SCIENCE CHINA
Information Sciences

June 2021, Vol. 64 160409:1–160409:14
https://doi.org/10.1007/s11432-020-3248-y

• RESEARCH PAPER •
Special Focus on Near-memory and In-memory Computing

# Towards efficient allocation of graph convolutional networks on hybrid computation-in-memory architecture

Jiaxian CHEN, Guanquan LIN, Jiexin CHEN & Yi WANG*

*College of Computer Science and Software Engineering, Shenzhen University, Shenzhen* 518060*, China*

Received 31 December 2020/Revised 10 March 2021/Accepted 15 April 2021/Published online 10 May 2021

**Abstract** Graph convolutional networks (GCNs) have been applied successfully in social networks and recommendation systems to analyze graph data. Unlike conventional neural networks, GCNs introduce an aggregation phase, which is both computation- and memory-intensive. This phase aggregates features from the neighboring vertices in the graph, which incurs significant amounts of irregular data and memory access. The emerging computation-in-memory (CIM) architecture presents a promising solution to alleviate the problem of irregular accesses and provide fast near-data processing for GCN applications by integrating both three-dimensional stacked CIM and general-purpose processing units in the system. This paper presents Graph-CIM, which exploits the hybrid CIM architecture to determine the allocation of GCN applications. Graph-CIM models the GCN application process as a directed acyclic graph (DAG) and allocates tasks on the hybrid CIM architecture. It achieves fine-grained graph partitioning to capture the irregular characteristics of the aggregation phase of GCN applications. We use a set of representative GCN models and standard graph datasets to evaluate the effectiveness of Graph-CIM. The experimental results show that Graph-CIM can significantly reduce the processing latency and data-movement overhead compared with the representative schemes.

**Keywords** computation-in-memory, graph convolutional networks, hybrid architecture, scheduling, inference, accelerator

**Citation** Chen J X, Lin G Q, Chen J X, et al. Towards efficient allocation of graph convolutional networks on hybrid computation-in-memory architecture. Sci China Inf Sci, 2021, 64(6): 160409, https://doi.org/10.1007/s11432-020-3248-y

## 1 Introduction

Graph neural networks (GNNs) are specifically designed for large-scale graph processing and are widely used in social networks and recommendation systems. Among the various types of GNNs, the graph convolutional network (GCN) is the most commonly used and mainly consists of two types of operations: aggregation and combination. These operations dominate the processing latency of a GCN. The aggregation operation is similar to graph processing, where each vertex must be processed to aggregate the features of all its neighboring vertices. The combination operation typically uses a multilayer perceptron (MLP), which is similar to a conventional neural network.

To efficiently accelerate the GCN processing, the system architecture should exploit both the irregular pattern in the aggregation operation and the regular pattern in the combination operation. Owing to different characteristics in data access and memory access, the aggregation and combination operations impose different requirements on the system architecture. It is difficult to find a specific accelerator that can fit both operations.

One promising solution to this problem is to adopt a hybrid architecture that integrates two different kinds of hardware architectures to handle these two operations. The three-dimensional (3D)-stacked computation-in-memory (CIM) architecture provides a viable solution to reduce the data-movement overhead. In this architecture, the computation resources are placed near or inside the memory resources to

* Corresponding author (email: yiwang@szu.edu.cn)

 

effectively handle the irregular memory access in the aggregation operation. Thus, we argue that the CIM architecture can partially solve the problem. The regular memory access in the combination operation cannot be handled effectively by the CIM architecture. Therefore, we adopt both the CIM architecture and general-purpose processing units, such as a central processing unit (CPU) or graphics processing unit (GPU), in the design.

This paper presents Graph-CIM, an efficient task-allocation strategy to speed up the inference of graph convolutional networks on hybrid computation-in-memory architectures. Graph-CIM captures the data- and memory-access requirements during GCN processing, and utilizes the properties of both the general-purpose processing units and 3D-stacked CIM architecture in order to reduce the processing latency of GCNs and reduce the migration overhead. Graph-CIM adopts a two-phase framework to determine the allocation of tasks on its hybrid architecture. In the first phase, Graph-CIM models the process of a GCN application as a directed acyclic graph (DAG) and further partitions the DAG into a fine-grained DAG to improve the utilization of the hybrid architecture. For the aggregation operation, Graph-CIM abstracts the hierarchical community structure in the input graph, and partitions the graph. In the second phase, Graph-CIM allocates tasks to the DAG. The task allocation is jointly determined by both the actual utilization status of the processing units and the properties of the aggregation and combination operations. Graph-CIM aims to achieve load balancing across different processing units and reduce the processing latency of the GCN application.

We evaluate the proposed technique with a set of representative GCN models and standard graph datasets. The abstractions of GCN models are obtained from the deep learning framework PyTorch. We compare Graph-CIM with representative schemes in terms of processing latency and computing resource utilization. Experimental results show that Graph-CIM can achieve a significant reduction in execution time and effectively utilize the hybrid CIM architecture.

The main contributions of this paper are as follows.

- The proposed scheme takes advantage of the hybrid CIM architecture to capture the irregular characteristics of different GCN applications.
- The proposed scheme models the processing of a GCN application as a DAG and efficiently allocates the workloads on the hybrid CIM architecture.
- As a proof of concept, we compare the proposed scheme with representative schemes using a set of real GCN workloads and standard graph datasets.

The rest of this paper is organized as follows. Section 2 provides an overview of the background and discusses the motivations of this paper. Section 3 presents our proposed scheme, Graph-CIM, in detail. Section 4 presents our experimental results. Section 5 discusses the related work, and Section 6 concludes this paper and discusses future work.

## 2 Background and motivation

### 2.1 Graph convolutional networks

GNNs are advanced machine learning techniques to learn the representations of graph properties from non-Euclidean input graph structures [1]. Among the various kinds of GNNs, GCNs are among the most popular. GCNs exploit hierarchical patterns in a graph via shared weights and the multilayer structure. GCNs mainly iterate via convolutional layers, which dominate the computation time, and can be categorized as aggregation phase and combination phase.

Figure 1 illustrates the typical processing procedure for GCNs. The process starts with the input graph, which consists of the graph structure, vertex features, and other graph properties. The input graph $G(V, E)$ can be defined by a set of vertices $V = \{V_1, V_2, \ldots, V_n\}$ and a set of edges $E \subseteq V \times V$. The features of a vertex $v \in V$ can be represented by a feature vector $h_v^0$. Each vertex $v \in V$ has a neighborhood set $N(v)$ determined by the vertices connecting to it. In this example, the input graph has 7 vertices, each of which has a 4-dimensional feature vector.

In the aggregation phase, each vertex $v \in V$ gathers its neighboring vertices' feature vectors and generates an aggregated feature vector. The aggregated feature vector of $v$ can be expressed as $a_v^k$, where $k$ denotes the number of GCN iterations. The aggregation phase is dominated by aggregators (or aggregation functions). Different GCNs have different aggregators. For example, for each $v \in V$, the mean aggregator in Figure 1 takes the elementwise mean of the feature vectors $N(v)$, and generates a

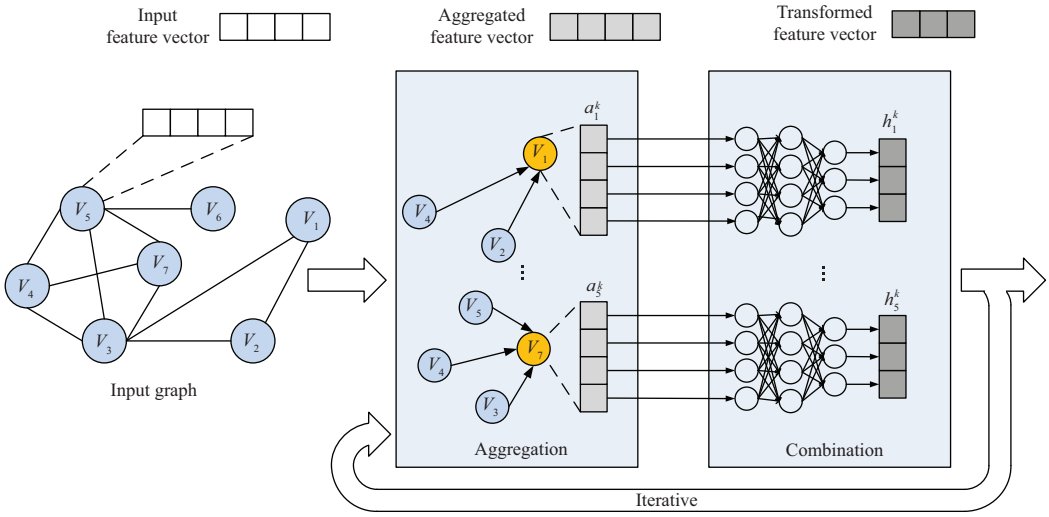


**Figure 1** (Color online) The typical processing procedure for GCNs.

4-dimensional aggregated feature vector.

