Towards Memory-Efficient Processing-in-Memory Architecture for Convolutional Neural Networks

Yi Wang[1], Mingxu Zhang[1,3], Jing Yang[2]

[1]College of Computer Science and Software Engineering, Shenzhen University, China
[2]Experimental and Innovation Practice Center, Harbin Institute of Technology, Shenzhen, China
[3]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, China
yiwang@szu.edu.cn

Abstract

Convolutional neural networks (CNNs) are widely adopted in artificial intelligent systems. In contrast to conventional computing-centric applications, the computational and memory resources of CNN applications are mixed together in the network weights. This incurs a significant amount of data movement, especially for high-dimensional convolutions. Although recent embedded 3D-stacked Processing-in-Memory (PIM) architecture alleviates this memory bottleneck to provide fast near-data processing, memory is still a limiting factor of the entire system. An unsolved key challenge is how to efficiently allocate convolutions to 3D-stacked PIM to combine the advantages of both neural and computational processing.

This paper presents *Memolution*, a compiler-based *memo*ry-efficient data allocation strategy for convo*lution*al neural networks on PIM architecture. Memolution offers thread-level parallelism that can fully exploit the computational power of PIM architecture. The objective is to capture the characteristics of neural network applications and present a hardware-independent design to transparently allocate CNN applications onto the underlining hardware resources provided by PIM. We demonstrate the viability of the proposed technique using a variety of realistic convolutional neural network applications. Our extensive evaluations show that, Memolution significantly improves performance and the cache utilization compared to the baseline scheme.

CCS Concepts • **Computer systems organization** → *Embedded systems*; • **Hardware** → *Memory and dense storage*; *Neural systems*; *3D integrated circuits*; *Non-volatile memory*; *Communication hardware, interfaces and storage*; • **Software and its engineering** → *Allocation/deallocation strategies*; *Scheduling*

Keywords Processing-in-memory, neuromorphic computing, non-volatile memory, scheduling, parallel computing

1. Introduction

Deep convolutional neural networks (CNNs) have been successfully deployed in various application domains such as sensory processing, speech recognition, and image processing. In contrast to conventional computing-centric applications, CNNs are known to be both computational and memory intensive. Each convolution layer of CNN requires iterative use of the available processing engines. This produces a large amount of intermediate data that will be gradually streamed out to memory as the computation progresses. The intermediate data will be streamed back to the same processing engine upon the completion of each layer [3]. The data movement between computation and memory directly impacts performance and makes memory become the major bottleneck of the resource constrained system.

The emerging processing-in-memory (PIM) architecture offers a promising solution to the data movement challenge. State-of-the-art embedded PIM architecture stacks multiple layers of embedded DRAM (eDRAM) and allows processing engine close to the data in memory. This three-dimensional structure moves computation inside or near memory, reducing the expensive access latency for load and store operations. Although PIM alleviates the issue for the data movement, it has little reconfiguration or adaptive capabilities. The hardware resources could not proactively adapt to the workload of CNN applications. This problem is further exacerbated as the computational and memory resources of CNN applications are mixed together in the network weights and neuron activity [1]. The overall system performance is often degraded due to poor utilization of memory and insufficient memory to store the intermediate processing results (e.g., partial sums).

We observe that an unsolved key challenge is how to efficiently allocate convolutions to 3D-stacked PIM and fully exploit its computational power. As neural networks are highly parallelizable, both thread-level parallelism and data-level parallelism can be exploited. Although existing hardware-based approaches (e.g., ASIC or FPGA) offer one part of the solution, there is a need to adopt a compiler or runtime based approach to determine the mapping of parallelism onto the underlying hardware. This flexible allocation of convolutions allows no change to existing training process of CNNs, removing the burden of application-specific hardware design.

This paper presents *Memolution*, a *memo*ry-efficient allocation strategy for convo*lution*al neural networks on PIM architecture. We first conduct a theoretical analysis and show the impact of adjusting mapping of parallelism for hardware resources. This analysis captures the characteristics of neural network applications and determines the memory usage for intermediate processing results. Based on this analysis, Memolution jointly optimizes the allocation of convolutions and the intermediate processing results. It aims to fully utilize the valuable memory space and reduce the penalty for data movement. Memolution adopts an efficient data allocation scheme to exploit thread-level parallelism and assign convolutions to the appropriate processing engines.

LCTES'17, June 21–22, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-5030-3/17/06...$15.00
http://dx.doi.org/10.1145/3078633.3081032

To demonstrate the viability of the proposed technique, we use a variety of realistic convolutional neural network applications running on Caffe [14], an open-source deep learning framework. We compare Memolution with the baseline scheme [9] in terms of throughput, and evaluate the scalability and off-chip fetching penalty. Our extensive evaluations show that, Memolution can significantly improve the throughput (by 25.35% on average) and effectively utilize the PIM architecture with the negligible space and timing overhead.

The main contributions of this paper are:

- A hardware-independent data allocation strategy is proposed to enable the transparent mapping for CNN applications onto 3D-stacked PIM architecture.
- An analysis with the joint optimization of convolutions and the intermediate processing results has been conducted.
- As a proof of concept, we compare the proposed technique with representative schemes using a set of real-world CNN applications.

The rest of this paper is organized as follows. Section 2 introduces models and concepts used in this paper. Section 3 presents our proposed technique in detail. Section 4 presents experimental results. Section 5 discusses the related work. Section 6 concludes the paper and discusses future work.

2. Background and Motivation

2.1 Convolutional Neural Networks

Convolutional neural networks are one of the most popular machine learning techniques for high accuracy computer vision tasks. Neural network is inspired by the biological nervous system. It consists of a large number of highly interconnected processing elements (called neurons) to solve specific problems. CNNs consist of a combination of multiple layers, which can be broadly categorized as convolutional layers, activation layers, pooling layers, and fully-connected layers. Convolutional layers alternate with polling layers to achieve hierarchical neural networks, and convolutional layers dominate the computation time of a CNN.

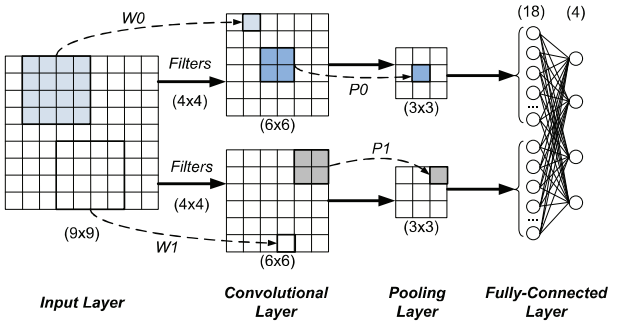

Figure 1. The standard procedure for CNN with multiple stacked layers.

Figure 1 illustrates the standard processing procedure for CNN. The processing of CNN starts with the input layer (image processing layer). The input layer is an optional pre-processing layer that provides the initial image and defines the training parameters of filters. Given an input feature map (9×9 neurons) in the first layer, the convolutional layer performs an inner product calculation, where pairwise multiplications are performed among the input elements (or neurons) and the filter weights (or synapses). A convolutional layer applies filters on the input feature maps to extract embedded visual characteristics and generate the output feature maps. Different filters are applied to each feature map with different weights ($W0$ and $W1$). In this example, the convolution kernel with the window size of 4×4 can produce two 6×6 output feature maps.

