reduce the maximum retiming value. When $\mathbb{R}[S][m-1]$ is greater than $\mathbb{R}[S - cr_m][m-1] + max(\triangle R(m) - \triangle R(r_{max}), 0)$, even if I_m is placed in the PFB, it does not reduce the retiming value, so it is placed in the DRAM. The dynamic programming model iteratively fetches each value of the matrix $\mathbb{R}[S][m]$, and this matrix can produce an optimal assignment for the intermediate result.

IV. EVALUATION

A. SIMULATION SETUP

To verify the performance of the proposed task scheduling, we used a ZSim-HMC [26], which implements the HMC-based PIM operation environment on an x86-64 simulator. According to the HMC 2.1 specification, the bandwidth of each memory was set to 10GB/s. The logic layer of each vault in the HMC included an in-order core operating at a 1GHz clock frequency, a 4MB PFB for caching intermediate results, a router with four virtual channels, and a network interface for packetization. Similar to Tesseract, it was assumed

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3077294, IEEE Access

Algorithm 4 DNN task scheduling, $TSch(\mathcal{M}_{\mathcal{P}^i \to \mathcal{C}_j})$

Input 1) Initial task schedule, sch_{init}
2) DNN graph $G(\mathcal{V}, \mathcal{E})$
3) PFB capacity S
4) Queue Q

Output 1) DNN task schedule sch

$m = 0$
for $\forall e^i_{j,k} \in \mathcal{E}$ **do**
  $I_m = enqueue(Q, I^i_{j,k})$
  obtain $\triangle R(m)$
  $m++$
end for
sort I_m in descending order by $\triangle R(m)/cr_m$
while Q is not empty **do**
  for $\forall C_m \subset \mathcal{C}$ **do**
    if $\mathbb{R}[S - cr_m][m-1] + max(\triangle R(m) - \triangle R(r_{max}), 0) > \mathbb{R}[S][m-1]$ **then**
      allocate I_m on the PFB
      $sch(I_m) = es(I_m)$
    else
      allocate I_m on the DRAM
      $sch(I_m) = es(I_m) - T$
    end if
    $dequeue(Q, I_m)$
  end for
end while
return sch

that 16 HMCs were connected in a dragonfly topology, and simulations were conducted by dividing the cases where HMCs had 16 and 32 vaults. The simulation was conducted with a server equipped with Intel(R) Xeon(R) Gold 6152 CPU @ 2.10GHz and 512GB DDR4 DRAM @2666MHz.

We ran a simulation using benchmarks of real-life DNN applications. DNN graphs for AlexNet, LeNet-5, ZFNet-1, ZFNet-2, VGG-F, VGG-16, and ResNet-50 were implemented using the deep learning framework Caffe [27]. The proposed scheduling model used SPARTA, a throughput-aware scheduling model, and data-level parallelism, and compared it with the Para-Net model that maximizes resource utilization in a single HMC. Because SPARTA and Para-Net performed scheduling in a single HMC, they were configured to operate independently in each HMC. The DNN processing time, prologue time, PE utilization ratio, and time cost for scheduling were compared.

B. SIMULATION RESULTS

1) DNN processing time

Table 3 and Fig. 6 present the DNN processing time of the DNN application of SPARTA [23], Para-Net [7], and the proposed scheduling for HMCs composed of 16 and 32 PEs. According to the experimental results, our approach reduced the DNN processing time by 44.73% and 18.19% compared to SPARTA and Para-Net, respectively. SPARTA does not adopt the retiming operation and operates while preserving the dependency between tasks. In addition, for SPARTA and Para-Net, HMCs operate independently. The DNN processing time was optimized based on effective task allocation, but it did not increase parallelism in allocating intermediate results.