The combination phase transforms the aggregated feature vector of vertex $v \in V$ to a transformed feature vector based on its present state via combine functions. Similarly, during the $k$-th iteration, the transformed feature vector of $v$ can be expressed as $h_v^k$. Most of the time, a multilayer perception model is used as the combine function, which can be treated as a special kind of matrix multiplication. In Figure 1, the outputs of the aggregation phase can be flattened and concatenated to produce a matrix of size 7×4. This matrix is further multiplied with several weight matrices to generate an output matrix of size 7×3. The output matrix represents 7 transformed feature vectors with a length of 3.

Aggregation alternates with combination for multiple iterations to generate the final output feature vectors. After $k$ iterations of the graph convolutional layer, the output feature vectors extract the structural information within their $k$-hop neighboring vertices.

### 2.2 Heterogeneous CIM architecture

The CIM architecture is a promising solution to reduce the memory bandwidth and minimize the data transfer overhead between processing units and off-chip dynamic random access memory (DRAM). We adopt a hybrid CIM system to address the imbalance computation requirement of GCNs. Figure 2 illustrates a typical heterogeneous CIM architecture, where a 3D-stacked memory chip is placed adjacent to a general-purpose CPU chip. Both chips are linked via a memory link on a silicon interposer. The emerging CIM architecture integrates multiple embedded DRAM (eDRAM) dies and one logic die in a 3D-IC architecture. Each DRAM die is partitioned into 16 sub-dies. The sub-dies in the vertical direction form a vault, and each vault is individually controlled by a vault controller in the logic die. The logic die integrates vault controller (VC), programmable neurosequence generator (PNG), router, and processing engine (PE). Multiple routers are interconnected through a two-dimensional mesh or a fully connected network-on-chip (NoC). The data is further transmitted to PEs through the NoC network. Data could be stored either in each PE's cache or in the vault (i.e., DRAM). The PNG determines this selection and stores the data in the cache for the corresponding PE.

### 2.3 System model

Based on the processing dataflow, a GCN application can be modeled as a DAG $D(T, R)$, where $T = \{T_1, T_2, \ldots, T_m\}$ denotes a set of $m$ tasks. Each task $T_i$ represents a specific operation, such as the aggregation or combination phase, and is associated with three properties $(\mathrm{op}_i, s_i, p_i)$, where $\mathrm{op}_i$ represents the operation type (e.g., aggregation, combination), $s_i$ represents the scale of the input/output data, and $p_i$ represents the predefined parameters or data (e.g., the topology of the input graph for the aggregation phase). $R \in T \times T$ is the edge set of $D$, which denotes the data dependence between different tasks.

In this paper, the worst-case execution time of each operation is estimated based on the requirements of the computing resources and DRAM access [2]. The estimated execution time of each task $T_i$ is bounded

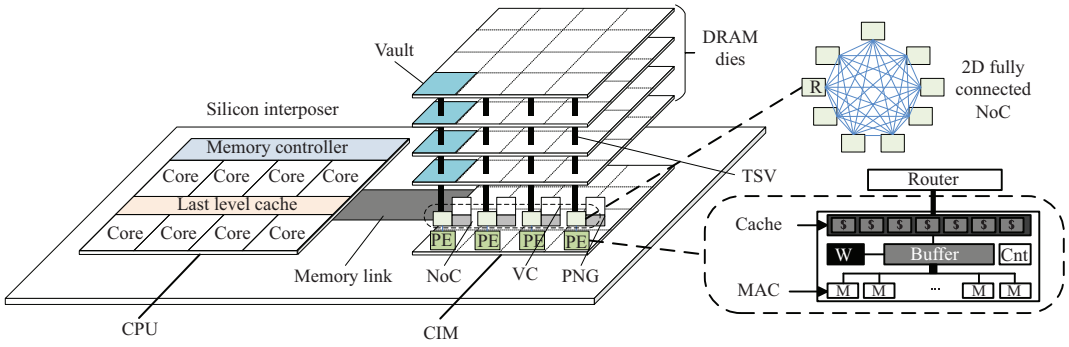


**Figure 2** (Color online) The system architecture of Graph-CIM, which consists of both general-purpose CPU and CIM architecture.

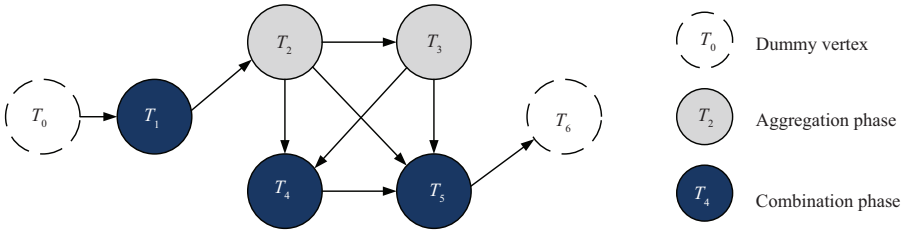


**Figure 3** (Color online) A typical graph convolutional network DAGNN is modeled as a DAG $D$.

by $t_i = t_{\text{mem}}(\text{op}_i, s_i, p_i, d, e) + t_{\text{comp}}(\text{op}_i, s_i, p_i, d, e)$, where $t_{\text{mem}}$ and $t_{\text{comp}}$ denote the parameters used to predict the memory-access and processing latencies, respectively. $d$ denotes the parameters of the computing resources, such as the number of cores in the general-purpose CPU, the number of PEs in the CIM, and cache capacity. $e$ denotes the execution pattern, such as using vertex-centric, edge-centric [3], or other types of execution [4] in the aggregation phase. Figure 3 illustrates the model of a typical GCN, called a deep adaptive graph neural network (DAGNN) [5]. We use a dummy vertex ($T_0$ in Figure 3) to denote the inputs of the GCN. Similarly, the outputs of the GCN can also be represented as a dummy vertex ($T_6$) in the graph. As a result, the DAGNN can be modeled as a DAG with 7 vertices and 9 edges. Note that the vertex in the DAG could be further partitioned into several vertices to exploit the fine-grained allocation of the hybrid architecture.

## 2.4 Motivation

For GCN applications, the memory-access pattern of the aggregation phase is irregular. In the aggregation phase, each vertex must be processed by aggregating features from all its neighboring vertices in the graph [1,5–7]. Different vertices in a GCN application's input graph have different numbers of neighboring vertices and different topology relations. The memory-access pattern for the processing of one vertex is significantly different from that of another vertex.

In order to exploit the locality in memory accesses and speed up aggregation-phase processing, it is necessary to find the vertices that share similar topology relations. Therefore, Graph-CIM presents a graph-partitioning algorithm that adopts community detection to find vertices with similar topology relations and allocate them to the cache in the hybrid architecture. This can exploit the locality of the input graph and increase the processing speed of the aggregation phase.

The combination phase is normally implemented by a variation of the conventional MLP, which can be integrated as regular matrix multiplication operations [1, 5–7]. Unlike the conventional MLP, such as the fully-connected layer in a CNN, the weight matrix in the combination phase can be reused by the feature vectors of each vertex in the input graph. Owing to this reusability, the combination phase is bounded by computation instead of memory, and data movement is no longer the major bottleneck.