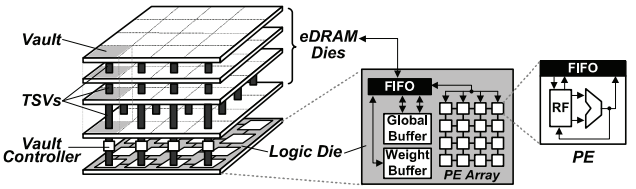

Figure 2. The emerging three-dimensional PIM architecture for CNN processing.

The pooling layer typically involves a simple maximum or average operation ($P0$ and $P1$) on a small set of input numbers. For example, the max-pooling in Figure 1 will select the maximum value in each 2×2 elements (i.e., four values). The output of the pooling layer is two 3×3 feature maps. The pooling operation enables position invariance over larger local regions and can effectively downsample the input image.

A fully-connected layer (inner product layer) also applies filters on the input feature maps as in the convolutional layers. It can be treated as a special kind of convolutional layers. The fully-connected layer arranges the output of the pooling layer (i.e., 2×3×3=18 elements). The outputs from pooling operations $P0$ and $P1$ are flattened and concatenated to produce a 1×32 input vector. This one dimensional vector is further multiplied with the weight matrix of size 18×4 to generate an output vector with size 1×4. Each of the convolutional layer or fully-connected layer is also immediately followed by an activation layer, such as ReLu (rectified linear units).

2.2 3D PIM Architecture

The emerging PIM architecture integrates multiple embedded DRAM (eDRAM) dies and one logic die in a 3D-IC architecture [15]. Figure 2 illustrates a typical PIM neuromorphic processing architecture. In this architecture, memory is organized into many vaults. Each vault is functionally independent and controlled by a vault controller (memory controller). Processing engines (PEs), a global buffer, and vault controllers are on the logic die. Each PE integrates a PE FIFO, an ALU datapath, and a register file (RF). Multiple PEs can concurrently communicate with multiple eDRAM vaults through high-speed through-silicon via (TSV) technology. Advanced 3D memory standards (e.g., Wide I/O-2 and Hybrid Memory Cube (HMC)) adopt the similar architecture.

On the logic die, two buffers (i.e., global buffer and weight buffer) are used to store the intermediate data and hide the eDRAM access latency. They can communicate with the PE array and eDRAM dies through the input and output FIFO. The global buffer stores the intermediate CNN processing results, such as feature maps. The weight buffer maintains the predefined convolutional weight for convolution layers or fully-connected layers. One major advantage of CNN is the use of shared weight in convolutional layers. The same filter weight is used for each pixel in the same layer. Therefore, a dedicated buffer is allocated for weights to minimize the overhead of accessing weights from the memory.

2.3 Application Model

A CNN application consists of a set of deterministic convolution operations. The functionality of each processing procedure in a CNN can be specified as a number of independent tasks. A set of concurrent convolution or pooling operations can communicate with each other by sending and receiving intermediate processing re-

sults via FIFO buffers. The rates at which CNN operations process and generate intermediate data are predefined and known at compile time. Therefore, a CNN can be represented by a synchronous data flow graph or a directed acyclic graph [6, 16].

A weighted graph $G = (V, E, R)$ is used to model the processing procedure of a CNN. $V = \{T_1, T_2, \ldots, T_n\}$ is the vertex set, and each vertex represents a convolution or pooling operation. Each vertex denotes a computing operation, such as inner product, addition, and comparison. One or more vertices can be allocated to one single PE. $E \subseteq V \times V$ is the edge set, and each directed edge, $(T_i, T_j) \in E$ $(T_i, T_j \in V)$, represents the intermediate processing results generated from vertex T_i and requested by vertex T_j. For each directed edge $(T_i, T_j) \in E$, an intermediate processing result $I_{i,j}$ denotes the corresponding data transfer from convolution operation T_i to convolution operation T_j. That is, the outputs (feature map and partial sums) of a convolution or pooling operation become the inputs of the next convolution or pooling layer. The intermediate processing result does not require computation resources. It only demands storage capacity in cache or eDRAM.

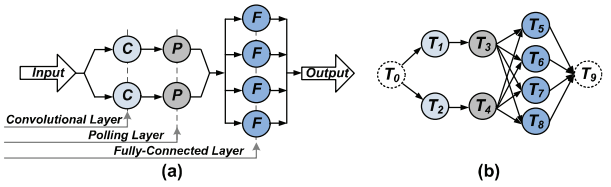

Figure 3. The data flow graph to model the processing procedure of a CNN.

The abstract of the standard procedure for a CNN (shown in Figure 1) can be modelled as a dataflow graph in Figure 3(a). The nodes with symbols C, P, and F represent a convolution operation, a pooling operation, and a fully-connected operation, respectively. The inputs of a CNN, including initial feature maps and predefined weights, have data dependency relationship with the following convolution operations. It can be treated as special preprocessing operation. We use a dummy vertex (T_0 in Figure 3(b)) to denote the inputs of the CNN. Similarity, the outputs of a CNN can also be represented as a dummy vertex (T_9) in the graph. As a result, the CNN in Figure 1 can be modelled as a data flow graph in Figure 3 with 10 vertices and 16 edges.

Each data flow graph represents the processing procedure of one initial input (e.g., a single pixel image). For a CNN application with multiple visual patterns, the same processing procedure will be applied in many times. Therefore, a CNN application can be further modelled as periodic dependent tasks. Let p be the period of each convolution operation T_i and that of each associated intermediate processing result $I_{i,j}$. Convolution operation T_i is associated with three tuples $T_i(s_i, c_i, d_i)$, where s_i is the start time, c_i is the execution time, and d_i is the deadline of T_i. For T_i in the ℓ-th period, T_i^ℓ, three tuples become $T_i^\ell(s_i^\ell, c_i^\ell, d_i^\ell)$, where $s_i^\ell = s_i + (\ell - 1) \cdot p$, $c_i^\ell = c_i$, and $d_i^\ell = d_i + (\ell - 1) \cdot p$, $\ell \geq 1$. Similarly, intermediate processing result $I_{i,j}$ in the ℓ-th period can be expressed by $I_{i,j}^\ell(s_{i,j}^\ell, c_{i,j}^\ell, d_{i,j}^\ell)$.

DEFINITION 2.1. *(Phase) Given a graph $G = (V, E, R)$, phase is the process of repeating a set of vertices $T_i \in V$ of the graph G.*

Phase is a computational procedure in which a cycle of convolution or pooling operations is repeated. For a CNN application, a phase will be repeated a specified number of times until a specific learning objective is achieved. The time interval between two successive occurrences of a recurrent phase on the same processing engine is regarded as the period of the graph G.

DEFINITION 2.2. *(Iteration) Given a graph $G = (V, E, R)$, iteration is a set of phases that are concurrently processed by a number of processing engines.*

An iteration consists of at least one phase. The processing time of one iteration is normally the same as that of a phase. In one iteration, convolution or pooling operations from two or more different phases are concatenated to form an iteration. It reflects the repeated execution of phases on multiple processing engines.