TABLE 3. DNN processing time for SPARTA, Para-Net, and the proposed method

Benchmarks	DNN processing time (ns)					
	16-PE			32-PE		
	SPARTA	Para-Net	Proposed	SPARTA	Para-Net	Proposed
AlexNet	45,974	22,732	16,537	35,954	19,855	16,434
LeNet-5	55,734	27,998	24,831	49,431	22,760	19,682
ZFNet-1	103,896	47,329	40,229	79,920	44,995	39,394
ZFNet-2	49,950	28,686	24,845	35,964	26,988	17,081
VGG-F	34,881	29,014	20,493	31,329	26,126	18,437
VGG-16	38,363	31,901	27,115	34,267	28,709	24,435
ResNet-50	46,271	39,503	34,132	41,439	35,527	30,788

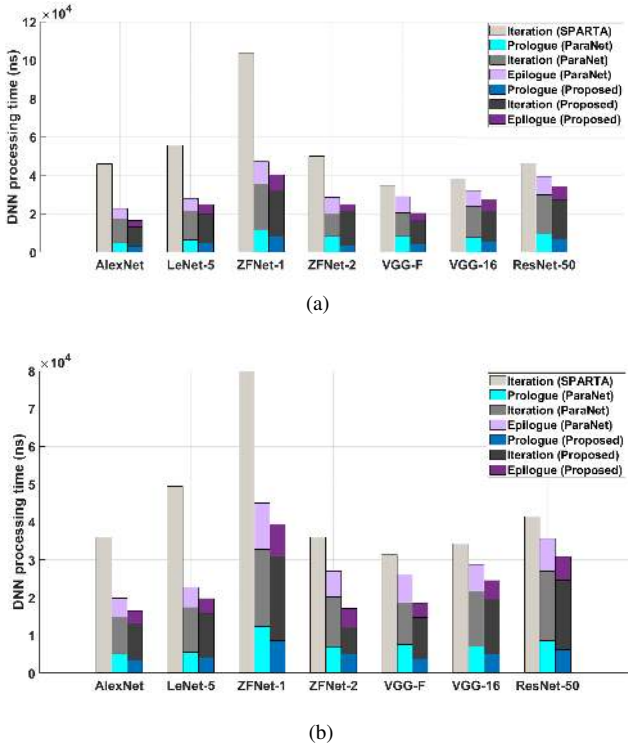


FIGURE 6. DNN processing time for SPARTA, Para-Net, and the proposed method using (a) HMC with 16 PEs and (b) HMC with 32 PEs

2) Prologue time

Prologue time is the time the DNN processing takes during the preprocessing step. This metric affects the total execution time and system throughput. This is presented in Table 4. Both the proposed scheme and Para-Net had a normalized prologue length because they utilized the concept of retiming. As the number of PEs increased, the prologue length decreased, and Para-Net had a longer prologue length than the proposed scheme because an initial schedule was generated for DNN processing. Our method and Para-Net integrate the operations of several periods into one iteration to guarantee PE utilization. Therefore, they introduced prologues for multiple periods. The prologue runs only once. Compared

to the advantages obtained by greatly reducing the DNN processing time in each iteration, prologue execution does not induce significant timing overhead. Furthermore, because the proposed scheme divides DNN tasks into partitions, it has a smaller task size and iteration time per HMC than Para-Net.

TABLE 4. Prologue time for Para-Net and the proposed method

Benchmarks	Prologue time (ns)			
	16-PE		32-PE	
	Para-Net	Proposed	Para-Net	Proposed
AlexNet	4,966	3,307	4,997	3,286
LeNet-5	6,549	4,966	5,475	3,936
ZFNet-1	11,595	8,045	12,279	8,478
ZFNet-2	8,369	3,416	6,889	4,969
VGG-F	8,359	4,098	7,531	3,687
VGG-16	7,816	5,423	7,024	4,887
ResNet-50	9,511	6,826	8,527	6,157

3) PE utilization ratio

The utilization ratio of the PEs indicates how the DNN application utilizes the computational resources of the PIM architecture. Because our target PIM architecture is memory-bounded, the PFB in the PE can be completely used. Then, the PE utilization ratio affects the throughput and the final task schedule. This is presented in Table 5. Our method achieved the highest utilization ratio for all benchmarks: 91.81%–97.40%, with an average ratio of 94.67%.

Compared to the proposed method, the SPARTA could only utilize 42.24% of the PE. The utilization ratio of SPARTA decreased as the number of PEs increased. These results indicated that SPARTA does not fully consider the data-level parallelism of DNN tasks and could not utilize all PEs. In contrast, our method and Para-Net utilized more than 90% of PE parallelism by allocating tasks to idle PEs. Therefore, the proposed method and Para-Net together achieved a utilization ratio similar to that of the proposed method.