We conduct a deep analysis of the GCN workload and explore its design on the hybrid architecture. We test the processing latency of a typical benchmark (i.e., Citeseer) with four GCN networks on both CPU-based and 3D-stacked CIM architectures. Four GCN models, GCN [1], graph wavelet neural network (GWNN) [6], DAGNN [5], and simplifying graph convolutional network (SGC) [7], are used in the experiments. The standard dataset Citeseer is adopted as the inference test case. The detailed experimental

**Table 1** The processing latency of the aggregation phase with four GCN networks on both CPU and 3D-stacked CIM architecture

| Dataset | CPU | CIM | Speed increase (CIM vs. CPU) |
|---|---|---|---|
| DAGNN | 1.0 | 0.05 | 20.35× |
| GCN | 1.0 | 0.11 | 8.93× |
| GWNN | 1.0 | 0.06 | 17.42× |
| SGC | 1.0 | 0.09 | 11.10× |
| Average | – | – | 14.45× |

**Table 2** The processing latency of the combination phase with four GCN networks on both CPU and 3D-stacked CIM architecture

| Dataset | CPU | CIM | Speed increase (CPU vs. CIM) |
|---|---|---|---|
| DAGNN | 1.0 | 2.55 | 2.55× |
| GCN | 1.0 | 2.45 | 2.45× |
| GWNN | 1.0 | 2.52 | 2.52× |
| SGC | 1.0 | 2.60 | 2.60× |
| Average | – | – | 2.53× |

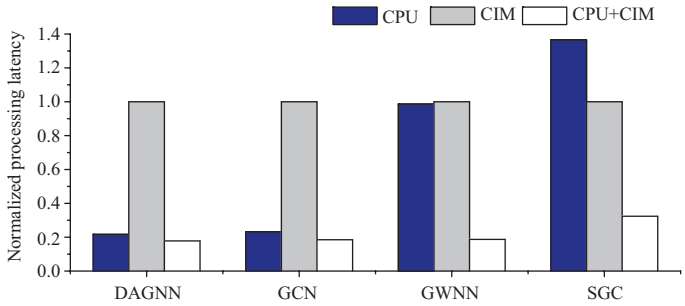


**Figure 4** (Color online) A motivational example for running dataset Citeseer on CPU, 3D-stacked CIM, and the hybrid architecture.

setup is presented in Section 4.

The experimental results are illustrated in Tables 1 and 2. We measure the processing latency of the aggregation phase and the combination phase and normalize it to the results of running the Citeseer dataset on a CPU. As shown in Table 1, the 3D-stacked CIM architecture can significantly accelerate the aggregation phase and achieve an average 14.45× speed increase compared to that on the CPU. Due to the irregular topology of the input graph, the aggregation phase may incur frequent cache misses in the CPU, which leads to additional processing latency. The 3D-stacked CIM architecture exploits the near-data processing architecture and can achieve low data latency by reducing data-movement overhead. As shown in Table 2, the general-purpose CPU is more suitable for combination-phase processing and can achieve an average speed increase of 2.53× compared to that on the 3D-stacked CIM architecture. Modern general-purpose processing units, such as CPUs and GPUs, utilize multilevel caches and single-instruction-multiple-data (SIMD) technique to optimize the matrix multiplication process. In contrast to general-purpose processing units, the 3D-stacked CIM architecture has a limited cache space and limited number of processing units owing to the limitations of 3D-stacked technology [8]. Because the combination phase is bounded by computation, the 3D-stacked CIM architecture may not handle the combination phase efficiently.

We further conducted a set of experiments to demonstrate the necessity of the hybrid architecture. Figure 4 presents the results for running dataset Citeseer on a CPU, 3D-stacked CIM, and hybrid architecture of both. The results are also normalized by the results of the Citeseer dataset on CIM. From the results, the processing latency of some GCN models (i.e., DAGNN, GCN, and GWNN) on a general-purpose CPU is less than that on the 3D-stacked CIM architecture. Other GCN models (i.e., SGC) show a different trend, where the 3D-stacked CIM architecture outperforms the general-purpose CPU. We also observe that the hybrid architecture with both a general-purpose CPU and 3D-stacked CIM architecture can achieve the lowest processing latency among the three architectures. The processing latency of the hybrid architecture is less than half of that for both the general-purpose CPU and 3D-stacked CIM.

Based on our preliminary results, we argue that it is necessary to (1) allocate the aggregation phase on the 3D-stacked CIM architecture to handle the irregular data and memory accesses, (2) allocate the

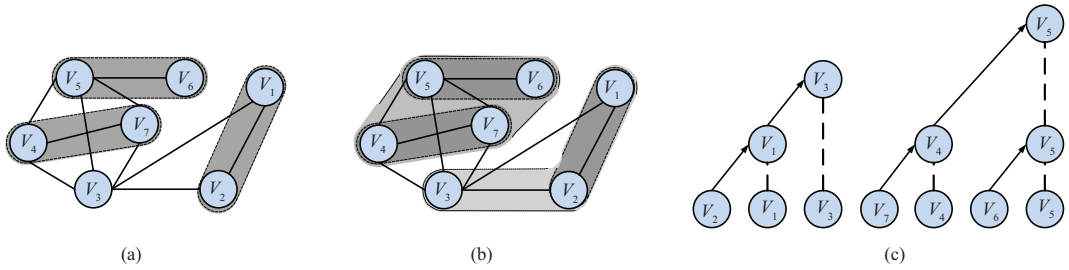


**Figure 5** (Color online) Community detection for the input graph to abstract the hierarchical community structures. (a) and (b) The procedure of modularity-based community detection; (c) the hierarchical community structures.

combination phase on the general-purpose processing units to capture the regular processing pattern, and (3) perform extra data migration to fully utilize the computing resources of both the general-purpose processing units and the 3D-stacked CIM architecture, because of the mismatched resource consumer (i.e., GCN applications) and resource provider (i.e., the hybrid architecture). A task allocation or scheduling scheme should be proposed to allocate the processes of GCNs based on the process type and to balance the workloads across the two different computing resources. These observations motivate the proposition of an efficient task-allocation strategy to speed up the inference of GCNs on the hybrid CIM architecture.

## 3 Graph-CIM: a task allocation strategy for GCNs on hybrid architecture

### 3.1 Overview

Graph-CIM aims to optimize the inference of GCNs on the hybrid CIM architecture. Graph-CIM presents a task-allocation strategy that consists of two major steps. In the first step, Graph-CIM divides the aggregation task into a set of subtasks based on the topological properties of the input graph. To speed up processing during the aggregation phase, Graph-CIM analyzes the requirements for the computing and memory resources and performs graph partitioning of the input graph. This can exploit parallelism and alleviate the irregular memory accesses in the aggregation phase. In the second step, Graph-CIM conducts task scheduling on the hybrid architecture to take full advantage of both the general-purpose CPU and 3D-stacked CIM architecture. Graph-CIM adaptively adjusts the scheduling based on the utilization status of the hybrid processing units and migrates heavy workloads to reduce the processing latency. This section first presents the proposed graph-partitioning strategy in Subsection 3.2. Then, we present the proposed fine-grained scheduling algorithm on the hybrid architecture in Subsection 3.3.

### 3.2 Graph partitioning

To exploit the parallelism of the multiprocessor device in the hybrid architecture, Graph-CIM performs graph partitioning. The objective is to achieve fine-grained task scheduling and guide the task mapping to match the workload with the hybrid architecture. The input graph of a GCN application is normally a power-law graph, and its vertices do not have the same number of edges. This property affects the effectiveness of graph partitioning by destroying the potential data locality of the input graph. Therefore, Graph-CIM adopts community detection to guide the graph-partitioning process.

Graph-CIM performs community detection, which preprocesses an input graph and divides it into multiple vertex sets, to abstract the hierarchical community structure, where a community denotes a vertex set. The vertices within a set have strong connections, whereas those among different sets have weak connections. We adopt modularity-based algorithms [9], which use modularity as the metric to detect vertices with strong connections [9], to perform community detection. For the first step, each vertex of the input graph is initialized as the root vertex of its own independent community. Second, a community $c_i$ selects a neighboring community that has the maximum gain of modularity, and this community $c_i$ is connected to this neighboring community. The root vertex of this neighboring community becomes the new root vertex of community $c_i$. This step is repeated multiple times to abstract the hierarchical community structure within an input graph.