DEFINITION 2.3. *(Retiming) Given a graph $G = (V, E, R)$, retiming $R : V \mapsto \mathbb{Z}$ is a function that maps each vertex $T_i \in V$ to an integer $R(i)$. $R(i)$ is initially equal to zero; by retiming T_i once, $R(i) = R(i) + 1$ and one iteration of T_i is rescheduled into the prologue.*

Retiming is originally designed for register allocation problem in a digital circuit [19]. It moves the structural location of registers to improve its performance. We apply the concept of retiming to describe the delay model of intermediate processing results. For an intermediate processing result $I_{i,j}$, if it can not be allocated between the finishing time of T_i and the release time of T_j, at least one iteration of T_i will be rescheduled into the previous iteration. In this paper, retiming represents the delay of the execution of T_j relative to that of T_i. It can also be interpreted as the newly added iterations (called *prologue*) ahead of the original iterations.

2.4 Motivation

In convolutional neural networks, a convolution or pooling operation needs to perform the same procedure on the feature maps for many times. A significant amount of intermediate processing results are then generated. These intermediate processing results have to be carefully allocated, as the load operation from off-PE memory will cost inevitable delay for the following convolution operations. In fact, current PIM architecture can only provide 100-300KB cache capacity for the entire PE array [6]. How to effectively utilize this limited cache space becomes a critical issue.

Although hardware-based solutions such as FPGA and ASIC can provide application-specific design to fully take advantage of the cache space, they cannot offer a general framework that can accelerate the processing procedure for *any* CNN application on *any* hardware platform. We believe that a software-based solution is a promising direction, as it can provide hardware-independent support for various types of neural network applications. The applications do not have to be trained for a specific hardware platform. On the other hand, the hardware platform also does not have to know the unique properties of the application. It is the responsibility of software components (e.g., compiler, device driver) to handle the data allocation of application onto existing hardware resources.

There are several challenges to enable transparent data allocation and mapping to 3D-stacked PIM architecture. First, a compiler or operating system support is required to identify the characteristics of a specific neural network application. The data flow has to be interpreted and further modelled as fine-grained data structures. Second, a software/hardware co-design framework is required to perform data allocation to either computing or memory components. Since memory is the major bottleneck in 3D-stacked PIM architecture, the solution should take fully advantage of the resource-constrained shared buffer to alleviate the memory overhead.

The behavior of convolutional connections is deterministic. Therefore, it can be predicted at compile time. Compile time techniques do not burden runtime execution, and they can consider a much wider range of the applications than a runtime approach. This offers many opportunities to exploit the advanced parallel processing techniques to increase the amount of parallelism. The analysis of thread-level parallelism focuses on determining whether data ac-

cesses in later iterations are dependent on intermediate processing results produced in earlier iterations. This fine-grained parallelism helps improve the allocation with efficient usage of memory in 3D PIM architecture. These observations motive us to propose a novel memory-efficient strategy to transparently allocate convolutional connections in PIM architecture.

3. Memolution: Memory-Efficient Convolution Allocation on PIM Architecture

3.1 Overview

Neural networks are widely adept at various intelligent systems, but are limited in their ability to store intermediate data over long timescales [1]. Current CNN application has the trend of adopting more and more convolutional layers to perform deep learning. The intermediate processing results generated by each convolution or pooling operation will demand fairly large memory capacity. Memory is critical in the PIM architecture. The data movement in memory system directly impacts performance and power efficiency, two fundamental attributes of the entire system. If an intermediate processing result can be allocated to the PE's on-chip cache, no time delay will be incurred for the coming convolution operation.

Memolution aims to optimize the memory usage of convolutional neural networks on emerging PIM architecture. Memolution has two major design objectives. First, it fully utilizes the computational power of PIM architecture to speedup the CNN processing. Second, it captures the characteristics of CNN applications and performs convolution allocation to hide memory latency. Memolution provides a hardware-independent memory efficient data allocation strategy to coordinate between resource producer and resource consumer.

In this section, we first analyze the data allocation for intermediate processing results in Section 3.2. Then we present the convolution allocation to generate an initial schedule in Section 3.3. The rescheduling of the initial schedule to balance cache requirement is presented in Section 3.4. The target problem is further formulated to obtain an efficient solution in Section 3.5.

3.2 The Analysis of Data Allocation

This section analyzes how the allocation of a convolution operation impacts the corresponding intermediate processing results. For the target 3D PIM architecture, fetching data from eDRAM vault costs two times or more access latency than that from the PE array's global buffer. For an intermediate processing result $I_{i,j}$, a convolution or pooling operation T_i produce $I_{i,j}$, and it will be required by T_j. Each intermediate processing result $I_{i,j}$ is associated with two non-negative weights $L_{cache}(I_{i,j})$ and $L_{RAM}(I_{i,j})$, which denote the cost to allocate the intermediate processing result $I_{i,j}$ on the PE's shared buffer and the eDRAM in the 3D stacked memory, respectively. The cost L can be considered as the access latency to two different storage media, and $L_{cache}(I_{i,j})$ is much smaller than $L_{RAM}(I_{i,j})$. As c_j^i denotes the execution time of I_j^i, the terms c_j^i and L can be used interchangeably based on the content.

THEOREM 3.1. *For a pair of vertices $(T_i, T_j) \in E$ $(T_i, T_j \in V)$ in the ℓ-th iteration, by retiming T_i for $R(i)$ times and retiming T_j for $R(j)$ times, as long as the retimed vertex $T_{i,\ell-R(i)}$ is rescheduled at most one more iteration ahead of the retimed vertex $T_{j,\ell-R(j)}$, the associated intermediate processing result $I_{i,j}$ is always schedulable on PIM during the time span between the finishing time of $T_{i,\ell-R(i)}$ and the start time of $T_{j,\ell-R(j)}$.*

PROOF 3.1. *The initial computation schedule of a CNN application obeys the data dependency relations in the graph G. In the ℓ-th iteration of the initial schedule, the finishing time of T_i is no later than the start time of T_j. After retiming T_i for one time relative to the retimed vertex $T_{j,\ell-R(j)}$, T_i will be scheduled in iteration ℓ-$R(j)$-1, which is one iteration ahead of $T_{j,\ell-R(j)}$. Then the time span between the finishing time of $T_{i,\ell-R(i)}$ and the start time of $T_{j,\ell-R(j)}$ is always greater than or equal to the time of one iteration (period p), As all immediate processing results repeatedly execute in each iteration, in one iteration of time, at least one I_j^i has data dependency with T_i and T_j. Let this I_j^i be the immediate processing results of $T_{i,\ell-R(i)}$ and $T_{j,\ell-R(j)}$. Its start time is no earlier than the finishing time of $T_{i,\ell-R(i)}$, and its finishing time is no later than the start time of $T_{j,\ell-R(j)}$. Therefore, I_j^i is always schedulable on PIM during that time span. An example is given in Figure 4.*

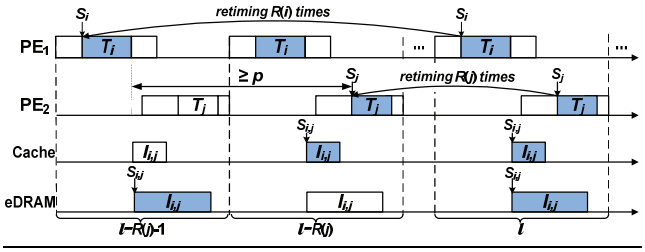

Figure 4. The exemplary allocation for convolutional connections with data dependency relations.