TABLE 5. PE utilization ratio for SPARTA, Para-Net, and the proposed method

Benchmarks	PE utilization ratio					
	16-PE			32-PE		
	SPARTA	Para-Net	Proposed	SPARTA	Para-Net	Proposed
AlexNet	61.07%	88.30%	92.91%	54.67%	87.00%	91.81%
LeNet-5	60.32%	95.58%	95.48%	53.39%	91.56%	94.30%
ZFNet-1	61.65%	90.92%	95.02%	50.01%	92.26%	93.43%
ZFNet-2	60.84%	94.57%	96.68%	51.08%	81.13%	95.86%
VGG-F	62.47%	89.77%	97.40%	50.41%	82.78%	95.56%
VGG-16	59.16%	92.98%	97.18%	51.96%	89.84%	93.02%
ResNet-50	58.90%	92.19%	93.78%	48.07%	91.26%	92.98%

4) Time cost for scheduling

TABLE 6. Time cost to generate a schedule in SPARTA, Para-Net, and the proposed method

Benchmarks	Time cost to generate a schedule (s)					
	16-PE			32-PE		
	SPARTA	Para-Net	Proposed	SPARTA	Para-Net	Proposed
AlexNet	16.02	22.88	34.57	15.21	17.44	30.27
LeNet-5	15.69	28.29	45.28	12.89	26.75	39.08
ZFNet-1	16.64	29.47	50.15	9.01	31.37	57.21
ZFNet-2	19.89	29.31	53.71	16.04	20.81	37.45
VGG-F	18.14	30.08	58.11	19.40	32.45	54.30
VGG-16	20.43	34.58	60.17	18.19	30.83	63.99
ResNet-50	22.05	30.45	66.08	19.64	27.61	66.53

The execution time of each scheduling algorithm reflects its time complexity. These performance indicators were quantified, and the results are presented in Table 6. The time cost for scheduling in the proposed scheme was 303.82% and 178.57% higher than in SPARTA and Para-Net, respectively. Similar to Para-Net, the proposed method adopted a dynamic programming model and had a longer processing time than heuristic-based SPARTA. The proposed scheme had a longer time cost than Para-Net because it had to perform as many scheduling as the number of task partitions. However, the results revealed the time cost of the three scheduling algorithms to be of acceptable magnitude. This was because the proposed technique and the dynamic programming model of Para-Net were based on the initial schedule. Not all intermediate results are selected as inputs for the dynamic programming model. Accordingly, the time cost of the proposed technique did not incur significant overhead unlike other existing dynamic programming-based algorithms.

V. CONCLUSION

In this paper, we proposed scheduling considering interconnect latency variance that limits data-level parallelism of DNN in multi-HMC based PIM. Inter-HMC connections show various communication latencies according to transmission path and link load owing to packet-based communications. The proposed scheduling improved PE utilization by reducing the variance of communication latency through task partitioning and mapping to the HMC. The evaluation was conducted under a widely used deep learning application through the ZSim-HMC system-level simulator. The proposed method increased the schedule generation time by 178.57% compared to the existing methods but reduced the DNN processing time by 17.87%. This indicated that the proposed scheduling increased the utilization of PEs by reducing the variance of inter-HMC communications.

REFERENCES

[1] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *J. Mach. Learn. Res.*, vol. 12, pp. 2493–2537, Aug. 2011.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *in Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[3] M. A. Hanif, A. Manglik, and M. Shafique, "Resistive Crossbar-Aware Neural Network Design and Optimization," *IEEE Access*, Dec. 2020.

[4] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," *in Proc. 53rd ACM/EDAC/IEEE Des. Autom. Conf.*, Jun. 2016, pp. 1–6.

[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi "A scalable processing-in-memory accelerator for parallel graph processing," *in Proceedings of the 42nd Annual International Symposium on Computer Architecture*, Jun. 2015, pp. 105–117.

[6] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, and H. Yang, "GRAPHH: A processing-in-memory architecture for large-scale graph processing," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 38, no. 4, pp. 640—653, Apr. 2019.

[7] Y. Wang, W. Chen, J. Yang, and T. Li, "Exploiting parallelism for CNN applications on 3D stacked processing-in-memory architecture," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 589–600. Mar. 2019.