Figure 5 illustrates an example. In Figure 5(a), vertices $V_1$ and $V_2$ can create a vertex set based on their similarity. In the same manner, vertices $V_4$ and $V_7$, and vertices $V_5$ and $V_6$, create two separate vertex sets. In Figure 5(b), vertex $V_3$ is further merged into the set with vertices $V_1$ and $V_2$. Two vertex

sets $\{V_4, V_7\}$ and $\{V_5, V_6\}$ create a new vertex set. The input graph can form a hierarchical community structure, as shown in Figure 5(c).

Due to the mismatch between the scale of the vertex sets and the cache capacity, the vertex sets must be further transformed into subtasks in order to fully utilize the valuable cache space. This requires large vertex sets that cannot be entirely loaded into the cache to be partitioned and several small vertex sets to be merged to fill the spare cache space.

For the $k$-th aggregation operation, a vertex with feature vector requires $f_k$ memory space. Let $C_p$ denote the cache capacity. Then the cache can maintain up to $\mathrm{mv}_k = \lfloor \frac{C_p}{f_k} \rfloor$ vertices. For the hybrid architecture that contains $N_p$ processing units, the $k$-th aggregation operation is partitioned into $\mathrm{nt}_k$ subtasks according to

$$\mathrm{nt}_k = \max\left(\left\lceil \frac{n \cdot f_k}{C_p} \right\rceil, N_p\right). \tag{1}$$

Algorithm 1 presents the procedure to generate a set of aggregation subtasks, which consists of two parts: graph partitioning and merging of small vertex sets. Algorithm 1 first initializes $\mathrm{nt}_k$ empty subtasks to store multiple vertices (line 1). There are two types of vertex sets, $C_{\mathrm{big}}$ and $C_{\mathrm{small}}$, which are defined as follows.

**Algorithm 1** Generate aggregation subtasks $(C, \mathrm{nt}_k, \mathrm{mv}_k)$

**Require:** A list of vertex sets after community detection $C$, the number of subtasks $\mathrm{nt}_k$, and the maximum number of vertices that the cache can maintain $\mathrm{mv}_k$.
**Ensure:** A set of aggregation subtasks St.
1: Initialize St with $\mathrm{nt}_k$ empty subtasks;
2: $C_{\mathrm{big}} \leftarrow \{u | u \in C$ and $u.\mathrm{size} > \mathrm{mv}_k\}$;
3: $C_{\mathrm{small}} \leftarrow \{u | u \in C$ and $u.\mathrm{size} \leqslant \mathrm{mv}_k\}$;
4: **while** $C_{\mathrm{big}} \neq \emptyset$ **do**
5: $u \leftarrow$ DEQUEUE($C_{\mathrm{big}}$);
6: $v \leftarrow$ DEQUEUE($u$.children);
7: $u.\mathrm{size} \leftarrow u.\mathrm{size} - v.\mathrm{size}$;
8: $C_{\mathrm{tmp}} = \{u, v\}$;
9: **for** $w \in C_{\mathrm{tmp}}$ **do**
10: **if** $w.\mathrm{size} \leqslant \mathrm{mv}_k$ **then**
11: ENQUEUE($C_{\mathrm{small}}$, $w$);
12: **else**
13: ENQUEUE($C_{\mathrm{big}}$, $w$);
14: **end if**
15: **end for**
16: **end while**
17: Sort $C_{\mathrm{small}}$ by the size of a vertex set;
18: **for** $u \in C_{\mathrm{small}}$ **do**
19: subtask $\leftarrow$ the subtask in St with the minimum number of vertices;
20: ENQUEUE(subtsk, $u$);
21: **end for**
22: **return** St.

**Definition 1.** $C_{\mathrm{big}}$ is a type of vertex sets. The number of vertices in each vertex set in $C_{\mathrm{big}}$ is greater than $\mathrm{mv}_k$.

**Definition 2.** $C_{\mathrm{small}}$ is a type of vertex sets. The number of vertices in each vertex set in $C_{\mathrm{small}}$ is less than or equal to $\mathrm{mv}_k$.

Graph partitioning is intended to divide vertex sets in $C_{\mathrm{big}}$ into several vertex sets. The partitioned vertex sets then become $C_{\mathrm{small}}$ and can fit the cache capacity (lines 4–16). The merging of small vertex sets is intended to form a set of subtasks, each of which fills the cache capacity (lines 17–21). Different vertex sets (i.e., all sets of $C_{\mathrm{small}}$) are sorted in descending order according to their sizes (line 17). Then, the vertex sets are assigned successively to the subtask with the minimum number of vertices (lines 18–21).

A GCN application consists of multiple aggregation operations. Algorithm 1 presents the partitioning process for one aggregation operation. To reduce the number of transfers from $C_{\mathrm{big}}$ to $C_{\mathrm{small}}$, different aggregation operations are sorted in ascending order according to the dimension of the feature vector. The aggregation operation with a lower-dimension feature vector is selected to conduct partitioning in Algorithm 1. The partitioning is based on the results of previous partitions.

Each set in $C_{\mathrm{big}}$ is partitioned into two sets, which takes $O(1)$ time. $C_{\mathrm{big}}$ consists of at most $n$ sets. Therefore, graph partitioning takes $O(n)$ time (lines 4–16). The merging process first sorts the vertex

size in descending order, which takes $O(n)$ time (line 17). Then, merging the small vertex sets takes $O(n \cdot \log(n))$ time (lines 18–21). Therefore, the time complexity of Algorithm 1 is $O(n \cdot \log(n))$.

### 3.3 Task scheduling on the hybrid architecture

Graph-CIM performs task scheduling to take full advantage of the hybrid architecture. Algorithm 2 presents the detailed steps to generate a feasible task schedule. This algorithm divides the tasks into several groups. Data dependence exists among the different groups, whereas tasks within a group do not have data dependence. For each group of tasks, Graph-CIM performs task mapping and migration. By scheduling tasks within each group, these independent tasks can be allocated to processing units concurrently, which can reduce processing latency. In Algorithm 2, breadth first search (BFS) is adopted to divide the task set into several groups (line 1).

$$\mathrm{EST}_i = \max(\mathrm{FT}_l), \quad T_l \in \mathrm{pred}(T_i). \tag{2}$$

**Algorithm 2** Schedule ($T$, usage[])

**Require:** A set of tasks $T = \{T_1, T_2, \ldots, T_m\}$, the using status of Cores and PEs usage[].
**Ensure:** A task schedule on the hybrid architecture.

```
1:  Get groups GP by performing BFS with task T;
2:  for group gp ∈ GP do
3:      Initialize Array_cpu, Array_cim;
4:      for task T_i ∈ gp do
5:          if op_i is COMBINATION then
6:              Add T_i to Array_cpu;
7:          else
8:              Add T_i to Array_cim;
9:          end if
10:     end for
11:     Obtain EST_i from (2) for each T_i ∈ gp.
12:     Sort tasks in Array_cpu and Array_cim respectively in ascending order of EST;
13:     Allocate tasks in Array_cpu successively to the idlest CPU core;
14:     Allocate tasks in Array_cim successively to the idlest CIM PE;
15:     end_move ← False;
16:     Update A_busy and A_idle;
17:     while end_move is False do
18:         Obtain the busiest core P_busy in A_busy;
19:         Obtain the idlest core P_idle in A_idle;
20:         Obtain the final task T_tail of core P_busy;
21:         Obtain the execution time t_tail of task T_tail in A_idle;
22:         if usage[P_idle] + t_tail < usage[P_busy] then
23:             Migrate T_tail from core P_busy to P_idle;
24:         else
25:             end_move ← True;
26:         end if
27:     end while
28: end for
```

The tasks are allocated based on their preference for different architectures (lines 3–14). $\mathrm{Array}_{\mathrm{cpu}}$ and $\mathrm{Array}_{\mathrm{cim}}$ are employed to classify tasks with a preference for CPU and CIM architecture, respectively (lines 3–10). An aggregation task is preferentially allocated to the CIM architecture, whereas a combination task is allocated to the general-purpose CPU. Graph-CIM further maps tasks to processing units according to the earliest start time (EST) (lines 11–14). Eq. (2) is used to calculate EST of $T_i$ (line 11), where $\mathrm{EST}_i$ denotes the EST of task $T_i$. $T_i$ has several predecessor tasks, and $\mathrm{FT}_l$ denotes the finishing time of task $T_l$. Then $\mathrm{EST}_i$ is equal to the maximum of $\mathrm{FT}_l, l \in \mathrm{pred}(T_i)$. The tasks in both $\mathrm{Array}_{\mathrm{cpu}}$ and $\mathrm{Array}_{\mathrm{cim}}$ are sorted in ascending order of EST (line 12), and then the tasks in $\mathrm{Array}_{\mathrm{cpu}}$ and $\mathrm{Array}_{\mathrm{cim}}$ are mapped successively to the idlest processing unit of the CPU and CIM, respectively (lines 13 and 14). This ensures that the task with the EST can be processed first.