Retiming technique maintains the data dependencies for each pair of vertices and preserves the functionality of the CNN. Theorem 3.1 presents the upper bound of the maximum relative retiming value of each pair of vertices. This tight constraint presents a good property for rescheduling each vertex. This upper bound also implies that the maximum iteration added to the initial schedule is only one iteration of time. Let R_{max} be the maximum retiming value among all convolution operations, $R_{max} = \{\max R(T_i), T_i \in V\}$. The prologue time can be obtained by $R_{max} \times p$, and this denotes the preprocessing time before the loop kernel.

PROPERTY 3.1. *For a pair of vertices T_i and T_j, $(T_i, T_j) \in E$ $(T_i, T_j \in V)$, if $S_i + c_i + L_{RAM}(I_{i,j}) \leq S_j$, there is no need to perform retiming for T_i. That is, $R(i) = R(j)$.*

PROPERTY 3.2. *For a pair of vertices T_i and T_j, $(T_i, T_j) \in E$ $(T_i, T_j \in V)$, if $S_i + c_i + L_{cache}(I_{i,j}) \geq S_j$, T_i needs to retiming once and exactly once relative to T_j to ensure the schedulability of I_j^i. That is, $R(i) = R(j) + 1$.*

PROPERTY 3.3. *For a pair of vertices T_i and T_j, $(T_i, T_j) \in E$ $(T_i, T_j \in V)$, if $S_i + c_i + L_{RAM}(I_{i,j}) > S_j > S_i + c_i + L_{cache}(I_{i,j})$, the value of $R(i)$ depends on the allocation of I_j^i, and $R(i) - R(j) \leq 1$.*

Properties 3.1 to 3.3 further classify the allocation to either PE's shared buffer or off-PE eDRAM into three cases. For Property 3.1 and 3.2, the difference between $R(i)$ and $R(j)$ is fixed. For Property 3.3, the difference between $R(i)$ and $R(j)$ becomes zero if I_j^i can be allocated to PE's shared buffer; and becomes one if I_j^i has to be allocated to eDRAM. In this case, intermediate processing result I_j^i may postpone the execution of T_j.

3.3 Generating Initial Schedule with Fine-Grained Thread-Level Parallelism

Based on the analysis in Section 3.2, the allocation of vertices T_i and T_j determines the release time and it may cause extra iteration of retiming operation for T_i. As the first step of the proposed solution, Memolution generates an initial schedule with fine-grained

thread-level parallelism. The initial schedule takes into account the allocation of immediate processing results and tries to avoid unnecessary retiming operations.

For the fine-grained thread-level parallelism, a CNN application is modelled as a data flow graph. We observe that the fine-grained partition of vertex will increase both the number of vertices and edges in the graph. By appropriately merging several vertices, it can reduce the complexity to process the graph and still preserve the functionality of application. A vertex T_u that can be merged with another vertex T_v has the following property. First, both T_u and T_v are not dummy vertices; Second, T_u and T_v can be assigned to the same processing engine without causing extra latency in pipelining, i.e., $c_u + c_v \leq max\{c_i\}, \{T_u, T_v, T_i\} \in V$.

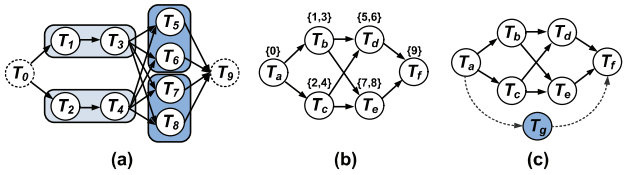

Figure 5. The merge of vertices and the general data flow graph with asymmetrical structure.

There are two typical cases for vertex merge operations. For the first case, called *sequential merge*, both vertices T_u and T_v, $\{T_u, T_v\} \in V, (T_u, T_v) \in E$, have only one incoming (or outgoing) edge. Vertex T_u is the predecessor of vertex T_v. T_1 and T_3 in Figure 5(a) shows this case. For the second case, called *parallel merge*, vertex T_u and vertex T_v have the same number of incoming edges and the same number of outgoing edges. Vertex T_u and vertex T_v are in the same layer. T_5 and T_6 in Figure 5(a) shows this case. After the merge operation, the original data flow graph in Figure 5(a) is transformed into a new graph in Figure 5(b).

The merged vertex represents the combination of two computation vertices. Its execution time is the total execution time of the two vertices. For example, vertex T_b in Figure 5(b) represents the merged vertex of vertices T_1 and T_3 in Figure 5(a). The merge operation will also merge the edges in the graph. The associated intermediate processing results will denote the total data transfer for incoming edges (or outgoing edges) of vertices T_u and T_v.

The data flow graph of a CNN application normally includes the core subgraph with symmetrical structure. For example, let two dummy vertices (T_a and T_f in Figure 5(b)) be the symmetrical axis, the vertices above the axis have the mirrored vertices below the axis. For example, T_b and T_c form the symmetrical structure. They have exactly the same number of incoming (outgoing) edges, and the execution time of these two vertices are the same. Besides this core structure, the data flow graph may also have several individual vertices that could not find mirrored vertex in the graph. We add such an individual vertex (T_g) to the merged graph, and a new data graph is illustrated in Figure 5(c).

The allocation of vertices needs to consider the intermediate processing result, which costs the latency $L_{cache}(I_{i,j})$ or $L_{RAM}(I_{i,j})$. In order to hide this latency, Memolution interleaves the execution of convolution connections from two different processing procedure of CNNs. The dependent computations are separated from one another, increasing the possibility that the successor vertex can be scheduled without stalls. An example is illustrated in Figure 6, where T_a is immediately followed by another T_a from another processing procedure of the CNN. This step is quite similar to loop unrolling. The major difference is that our technique only repeats symmetrical vertex (e.g., T_a) for once and the individual vertex (e.g., T_g) can be allocated based on the scheduling policy.

The interleaved execution of two processing procedures form the basic execution block, called phase. Figure 6 illustrates the

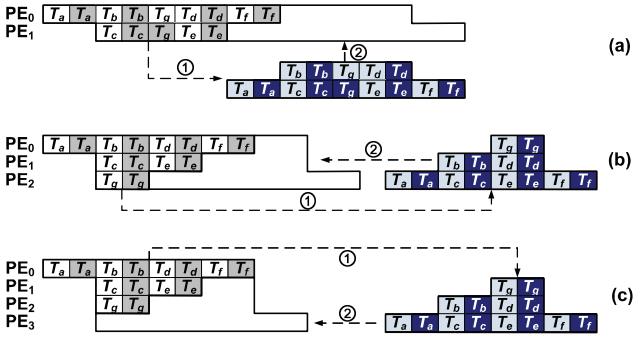

Figure 6. Generate a phase of convolution connections, and use *rotation* and *insertion* to form an iteration.

phase of convolution connections on 2, 3, and 4 PEs for the task graph in Figure 5(c). Additionally, by concatenating a set of phases, an iteration can be formed. This involves two steps. The first step, called *rotation*, tries to generate a complement phase that can improve the utilization of PEs. For Figure 6(a) with two PEs, T_a, T_b, T_d, and T_f are assigned to PE_0, and they will be assigned to PE_1 in the complement phase.