[8] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," *in Proc. MICRO*, 2013.

[9] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85-117. Jan. 2015

[10] J. T. Pawlowski, "Hybrid memory cube (HMC)," *in Proc. IEEE Hot Chips 23 Symp*., Aug. 2011, pp. 1–24.

[11] L. Han, Z. Shen, D. Liu, Z. Shao, H. H. Huang, and T. Li, "A novel ReRAM-based processing-in-memory architecture for graph traversal," *ACM Trans. Storage*, vol. 14, no. 1, pp. 9:1–9:26, Feb. 2018.

[12] C. Zhang, T. Meng, and G. Sun, "PM3: Power modeling and power management for processing-in-memory," *in Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, Feb. 2018, pp. 558–570.

[13] M. Imani, S. Gupta, and T. Rosing, "GenPIM: Generalized processing in-memory to accelerate data intensive applications," *in Proc. Des. Autom. Test Eur. Conf. Exhib.*, Mar. 2018, pp. 1155–1158.

[14] D. Gao, D. Reis, X. S. Hu, and C. Zhuo, "Eva-CiM: A System-Level Performance and Energy Evaluation Framework for Computing-in-Memory Architectures," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 39, no, 12, pp. 5011–5024, Dec. 2020.

[15] J. Wang, D. Park, Y. Papakonstantinou, and S. Swanson, "SSD instorage computing for search engines," *IEEE Trans. Comput.*, vol. PP, no. 99, p. 1, 2016, doi: 10.1109/TC.2016.2608818.

[16] Lenovo Group Ltd., 2018. [Online]. Available: http://www.lenovo.com/images/products/system-x/pdfs/datasheets/exflash_memory_channel_storage.pdf

[17] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," *In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 2017, pp. 751–764

[18] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," *in Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, Jun. 2016, pp. 243–254.

[19] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *in Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, Jun. 2016, pp. 1–13.

[20] 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 sol.-state circuits*, vol. 52, no. 1, pp. 127—138, Nov. 2016.

[21] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," *in Proc. IEEE 31st Int. Conf. Comput. Des.*, Oct. 2013, pp. 13–19.

[22] A. Dundar, J. Jin, V. Gokhale, B. Martini, and E. Culurciello, "Memory access optimized routing scheme for deep networks on a mobile coprocessor," *in Proc. IEEE High Perform. Extreme Comput. Conf.*, Sep. 2014, pp. 1–6.

[23] B. Donyanavard, T. Muck, S. Sarma, and N. Dutt, "SPARTA: Runtime task allocation for energy efficient heterogeneous manycores," *in Proc. 11th IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codes. Syst. Synthesis*, 2016, pp. 27:1–27:10.

[24] Y. Wang, R. Chen, R. Mao, and Z. Shao, "Optimally Removing Synchronization Overhead for CNNs in Three-Dimensional Neuromorphic Architecture," *IEEE Trans. Ind. Electron*, vol. 65, no, 11, pp. 8973–8981, Nov. 2018.

[25] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell system technical journal*, vol. 49, no. 2, pp. 291—307, Feb. 1970

[26] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer architecture news*, vol. 41, no. 3, pp. 475-486, Jun. 2013.

[27] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *in Proc. 22nd ACM Int. Conf. Mult.*, Nov. 2014, pp. 675–678

YOUNG SIK LEE (S'16) received the B.S. degree in electronic and electrical engineering from Sungkyunkwan University, Suwon, South Korea, in 2016, where he is currently pursuing the M.S. and Ph.D. degrees in electrical and computer engineering. His research interests include NoC, machine learning, and computer architecture.

TAE HEE HAN (M'08) received the B.S., M.S., and Ph.D. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 1992, 1994, and 1999, respectively. From 1999 to 2006, he was with the Telecom R&D center of Samsung Electronics, where he developed 3G wireless, mobile TV, and mobile WiMax handset chipsets. Since March 2008, he has been with Sungkyunkwan University, Suwon, Korea, as a Professor. From 2011 to 2013, he had served as a full-time Advisor on System ICs for the Korean Government. His current research interests include SoC architecture for artificial intelligence, emerging memory systems, embedded software, deep learning, NVM, PIM, and NoC.