For task migration, Graph-CIM evaluates the utilization status of different architectures and their processing units to reallocate tasks (lines 15–27). Graph-CIM sets one architecture as the busy architecture $A_{\mathrm{busy}}$ and the other architecture as the idle architecture $A_{\mathrm{idle}}$ (lines 15 and 16). Then, Graph-CIM migrates the tasks in the busiest processing unit in $A_{\mathrm{busy}}$ to the idlest processing unit in $A_{\mathrm{idle}}$ (lines 17–27). If the utilization status is the same, the processing unit with the lowest logical unit number is assigned as the busiest processing unit, and processing unit with the greatest logical unit number is assigned as

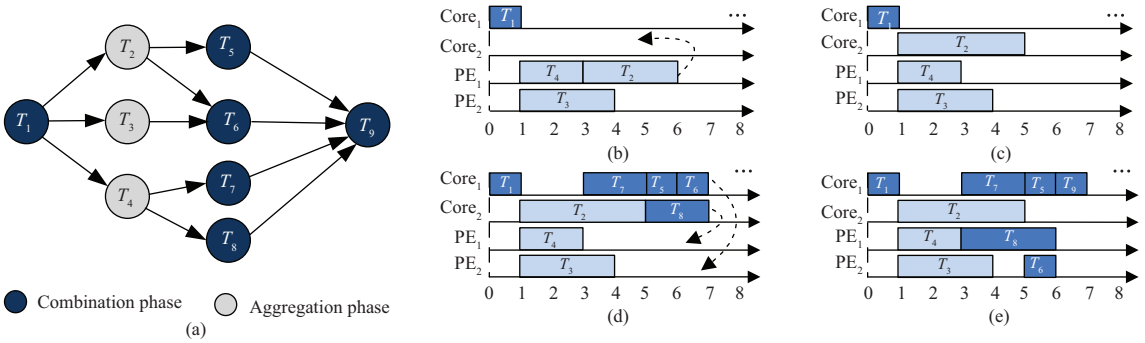


**Figure 6** (Color online) Task mapping and task migration of a GCN model. (a) The DAG used to represent a GCN; (b)–(e) the procedures of task mapping and task migration.

the idlest processing unit. Graph-CIM balances the allocation of tasks within each group. As the tasks within a group are independent, the load-balancing process exploits the search space of task scheduling and effectively reduces the processing latency of the workload.

We use an example to illustrate the basic procedures of task mapping and task migration. As shown in Figure 6, the task set is partitioned into four groups $\{T_1\}$, $\{T_2, T_3, T_4\}$, $\{T_5, T_6, T_7, T_8\}$, $\{T_9\}$. For the first group $T_1$, its EST is equal to 0. As $T_1$ is a combination operation, $T_1$ is allocated to $\text{Core}_1$ in CPU. For the group $\{T_2, T_3, T_4\}$, the EST of each task is equal to 1, and the tasks are allocated to the CIM architecture based on their types. After these tasks are sorted, task $T_4$ is allocated to the idlest processing unit $\text{PE}_1$. Then, tasks $T_3$ and $T_2$ are mapped to $\text{PE}_2$ and $\text{PE}_1$, respectively. Task migration is triggered by the imbalanced allocation of tasks, and task $T_2$ is migrated from $\text{PE}_1$ to $\text{Core}_2$. As $T_2$ is coarse-grained, the frequent memory access leads to longer execution time on the CPU.

The ESTs of the tasks in group $\{T_5, T_6, T_7, T_8\}$ are 5, 5, 3, and 3, respectively. These tasks are combination operations and mapped to the CPU. After sorting the ESTs, task $T_7$ is first allocated to the idlest core $\text{Core}_1$. Tasks $T_8$, $T_5$, and $T_6$ are further mapped to $\text{Core}_2$, $\text{Core}_1$, and $\text{Core}_1$, respectively, and tasks $T_8$ and $T_6$ are migrated to $\text{PE}_1$ and $\text{PE}_2$, respectively. The last task $T_9$ is a combination task, and is mapped to $\text{Core}_1$ based on its data dependence and operation type. Figure 6(e) shows the final task schedule for the GCN model.

## 4 Evaluation

To demonstrate the viability of the proposed scheme, we have conducted a set of experiments and compared our scheme with two representative schemes. In this section, we first introduce the experimental setup, and then present the experimental results with detailed analysis.

### 4.1 Experimental setup

The widely used open-source deep learning framework PyTorch [10] is used to train neural network models and abstract execution model representations. DAGs are abstracted from the log files of PyTorch and TensorBoardX to determine the functionality of the deep learning applications.

The hybrid CIM architecture consists of both CPU and 3D-stacked CIM architectures. The 3D-stacked CIM is implemented based on the Neurocube model [8], which is an extension of Micron's hybrid memory cube (HMC) [11], to support neural computing. The 3D-stacked CIM is configured to contain sixteen 300-MHz PEs, each with 16 MACs. The 3D-stacked CIM integrates a cache with a capacity of 1 MB, and DRAM with an 8 GB capacity and 160 GB/s bandwidth (HMC 2.0). The general-purpose CPU integrates an 8-core 3.0 GHz Intel Xeon CPU. The capacities of the cache and DRAM are 12 MB and 8 GB, respectively. The DRAM bandwidth is 41.6 GB/s with a DDR4 interface.

We compared Graph-CIM with two baseline architectures and different scheduling algorithms on the hybrid CIM architecture. A state-of-the-art framework, PyTorch Geometric (PyG) [12], is adopted as the baseline scheme. We compared the proposed scheduling algorithm with a well-known heterogeneous scheduling algorithm, HEFT [13], and a representative hybrid CIM scheduling algorithm, PEFT [14], on top of the hybrid architecture.

**Table 3** GCN models and standard graph datasets

| Dataset | Number of vertices | Number of edges | Number of features | Number of classes |
|---|---|---|---|---|
| Blogcatalog | 5196 | 171743 | 8189 | 7 |
| Citeseer | 3312 | 4715 | 3703 | 6 |
| Coauthor | 18333 | 81894 | 6805 | 15 |
| DBLP | 17716 | 52867 | 1639 | 4 |
| Pubmed | 19717 | 44324 | 500 | 3 |

To evaluate the effectiveness of the proposed Graph-CIM, we use a set of representative GCN models and standard graph datasets, as shown in Table 3. A GCN [1], GWNN [6], DAGNN [5], and SGC [7] are used in the experiments. These GCN applications are typical semi-supervised classification applications and are adopted as GCN models in the experiments. The datasets listed in Table 3 are used as the inputs of the GCN model. These datasets are obtained from citation networks and social networks[1)], and their characteristics are described in Table 3. Particularly, the columns "Number of features" and "Number of classes" represent the dimensions of the feature vector and the number of labeled classes, respectively.