Once a complement phase is selected, the second step, called *insertion*, allocates each vertex to the newly created complement phase according to the allocation order. The complement phase attempts to move its schedule as early as it can to eliminate any possible idle PE. The initial phase and the complement phase are packed into one iteration. Figure 7 shows two phases with 4 PEs and the corresponding iteration.

The iteration determines the allocation of vertices, and it will be repeatedly executed in the successive time slot. As a result, the initial schedule of vertices on a number of PEs can be generated. The corresponding cache requirement to allocate each intermediate processing result on PE's shared buffer can also be known. Figure 7 illustrates the life span of each intermediate processing result. Assume that each vertex takes one time unit for processing and each intermediate processing result causes the same amount of buffer space. Then the total cache requirement in each iteration can obtained.

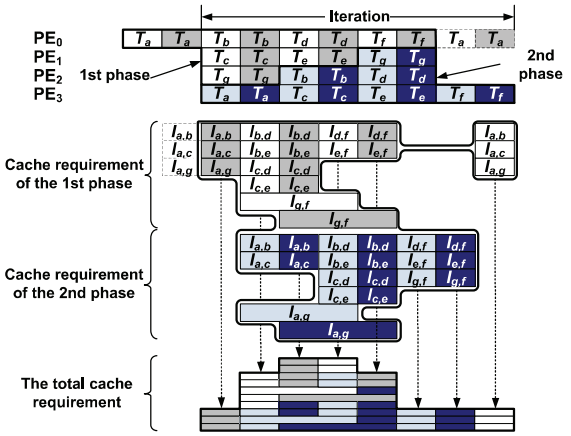

Figure 7. An iteration of the initial schedule and its cache requirement.

3.4 Balancing of Cache Requirement

As Memovolution adopts fine-grained thread-level parallelism, multiple iterations can execute in parallel to scale-up all available processing engines. The current neuromorphic processing architecture normally provides adequate computing resources. Memovolution adopts combinations of multiple iterations to speedup convolution processing. Figure 8 shows the system with 8 PEs, which can be divided into two groups of PEs. Each group contains 4 PEs. Using the data allocation from Figure 7, the first and the second iteration can be operated in parallel.

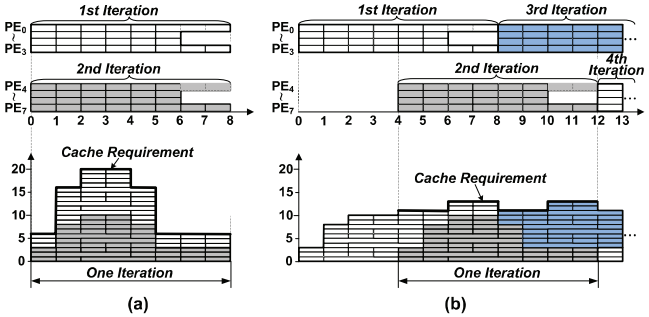

Figure 8. The allocation of iteration impacts the total cache requirement.

The partition of PEs offers various combinations of iterations. For example, a system with 8 PEs can have two iterations with 4 PEs per iteration. It can also be the combination of four identical iterations with 2 PEs per iteration. There exists the trade-off in terms of PE's utilization versus the time span to maintain intermediate processing results. As a memory-efficient design, Memolution fully exploits the thread-level parallelism to reduce the latency and cache requirement for intermediate processing results. Assume that each convolutional or pooling layer consists of at most n vertices, Memolution prefers to select the iteration with n or $n+1$ PEs. The PE's utilization is compensated by the adoption of rotation and insertion in the creation of an iteration.

The cache requirement represents the best-case buffer size, where the PE's shared buffer has enough capacity to hold all intermediate processing results. The allocation of iteration impacts the total cache requirement. Figure 8 shows this behavior graphically. In Figure 8(a), both the first and the second iteration execute in the time span from 0 to 8, and the maximum cache requirement is 20 units. Since the shape of a CNN graph is big in the middle but is small at both ends, the maximum cache requirement occurs at the middle of each iteration.

This unevenly distribution for cache requirement will cause extra latency, as the PE's shared buffer may not have enough capacity to concurrently store all intermediate processing results at the time with peak cache requirement (e.g., time span 2-4 in Figure 8(a)). In this case, several intermediate processing results have to be allocated to off-PE's eDRAM, which may cause the retiming operation for one iteration. Memolution solves this problem by rescheduling the start time of the iteration. As shown in Figure 8(b), the start time of iterations on PE4-PE7 is reallocated to time unit 4. Thus, the peak cache requirement is significantly reduced to 13 units, making the balanced requirement for PE's shared buffer in each iteration.

To facilitate the analysis of cache requirement, Memolution partitions the PEs into N_g identical groups, and each group adopts the same allocation of vertices in each iteration. Algorithm 3.1 presents the procedure to obtain the start time for each group of iterations. It handles three cases. For the case that the PE's shared buffer can hold all intermediate processing results (Line 1), there is no need to postpone the execution for each group of iteration (Line 2). The second case shows that, the total capacity of the shared buffer could not satisfy the requirement no matter how the schedule postpones (Line 4). In this case, several intermediate processing results have to be reallocated to eDRAM (Line 5). For the last case, it is possible to overlap the peak and the valley of cache requirment for different groups of iterations. Both the second and the last case will cause the delay of the execution (Line 7).

Algorithm 3.1 Obtain the start time for each group of iterations of a CNN application.

Input: An initial schedule S_{init} of an iteration, the period p, N_{ph} phases in each iteration, N_g identical PE groups, the capacity of the shared buffer C_t, the maximum C_{max} and the minimum C_{min} cache requirements in one iteration, the start time gs_0 of the schedule.

Output: The start time gs_i of the i-th group $i \in [1, N_g - 1]$.

1: **if** $C_{max} \cdot N_g \leq C_t$ **then**
2: $\quad gs_i \leftarrow gs_0, i \in [1, N_g - 1]$.
3: **else**
4: $\quad$ **if** $(C_{max}+C_{min}) \cdot N_g > 2C_{total}$ **then**
5: $\quad\quad$ Allocate intermediate processing results to eDRAM with at least $\frac{1}{2}$ $(C_{max}+C_{min})$-C_t capacity.
6: $\quad$ **end if**
7: $\quad gs_i \leftarrow gs_0 + \frac{1}{N_g} \cdot p \cdot i, i \in [1, N_g - 1]$.
8: **end if**

3.5 Convolution Allocation with Prologue Reduction

This section presents the allocation for convolution connections with minimum prologue. Based on the analysis of retiming value, to minimize the preprocessing time in the prologue is equivalent to the problem of reducing the maximum retiming value of all convolution operations in the application. In previous sections, we have formed the iteration and obtained the initial schedule. The PEs are further partitioned into several groups, and each group handles one iteration. Then the target problem is transformed into the reduction of the maximum retiming value of different groups.