## 4.2 Results and discussion

### 4.2.1 *Processing latency*

We first compare the processing latency of GCNs on four different types of architectures: (1) PyG on the general-purpose CPU, (2) Graph-CIM on the general-purpose CPU, (3) Graph-CIM on the 3D-stacked CIM, and (4) Graph-CIM on the hybrid architecture with both the general-purpose CPU and the 3D-stacked CIM. Figure 7 presents the processing latency of GCNs on four different types of architectures. From the experimental results, PyG on CPU (PyG-CPU) has the longest processing latency. In our technique, we adopt the graph-partitioning strategy for the input graph based on the graph community structure. This improves the irregular data access in the aggregation phase. We observe that the processing latency of some GCN models for Graph-CIM on CPU is less than that for Graph-CIM on CIM. This is because different GCN models have different ratios of the aggregation phase to the combination phase. As the general-purpose CPU is more suitable for processing the combination phase and the 3D-stacked CIM is more effective to reduce the latency of the aggregation phase, different combinations of these two phases affect the processing latency.

We also observe that the proposed Graph-CIM can obtain the lowest processing latency. Graph-CIM considers the properties of memory and data access and takes advantage of the hybrid architecture to accelerate the irregular data access of the aggregation phase and exploit the parallelism of the combination phase. From the experimental results, Graph-CIM can reduce the processing latency by 82.38%, 42.38%, and 44.80% on average compared to PyG-CPU, Graph-CIM on CPU, and Graph-CIM on CIM, respectively.

We further conduct a set of experiments to compare the processing latency of several GCN models with different scheduling algorithms. We compare the proposed Graph-CIM with two representative scheduling algorithms, HEFT [13] and PEFT [14], on the hybrid architecture. For all three scheduling algorithms, the input DAG is abstracted from the log files of PyTorch and TensorboardX. Figure 8 illustrates the experimental results, from which it can be observed that Graph-CIM obtains the lowest processing latency among the three scheduling algorithms. HEFT is an efficient heterogeneous scheduling algorithm. However, its scheduling granularity is still coarse, which leads to inefficient utilization of the hybrid architecture due to the data dependence of the GCN. PEFT is a scheduling algorithm intended for the heterogeneous CIM architecture. Although it adopts the fine-grained scheduling to partition tasks and exploits the ability to process tasks in parallel, it eliminates the data locality inside the aggregation and combination phases. PEFT may introduce extra irregular memory accesses, thereby causing longer processing latency. The proposed Graph-CIM jointly considers the characteristics of GCN applications and the properties of the hybrid architecture and can adaptively adjust the allocation of workloads based on the current status of the hybrid architecture. These optimization techniques effectively reduce the processing latency of GCNs.

1) Graph Data. https://github.com/EdisonLeeeee/GraphData. 2017.

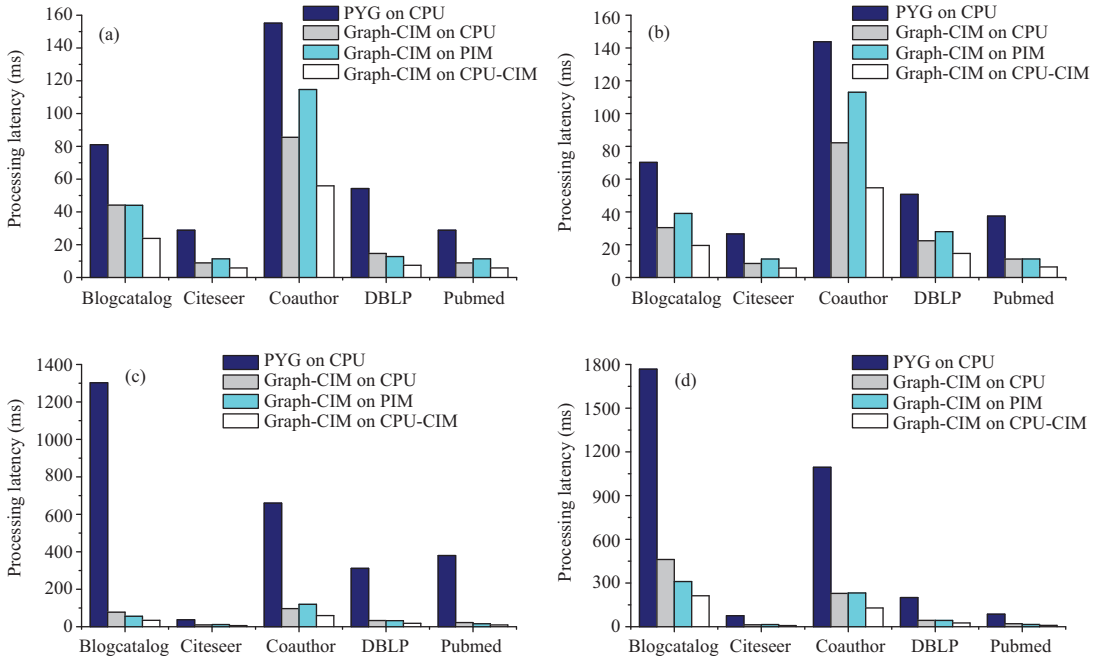


**Figure 7** (Color online) The processing latency of GCNs on four different types of architectures. (a) GCN; (b) DAGNN; (c) GWNN; (d) SGC.

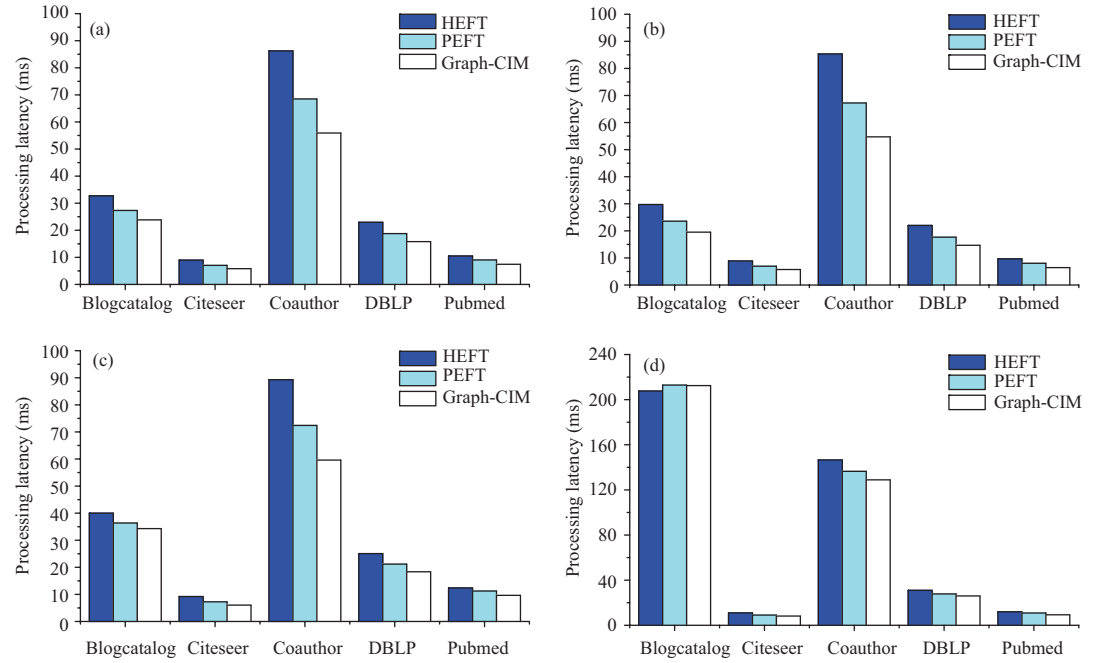


**Figure 8** (Color online) The processing latency of HEFT [13], PEFT [14], and Graph-CIM on the hybrid architecture. (a) GCN; (b) DAGNN; (c) GWNN; (d) SGC.

### 4.2.2 *Utilization ratio of processing units*

This subsection presents the experimental results of the utilization ratio of the general-purpose CPU and 3D-stacked CIM within the hybrid architecture. Figure 9 illustrates the experimental results for three scheduling algorithms, HEFT, PEFT, and Graph-CIM, with different GCN models. The input DAG in the scheduling is also abstracted from the log files of PyTorch and TensorboardX.

As shown in Figure 9, PEFT achieves the highest utilization ratio by adopting fine-grained task partitioning. Although PEFT has a better utilization ratio than Graph-CIM, this comes at the cost of eliminating data locality. Graph-CIM has a different design concept, where the utilization ratio is not

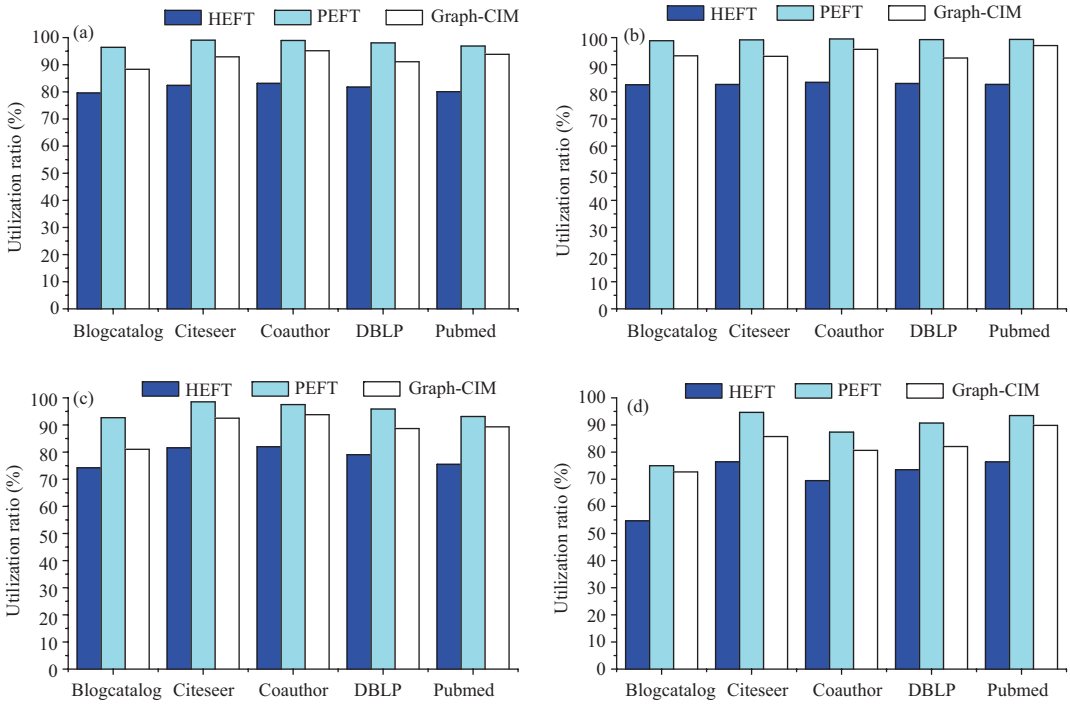


**Figure 9** (Color online) The utilization ratios of HEFT [13], PEFT [14], and Graph-CIM on the hybrid architecture. (a) GCN; (b) DAGNN; (c) GWNN; (d) SGC.

the primary design objective. Instead, Graph-CIM aims to reduce the processing latency by jointly considering the utilization of processing units in the hybrid architecture and the properties of the GCN workloads. The task-allocation process migrates the workloads to idle processing units to balance GCN processing across different types of processing units. Therefore, Graph-CIM may experience a slight degradation in terms of its utilization ratio.

## 5 Related work

**Accelerators for GCN.** Several hardware accelerators adopt GPU [15] or field-programmable gate array (FPGA) [16–18] to cater to the specific characteristics of GCN applications. Different GCN applications can be optimized using flexible general-purpose processing units or application-specific accelerators. In contrast, the proposed Graph-CIM adopts both general-purpose processing units and the memory-efficient CIM architecture in its design. This hybrid architecture can handle the unique properties of GCNs, especially the different computation requirements of the aggregation and combination phases. Therefore, Graph-CIM can still achieve the accuracy and computational efficiency of both general-purpose processing units and the CIM architecture.

**CIM architecture.** CIM or processing-in-memory (PIM) architectures can place computational resources inside or near the memory to speed up application processing. The resurgence of CIM is motivated by several new technologies, such as 3D-stacked memory, application-specific hardware accelerators, and data-intensive workloads [19–21]. CIM architecture can be implemented in emerging memory technologies. Among them, ReRAM-based solutions enable processing within the memory device [22–28]. Some studies adopt crossbar [29] or spin-orbit torque magnetic random access memory-based (SOT-MRAM) [30] CIM architecture to accelerate the workload processing. Other studies propose a novel CIM architecture by modifying the conventional DRAM architecture [31]. These studies can provide efficient processing of neural network applications.

By effectively reducing of data movement across different memory components, the CIM architecture provides a promising solution to improve the performance of different systems. Several CIM architectures have been proposed to solve specific problems [32–37]. Other studies have explored the in- or near-storage computing to improve the efficiency of nonvolatile memory and other storage systems [38,39]. These CIM architectures and solutions can offer fast near-data processing to reduce data movement. In contrast,

Graph-CIM adopts a different CIM architecture that focuses on the 3D-stacked PIM architecture, like HMC. Both the system abstractions of applications and memory-resources modeling in Graph-CIM can be applied to other CIM architectures to further improve application efficiency.

Dai et al. [40] proposed a PIM architecture for large-scale graph processing. It adopts a 3D-stacked PIM architecture with HMC. Other PIM architectures also adopt the HMC-based architecture to exploit parallelism for neural network applications [41–43]. Several solutions provide power modeling and power management for PIM architecture [44,45]. However, Graph-CIM is a task-scheduling algorithm to reduce the processing latency that adopts not only the 3D-stacked PIM architecture, but also the general-purpose processing units to form a hybrid architecture. Graph-CIM provides a viable solution for the inference of GCNs and fully utilizes the hybrid architecture. Graph-CIM may solve other types of graph-processing problems to further reduce processing latency.

**GCNs scheduling and optimization.** The hybrid computational pattern and enormous computation demand are key challenges for the processing of GCNs. Several techniques have been proposed to accelerate the training or inference process of GCNs by using pipeline, vertex/node reordering, or data reusing [46–48]. Several techniques such as collapsing weight matrices, removing nonlinearities, and sampling are adopted to eliminate unnecessary data transfers [49].

Graph-CIM is different from the above techniques. Graph-CIM is a software-based scheduling framework that can ease the burden of application-specific hardware design and efficiently minimize migration overhead. Graph-CIM adopts the task-allocation strategy to speed up the inference of GCNs on two different types of processing units. Therefore, Graph-CIM can be combined with the above techniques to further improve the utilization of constrained memory resources and improve the scheduling efficiency of various GCN applications.

## 6 Conclusion

In this paper, we proposed Graph-CIM, a novel data processing framework to optimize the task allocation of GCNs on hybrid CIM architecture. Graph-CIM exploits fine-grained parallelism to fully utilize the processing resources of both general purpose processing units and the 3D-stacked CIM architecture. The allocation of the aggregation and combination phases is jointly determined by the utilization status and properties of processing resources. We demonstrate the effectiveness of our approach by using a set of standard GCN workloads. The experimental results show that the proposed approach can effectively reduce the processing latency and fully utilize the processing units of the hybrid architecture.

In the future, we plan to investigate the simplified memory-access model of the CIM architecture, as it can potentially eliminate redundant data accesses to further improve the performance of our approach. We also plan to jointly study the effects of energy and processing latency. This can provide an integrated solution to further boost the system performance of the hybrid CIM architecture.

**Acknowledgements** This work was supported in part by National Natural Science Foundation of China (Grant No. 61972259), and Guangdong Basic and Applied Basic Research Foundation (Grant Nos. 2019B151502055, 2017B030314073, 2018B030325002).