The latency $\Delta L(i,j)$ of retiming is affected by the start time of the iteration. The reallocation of different vertices may incur different latency. As shown in Figure 9, assume that the cache requirement for intermediate processing results exceeds the capacity of the on-PE's shared buffer during time units 14-16. PE0-PE3 and PE4-PE7 form two PE groups. The group in PE0-PE3 initially starts at time unit 8 (the third iteration). By retiming T_a once, a new iteration (the first iteration) is added in the prologue, and the schedule will be started at time unit 0. This will cause the latency with 8 time units (from unit 0 to unit 8). On the other hand, the group in PE4-PE7 initially starts at time unit 12 (the fourth iteration). Retiming a vertex in the fourth iteration will only incur the latency with 4 time units (from unit 4 to unit 8).

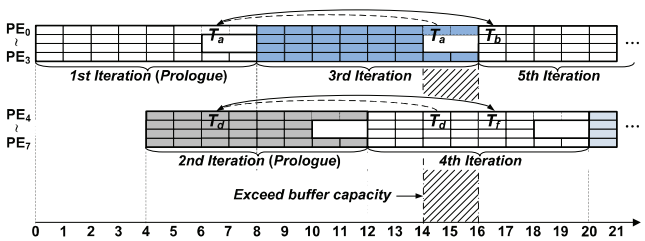

Figure 9. Retiming of vertices and the extra processing in prologue.

Table 1. Total execution time of LS, SPARTA [9] and our Memolution on 32, 64, and 128 processing elements.

	32-PE			64-PE			128-PE			IMP	IMP over
	LS	SPARTA	Memolution	LS	SPARTA	Memolution	LS	SPARTA	Memolution	over LS	SPARTA
speech-1	160,000	80,224	71,416	76,320	53,664	35,990	38,240	24,960	18,220	53.52%	23.64%
speech-2	255,791	130,208	119,756	127,985	87,100	60,068	64,082	43,784	30,224	53.03%	23.34%
character	227,211	90,180	88,406	113,685	72,360	44,344	56,922	33,120	22,312	60.96%	24.44%
image-processing-1	368,000	145,406	127,686	175,536	97,498	64,248	87,952	49,184	32,530	63.91%	26.72%
image-processing-2	276,054	130,364	116,734	132,400	87,308	58,736	65,207	40,664	29,738	55.92%	23.35%
image-processing-3	193,488	90,288	85,952	92,800	72,504	43,248	45,704	33,264	21,896	53.69%	26.44%
path-planning	278,000	130,416	123,556	132,606	102,714	62,560	66,442	52,728	31,694	53.56%	28.08%
statistical-modeling	347,778	165,462	147,968	166,800	109,265	74,452	82,149	51,744	37,694	55.64%	23.20%
video	325,000	150,660	147,022	162,760	129,690	73,976	81,640	61,110	37,454	54.48%	28.03%
remote-sensing	264,378	120,384	120,124	126,800	103,608	60,594	62,449	48,744	30,678	52.51%	26.27%
Average										55.72%	25.35%

Not all intermediate processing results get involved in this prologue reduction process. In Section 3.4, we have generated an initial schedule and obtained the start time of each group of iterations. Only the intermediate processing results that cause the cache requirement beyond the capacity of the shared buffer will influence the prologue. These intermediate processing results form a subset of all vertices in the graph.

Algorithm 3.2 presents the procedure to obtain the data allocation of intermediate processing results. The objective is to obtain the schedule with the minimum prologue. The first step gets the relative retiming value $(R(i) - R(j))$ for each intermediate processing result $I_{i,j}$ when it is allocated to eDRAM. Theorem 3.1 and three Properties in Section 3.2 can help derive each relative retiming value. If the retiming value is equal to zero, this intermediate processing result will not incur extra retiming operations. For other cases, $I_{i,j}$ will introduce the extra preprocessing with latency $\Delta L(i,j) = p \cdot (1 - \frac{u}{N_g}) \cdot (R(i) - R(j))$, where p is the period, N_g is the number of PE groups, and $I_{i,j}$ is allocated to group u.

Memolution sorts these n intermediate processing results V_{sch}, $V_{sch} \subseteq V$, according to $\Delta L(i,j)/\Delta C(I_{i,j})$, where $\Delta C(I_{i,j})$ is the difference between the space requirement for $I_{i,j}$ allocating to cache or eDRAM. This metric aims to provide a cost-effective solution to minimize the penalty to place the intermediate processing results and fully utilize the cache space. As long as the cache requirement $\sum C_{req}$ is beyond the total cache capacity C_t, the intermediate processing result will be assigned to eDRAM. Then the rest of the intermediate processing results will be remained in the cache, which do not require extra retiming operations.

Algorithm 3.2 Generate a schedule for a CNN application with the shortest prologue.

Input: A set of n intermediate processing results V_{sch}, $V_{sch} \subseteq V$, N_g identical PE groups, the intermediate processing result $I_{i,j}$ allocated to group u, the capacity of the shared buffer C_t, and a queue Q.

Output: The allocation and the start time $s_{i,j}$ of each intermediate processing result $I_{i,j}$.

1: **for** each $I_{i,j} \in V_{sch}$ **do**
2: **if** $R(i) - R(j) = 0$ **then**
3: $\Delta L(i,j) \leftarrow 0$.
4: **else**
5: $\Delta L(i,j) \leftarrow p \cdot (1 - \frac{u}{N_g}) \cdot (R(i) - R(j))$.
6: **end if**
7: $Enqueue(Q, I_{i,j})$.
8: $\sum C_{req} \leftarrow \sum C_{req} + C_{cache}(I_{i,j})$.
9: **end for**
10: **for** $I_{i,j} \in Q$ **do**
11: Sort $I_{i,j}$ in descending order by $\Delta L(i,j)/\Delta C(I_{i,j})$.
12: **end for**
13: **while** $\sum C_{req} - C_t > 0$ and $I_{i,j}$ is the head of Q **do**
14: Allocate $I_{i,j}$ on eDRAM.
15: $\sum C_{req} \leftarrow \sum C_{req} - C_{cache}(I_{i,j})$.
16: $Dequeue(Q, I_{i,j})$.
17: **end while**

4. Evaluation

4.1 Experimental Setup

We conduct experiments using a set of benchmarks from real-life CNN applications. The widely used deep learning framework Caffe [14] is used to abstract graph representation. The neural network model obtained from the training phase is sent to Caffe, and we added notations and variables to track the timing and other parameters of each convolution or pooling operation. Among these benchmarks, *speech-1* and *speech-2* are voice recognition applications to identify the speech of a dedicated speaker; *character* is a character recognition application to recognize the bitmap pattern of handwritten characters; *image-processing-1* to *image-processing-3* are image processing applications to identify the major object in the images; *path-planning* is an path planning application that finds the shortest path in the map; *statistical-modeling* models the pattern in the stock market; *video* is a video recognition application to track the moving object in the movie; *remote-sensing* is an application to process the infrared remote sensing images.

Our experiments are conducted based on the PIM architecture of Neurocube [15]. The Neurocube neuromorphic architecture is an extension of Micron's Hybrid Memory Cube to support neural computing. It provides high-density 3D stacked memory to integrate multiple tiers of DRAM and a number of processing engines. In our experiments, the architecture is configured to contain up to 128 processing engines with cross-bar interconnection.

4.2 Experimental Results

1) Processing Time: Table 1 presents the processing time of list scheduling (LS), SPARTA [9] and the proposed Memolution under 32, 64, and 128 PEs. SPARTA is a throughput-aware task allocation approach for many-core platforms. SPARTA collects sensor data to characterize tasks and uses this information to prioritize tasks when performing allocation. Therefore, it is selected as the baseline scheme. In order to make a fair comparison, SPARTA is combined with the retiming technique to preserve the data dependencies. The schemes LS, SPARTA, and our Memolution adopt the same input graphs. The experimental results are specified in microseconds. From the experimental results, Memolution can achieve average reductions of 55.72% and 25.35% compared to LS and SPARTA, respectively. These reductions come primarily from fine-grained parallelism. In Memolution, the interleaved scheduling of two processing procedures can provide enough time

Table 2. The number of intermediate processing results that are allocated to on-PE's shared buffer for LS, SPARTA [9] and our Memolution.

	32-PE			64-PE			128-PE			IMP	IMP over
	LS	SPARTA	Memolution	LS	SPARTA	Memolution	LS	SPARTA	Memolution	over LS	SPARTA
speech-1	39	44	86	39	23	86	39	19	86	2.21x	3.41x
speech-2	19	23	114	19	16	115	19	16	115	6.04x	6.44x
character	23	22	114	23	18	115	23	18	115	4.99x	5.99x
image-processing-1	34	27	70	34	19	70	34	19	70	2.06x	3.32x
image-processing-2	33	27	67	33	20	67	33	19	67	2.03x	3.12x
image-processing-3	44	32	84	44	20	84	44	15	84	1.91x	4.14x
path-planning	71	61	111	71	48	111	71	48	111	1.56x	2.15x
statistical-modeling	41	51	60	41	30	60	41	27	60	1.46x	1.80x
video	15	16	45	15	14	50	15	15	55	3.33x	3.35x
remote-sensing	32	27	63	32	23	63	32	21	63	1.97x	2.69x
Average										2.76x	3.64x

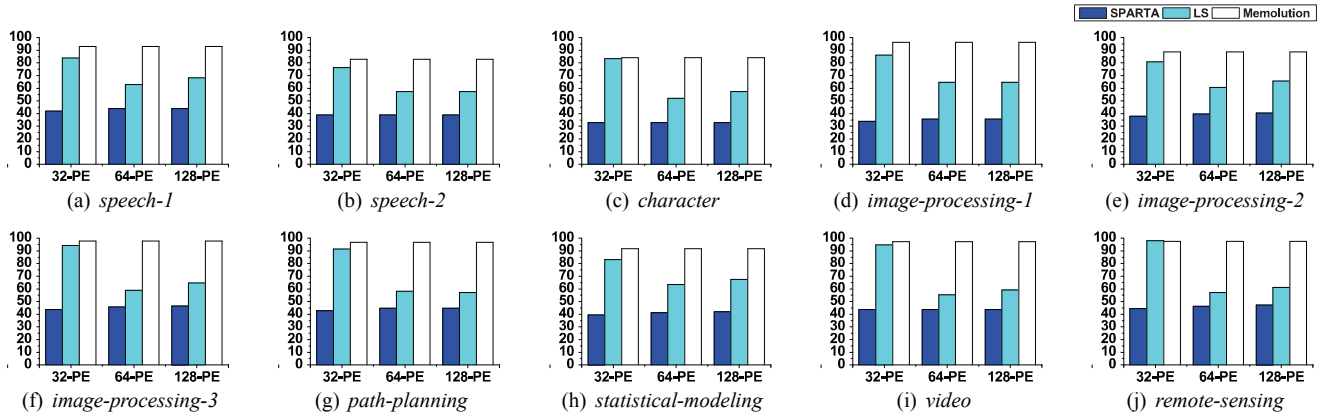

Figure 10. The utilization ratio of processing elements for LS, SPARTA [9] and Memolution on 32, 64, and 128 processing elements.

span to allocate intermediate processing results without incurring extra timing overhead of the entire schedule.

2) Utilization Ratio of PEs: Figure 10 presents the utilization ratio of processing elements. Not surprisingly, Memolution can achieve the highest utilization ratio, ranging between 82.9% and 97.8%. Memolution adopts the rotation and insertion process to generate the initial schedule. An iteration can be generated from two or more complementary phases. The LS scheme does not introduce inter-iteration data dependencies, so some PEs have to wait for the intermediate processing results. This causes idle PEs, resulting in the lowest utilization ratio. Although SPARTA is combined with retiming to further reschedule some iterations into the prologue, the fine-grained reallocation of intermediate processing results are not performed in SPARTA. Therefore, it can not generate a good initial schedule for CNN applications.

3) Cache Efficiency: As a memory-efficient optimization technique, Memolution aims to provide a data allocation with the minimum prologue for intermediate processing results. Table 2 presents the number of intermediate processing results that are allocated to the on-PE's shared buffer. The impact of cache efficiency is highly correlated to the utilization ratio of PEs. The higher utilization ratio of PEs denotes the better usage of the on-PE's shared buffer. Due to the nature of the shared buffer, its capacity should be fully utilized for all kinds of schemes. However, some intermediate processing results may occupy the valuable space, which may prevent the efficient usage of the cache. Memolution provides the analysis on the start time and finishing time of intermediate processing results. Therefore, it can efficiently use both computing and memory resources in the PIM architecture. From the experimental results, Memolution can allocate 2.76x and 3.64x of intermediate processing results on the PE's shared buffer.

4) Overhead of Retiming: Retiming operation will introduce preprocessing stage of convolutional connections, which will impact the total processing time of the application. This overhead is quantified, and the experimental results are illustrated in Table 3. On average, Memolution reduces the number of retiming operations by 61.17% and 80.52% for 64 and 128 PEs, respectively. We observe that the retiming value of SPARTA is not sensitive to the number of PEs, while our Memolution can get better scalability with a larger number of PEs. This is due to the fact that, Memolution uses a set of optimization techniques to generate the initial schedule and improve the utilization of PEs. The retiming operation is combined with these techniques to jointly optimize the allocation of both convolutional connections and intermediate processing results.

The fine-grained parallelism also helps reduce this retiming overhead. For the baseline scheme SPARTA, it combines the simple retiming technique without applying any further optimization techniques. Therefore, SPARTA will introduce more retiming operations. Although Memolution may introduce multiple iterations of prologue, compared to the benefit gained from the significant reduction of processing time in each iteration, this overhead is negligible and can be rectified at the compile time.

5. Related Work

FPGA/ASIC accelerators for CNNs. FPGA and ASIC accelerators provide hardware-based neuromorphic platforms to speedup the data processing of CNN applications. There have been a number of FPGA implementations [3, 25, 26] and ASIC accelerators [5, 22]. Compared to general purpose graphics processing unit (GPGPU), FPGA platforms can achieve much higher power and

Table 3. The number of retiming operations for SPARTA [9] and our Memolution on 64 and 128 processing elements.