**References**

1 Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks. In: Proceedings of the 5th International Conference on Learning Representations (ICLR), 2017. 1–14
2 Xie P, Sun G, Wang F, et al. V-PIM: an analytical overhead model for processing-in-memory architectures. In: Proceedings of IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2018. 107–108
3 Roy A, Mihailovic I, Zwaenepoel W. X-Stream: edge-centric graph processing using streaming partitions. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP), 2013. 472–488
4 Yuan P, Zhang W, Xie C, et al. Fast iterative graph computation: a path centric approach. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2014. 401–412
5 Liu M, Gao H, Ji S. Towards deeper graph neural networks. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2020. 338–348
6 Xu B, Shen H, Cao Q, et al. Graph wavelet neural network. In: Proceedings of the 7th International Conference on Learning Representations (ICLR), 2019. 1–13
7 Wu F, Zhang T Y, de Souza J A H, et al. Simplifying graph convolutional networks. In: Proceedings of the 36th International Conference on Machine Learning (ICML), 2019. 6861–6871
8 Kim D, Kung J, Chai S, et al. Neurocube: a programmable digital neuromorphic architecture with high-density 3D memory. In: Proceedings of ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016. 380–392
9 Arai J, Shiokawa H, Yamamuro T, et al. Rabbit order: just-in-time parallel reordering for fast graph analysis. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016. 22–31
10 Paszke A, Gross S, Massa F, et al. PyTorch: an imperative style, high-performance deep learning library. In: Proceedings of Advances in Neural Information Processing Systems, 2019. 8026–8037
11 Pawlowski J T. Hybrid memory cube (HMC). In: Proceedings of IEEE Hot Chips 23 Symposium (HCS), 2011. 1–24

12 Fey M, Lenssen J E. Fast graph representation learning with PyTorch Geometric. In: Proceedings of ICLR Workshop on Representation Learning on Graphs and Manifolds, 2019. 1–9
13 Topcuoglu H, Hariri S, Wu M-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distrib Syst, 2002, 13: 260–274
14 Xu D, Liao Y, Wang Y, et al. Selective off-loading to memory: task partitioning and mapping for PIM-enabled heterogeneous systems. In: Proceedings of the Computing Frontiers Conference (CF), 2017. 255–258
15 Chang F, Dean J, Ghemawat S, et al. Bigtable: a distributed storage system for structured data. ACM Trans Comput Syst, 2008, 26: 1–26
16 Zhang B, Zeng H, Prasanna V. Accelerating large scale GCN inference on FPGA. In: Proceedings of IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2020. 241
17 Zhang B, Zeng H, Prasanna V. Hardware acceleration of large scale GCN inference. In: Proceedings of IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2020. 61–68
18 Wang H, Wang K, Yang J, et al. GCN-RL circuit designer: transferable transistor sizing with graph neural networks and reinforcement learning. In: Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC), 2020. 1–6
19 Guo X X, Xiang S Y, Zhang Y H, et al. Enhanced memory capacity of a neuromorphic reservoir computing system based on a VCSEL with double optical feedbacks. Sci China Inf Sci, 2020, 63: 160407
20 Cheng W, Cai R, Zeng L F, et al. IMCI: an efficient fingerprint retrieval approach based on 3D stacked memory. Sci China Inf Sci, 2020, 63: 179101
21 Xi K, Bi J S, Majumdar S, et al. Total ionizing dose effects on graphene-based charge-trapping memory. Sci China Inf Sci, 2019, 62: 222401
22 Zha Y, Nowak E, Li J. Liquid silicon: a nonvolatile fully programmable processing-in-memory processor with monolithically integrated ReRAM. IEEE J Solid-State Circ, 2020, 55: 908–919
23 Li Z, Yan B, Li H. ReSiPE: ReRAM-based single-spiking processing-in-memory engine. In: Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC), 2020. 1–6
24 Zheng Q, Wang Z, Feng Z, et al. Lattice: an ADC/DAC-less ReRAM-based processing-in-memory architecture for accelerating deep convolution neural networks. In: Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC), 2020. 1–6
25 Gupta S, Imani M, Sim J, et al. SCRIMP: a general stochastic computing architecture using ReRAM in-memory processing. In: Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE), 2020. 1598–1601
26 Yang X, Yan B, Li H, et al. ReTransformer: ReRAM-based processing-in-memory architecture for transformer acceleration. In: Proceedings of IEEE/ACM International Conference On Computer Aided Design (ICCAD), 2020. 1–9
27 Wang F, Shen Z, Han L, et al. ReRAM-based processing-in-memory architecture for blockchain platforms. In: Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASPDAC), 2019. 615–620
28 Han L, Shen Z, Liu D, et al. A novel ReRAM-based processing-in-memory architecture for graph traversal. ACM Trans Storage, 2018, 14: 1–26
29 Chu C, Wang Y, Zhao Y, et al. PIM-Prune: fine-grain DCNN pruning for crossbar-based process-in-memory architecture. In: Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC), 2020. 1–6
30 Angizi S, He Z, Rakin A S, et al. CMP-PIM: an energy-efficient comparator-based processing-in-memory neural network accelerator. In: Proceedings of the 55th ACM/ESDA/IEEE Design Automation Conference (DAC), 2018. 1–6
31 Yang Y, Chen X, Han Y. Dadu-CD: fast and efficient processing-in-memory accelerator for collision detection. In: Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC), 2020. 1–6
32 Liu Z, Ren E, Qiao F, et al. NS-CIM: a current-mode computation-in-memory architecture enabling near-sensor processing for intelligent IoT vision nodes. IEEE Trans Circuits Syst I, 2020, 67: 2909–2922
33 Imani M, Pampana S, Gupta S, et al. DUAL: acceleration of clustering algorithms using digital-based processing in-memory. In: Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020. 356–371
34 Wan Z, Dai G, Soh Y J, et al. An order sampling processing-in-memory architecture for approximate graph pattern mining. In: Proceedings of the 2020 on Great Lakes Symposium on VLSI (GLSVLSI), 2020. 357–362
35 Xu S, Chen X, Qian X, et al. TUPIM: a transparent and universal processing-in-memory architecture for unmodified binaries. In: Proceedings of the 2020 on Great Lakes Symposium on VLSI (GLSVLSI), 2020. 199–204
36 Kwon Y, Lee Y, Rhu M. TensorDIMM: a practical near-memory processing architecture for embeddings and tensor operations in deep learning. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2019. 740–753
37 Gupta S, Imani M, Kaur H, et al. NNPIM: a processing in-memory architecture for neural network acceleration. IEEE Trans Comput, 2019, 68: 1325–1337
38 Imani M, Gupta S, Sharma S, et al. NVQuery: efficient query processing in nonvolatile memory. IEEE Trans Comput-Aided Des Integr Circ Syst, 2019, 38: 628–639
39 Chen C H, Hsia T Y, Huang Y, et al. Data prefetching and eviction mechanisms of in-memory storage systems based on scheduling for big data processing. IEEE Trans Parallel Distrib Syst, 2019, 30: 1738–1752
40 Dai G, Huang T, Chi Y, et al. GraphH: a processing-in-memory architecture for large-scale graph processing. IEEE Trans Comput-Aided Des Integr Circ Syst, 2019, 38: 640–653
41 Wang Y, Zhang M, Yang J. Exploiting parallelism for convolutional connections in processing-in-memory architecture. In: Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference (DAC), 2017. 1–6
42 Wang Y, Chen W, Yang J, et al. Towards memory-efficient allocation of CNNs on processing-in-memory architecture. IEEE Trans Parallel Distrib Syst, 2018, 29: 1428–1441
43 Wang Y, Chen W, Yang J, et al. Exploiting parallelism for CNN applications on 3D stacked processing-in-memory architecture. IEEE Trans Parallel Distrib Syst, 2019, 30: 589–600
44 Sun H, Zhu Z, Cai Y, et al. An energy-efficient quantized and regularized training framework for processing-in-memory accelerators. In: Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASP-DAC), 2020. 325–330
45 Zhang C, Meng T, Sun G. PM3: power modeling and power management for processing-in-memory. In: Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018. 558–570
46 Geng T, Li A, Shi R, et al. AWB-GCN: a graph convolutional network accelerator with runtime workload rebalancing. In: Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020. 922–936
47 Yan M, Deng L, Hu X, et al. HyGCN: a GCN accelerator with hybrid architecture. In: Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA), 2020. 15–29
48 Liang S, Wang Y, Liu C, et al. EnGN: a high-throughput and energy-efficient accelerator for large graph neural networks. IEEE Trans Comput, 2021. doi: 10.1109/TC.2020.3014632
49 Hamilton W L, Ying R, Leskovec J. Inductive representation learning on large graphs. In: Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS), 2017. 1025–1035