	64-PE		128-PE	
	SPARTA	Memolution	SPARTA	Memolution
speech-1	192	141	192	70
speech-2	260	47	260	23
character	216	35	216	17
image-processing-1	464	50	464	25
image-processing-2	364	138	364	69
image-processing-3	288	136	288	68
path-planning	408	391	416	207
statistical-modeling	520	175	528	87
video	630	174	630	87
remote-sensing	432	119	432	59
Average reduction		*61.17%*		*80.52%*

computational efficiency [21]. However, for FPGA platforms, once a neural network engine is synthesized, it cannot be programmed on-line [15]. ASIC accelerators can improve the training process and obtain even better performance for specific CNN applications [8]. It is limited by the scalability of on-chip memory or interface with external memory.

The proposed Memolution is different from these works. It is a software-based technique, which provides a hardware-independent task allocation interface. Memolution is not designed for a specific hardware platform or a specific CNN application. The underlying hardware resources are abstracted as computing units. Therefore, Memolution can still enjoy the accuracy and computational efficiency of ASIC or FPGA platforms and provide scalability of parallelism.

PIM for data-intensive workloads. Processing in memory architecture integrates computing units within or near memory. The resurgence of PIM and near data processing is motivated by new technologies such as 3D-stacked memory, accelerator use in specific domains, and data-intensive workloads with high degrees of parallelism. The hardware implementation of PIM can be the stacked memory modules like Hybrid Memory Cubes [20], flash-based memory-channel NVDIMM like Lenovo/IBM's eXFlash [17] and Viking's ArxCis-NV [23], or emerging memory technology like metal-oxide resistive random access memory (ReRAM) [7].

There have been a number of studies that focus on the performance improvement of PIM for data-intensive workloads [2, 11–13]. These works offer solutions to the problem of mapping workloads to PIM computing units. As a general data allocation strategy, our strategy can be applied to different PIM architectures to fully utilize their computing resources. Our strategy can be combined with these PIM frameworks to further capture the characteristics of workloads and obtain a better initial schedule.

Scheduling for dataflow applications. Streaming applications and other data-intensive applications are widely found in embedded systems. Streaming applications are commonly modelled as synchronous data flow graphs or directed acyclic graphs [10, 24]. Previous work on scheduling streaming applications onto multicore systems aims to exploit the parallelism of architecture to improve the performance, memory management, energy efficiency, and reliability [4, 18]. Even though a CNN application can also be represented as a directed acyclic graph, it has many special properties. The scheduling techniques for general directed acyclic graphs may not be applicable to solve the data allocation problem in CNNs.

Chen et al. proposed a CNN accelerator called Eyeriss [6], in which the CNN application is modelled as a directed acyclic graph. It exploits zero valued neurons by using run length coding for memory compression and data gating zero neuron computations to save power. Our Memolution aims at the utilization of cache for intermediate processing results. Fine-grained thread-level parallelism is adopted to hide the latency caused by intermediate processing results, which could also take advantage of the acceleration framework of Eyeriss to further improve energy efficiency.

6. Conclusion

This paper presents a novel data processing framework, called Memolution, to optimize the memory usage of convolutional neural networks on emerging processing-in-memory architecture. Memolution exploits fine-grained thread-level parallelism to fully utilize the processing engines in PIM and minimizes the data movement for fetching intermediate processing results from off-PE external memory. The data processing framework first generates an initial allocation of convolution operations that jointly considers computing resource and memory usage. An efficient data allocation scheme for intermediate processing results is then obtained. We demonstrate the effectiveness of our approach by using a set of CNN applications on Caffe, an open-source deep learning framework . Experimental results show that the proposed approach can effectively improve processing speed of CNNs and minimize the off-PE data transfers.

In the future, we plan to work on the parameterized compiler design space exploration and obtain accurate models of CNN applications. We also plan to investigate new architectural features for virtualization environments in PIM, which may improve the predictability of memory usage and further boost the processing speed of deep learning applications.

Acknowledgments

The work described in this article is partially supported by the grants from National Natural Science Foundation of China (61502309), Guangdong Natural Science Foundation (2014A030310269, 2014A030313553 and 2016A030313045), Shenzhen Science and Technology Foundation (JCYJ20150525092941059 and JCYJ20150529164656096), Natural Science Foundation of SZU (701-00036 0055905, 701-00037134, and 827-000073), and State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences (CARCH201608).

References

[1] A. Graves et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):1–6, Oct. 2016.

[2] B. Akin, F. Franchetti, and J. C. Hoe. Data reorganization in memory using 3D-stacked DRAM. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA '15)*, pages 131–143, June 2015.

[3] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*, pages 1–12, Oct 2016.

[4] K. M. Barijough, M. Hashemi, V. Khibin, and S. Ghiasi. Implementation-aware model analysis: The case of buffer-throughput tradeoff in streaming applications. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '15)*, pages 11:1–11:10, 2015.

[5] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, pages 269–284, 2014.

[6] Y. H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA '16)*, pages 367–379, June 2016.

[7] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *2016*

ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA '16), pages 27–39, June 2016.

[8] F. Conti and L. Benini. A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE '15)*, pages 683–688, March 2015.

[9] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt. Sparta: Runtime task allocation for energy efficient heterogeneous many-cores. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '16)*, pages 27:1–27:10, 2016.

[10] M. H. Foroozannejad, M. Hashemi, T. L. Hodges, and S. Ghiasi. Look into details: The benefits of fine-grain streaming buffer analysis. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '10)*, pages 27–36, 2010.

[11] M. Gao and C. Kozyrakis. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA '16)*, pages 126–137, March 2016.

[12] M. Gao, G. Ayers, and C. Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT '15)*, pages 113–124, Oct 2015.

[13] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA '16)*, pages 204–216, June 2016.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[15] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA '16)*, pages 380–392, June 2016.

[16] Y. LeCun, K. Kavukcuoglu, and C. Farabet. Convolutional networks and applications in vision. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS '10)*, pages 253–256, May 2010.

[17] Lenovo Group Ltd.'s eXFlash. http://www.lenovo.com/images/products/system-x/pdfs/datasheets/exflash_memory_channel_storage.pdf. 2017.

[18] P.-J. Micolet, A. Smith, and C. Dubach. A machine learning approach to mapping streaming workloads to dynamic multicore processors. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES '16)*, pages 113–122, 2016.

[19] N. L. Passos and E. H. M. Sha. Synchronous circuit optimization via multidimensional retiming. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 43(7):507–519, Jul 1996.

[20] J. T. Pawlowski. Hybrid memory cube (HMC). In *2011 IEEE Hot Chips 23 Symposium (HCS '11)*, pages 1–24, Aug 2011.

[21] A. Rahman, J. Lee, and K. Choi. Efficient FPGA acceleration of convolutional neural networks using logical-3D compute array. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE '16)*, pages 1393–1398, March 2016.

[22] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li. C-brain: A deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC '16)*, pages 123:1–123:6, June 2016.

[23] Viking Technology's ArxCis-NV. http://www.vikingtechnology.com/uploads/arxcis_productbrief.pdf. 2017.

[24] Y. Wang, Z. Shao, H. C. B. Chan, D. Liu, and Y. Guan. Memory-aware task scheduling with communication overhead minimization for streaming applications on bus-based multiprocessor system-on-chips. *IEEE Transactions on Parallel and Distributed Systems*, 25(7):1797–1807, July 2014.

[25] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li. DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC '16)*, pages 110:1–110:6, June 2016.

[26] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15)*, pages 161–170, 2015.