This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2821565, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

GraphH: A Processing-in-Memory Architecture for Large-scale Graph Processing

Guohao Dai, *Student Member, IEEE,* Tianhao Huang, Yuze Chi, Jishen Zhao, *Member, IEEE,* Guangyu Sun, *Member, IEEE,* Yongpan Liu, *Senior Member, IEEE,* Yu Wang, *Senior Member, IEEE,* Yuan Xie, *Fellow, IEEE,* Huazhong Yang, *Senior Member, IEEE*

***Abstract*—Large-scale graph processing requires the high bandwidth of data access. However, as graph computing continues to scale, it becomes increasingly challenging to achieve a high bandwidth on generic computing architectures. The primary reasons include: the random access pattern causing local bandwidth degradation, the poor locality leading to unpredictable global data access, heavy conflicts on updating the same vertex, and unbalanced workloads across processing units. Processing-in-memory has been explored as a promising solution to providing high bandwidth, yet open questions of graph processing on PIM devices remain in: (1) How to design hardware specializations and the interconnection scheme to fully utilize bandwidth of PIM devices and ensure locality; (2) How to allocate data and schedule processing flow to avoid conflicts and balance workloads.**

In this paper, we propose GraphH, a PIM architecture for graph processing on the Hybrid Memory Cube array, to tackle all four problems mentioned above. From the architecture perspective, we integrate SRAM-based On-chip Vertex Buffers to eliminate local bandwidth degradation; We also introduce Reconfigurable Double-Mesh Connection to provide high global bandwidth. From the algorithm perspective, partitioning and scheduling methods like Index Mapping Interval-Block and Round Interval Pair are introduced to GraphH, thus workloads are balanced and conflicts are avoided. Two optimization methods are further introduced to reduce synchronization overhead and reuse on-chip data. The experimental results on graphs with billions of edges demonstrate that GraphH outperforms DDR-based graph processing systems by up to two orders of magnitude and 5.12x speedup against the previous PIM design [1].

***Index Terms*—large-scale graph processing, Hybrid Memory Cube (HMC), memory hierarchy, on-chip networks**

Manuscript received by November 9th, 2017; revised by January 24th, 2018; accepted by March 7th, 2018; date of current version is March 21st, 2018. This work was supported by National Natural Science Foundation of China (No. 61622403, 61621091), National Key R&D Program of China 2017YFA0207600, Joint fund of Equipment pre-Research and Ministry of Education (No. 6141A02022608), and Beijing National Research Center for Information Science and Technology (BNRist). Guohao Dai is also supported by China Scholarship Council for his current visiting at University of California, Berkeley, CA, USA.

G. Dai, T. Huang, Y. Liu, Y. Wang, and H. Yang are with the Department of Electronic Engineering, Tsinghua University, Beijing, China, and Beijing National Research Center for Information Science and Technology (BNRist), 100084, e-mail: dgh14@mails.tsinghua.edu.cn, yu-wang@tsinghua.edu.cn

Y. Chi is with the Computer Science Department, University of California, Los Angeles, CA, USA, 90095.

J. Zhao is with the Computer Science and Engineering Department, Jacobs School of Engineering, University of California, San Diego, CA, USA, 92093.

G. Sun is with the Center for Energy-efficient Computing and Applications (CECA), School of EECS, Peking University, Beijing, China, 100871.

Y. Xie is with the Department of Electrical and Computer Engineering University of California at Santa Barbara, CA, USA, 93106.

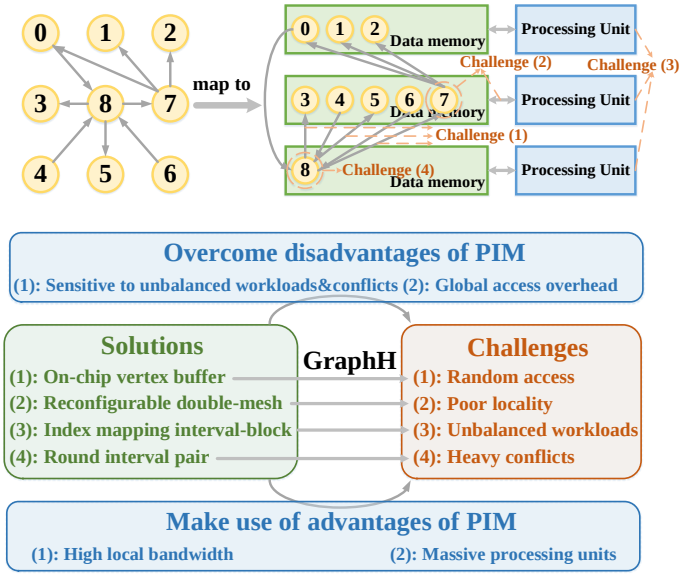


Fig. 1. Challenges in large-scale graph processing and solutions in GraphH. Our system makes use of advantages of PIM and overcomes disadvantages of PIM, providing solutions to each challenge in graph processing.

I. Introduction

As we are now in the "big data" era, the data volume collected from various digital devices has skyrocketed in recent years. Meanwhile, ever-growing analysis demand over these data brings tremendous challenges to both analytical models and computing architectures. Graph, a conventional data structure, can both store the data value and represent the relationships among data. The large-scale graph processing problem is gaining increasing attention in various domains. Many systems have been proposed and achieved significant performance improvement over large-scale graph processing problems, including Tesseract [1], Graphicionado [2], Graph-PIM [3], Gemini [4], GraphLab [5], etc [6]–[9].

The essential way to improve the performance of large-scale graph processing is to provide a higher bandwidth of data access. However, achieving high bandwidth of graph processing on conventional architectures suffers from four challenges: (1) **Random access.** The data access pattern in major graph algorithm is highly irregular, which leads to considerable local bandwidth degradation (e.g., >90% bandwidth degradation on CPU-DRAM hierarchy [10]) on conventional computation

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2821565, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE I
NOTATIONS OF A GRAPH

Notation	Meaning
G	a graph $G = (V, E)$
V	vertices in G, $\|V\| = n$
E	edges in G, $\|E\| = m$
v_i	vertex i
$e_{i.j}$	edge from v_i to v_j
e_{src}, e_{dst}	source & destination vertex of edge e
I_x	interval x
S_y	shard y, containing $e_{i.j}$ where $v_j \in I_y$
$B_{x.y}$	block $x.y$, containing $e_{i.j}$ where $v_i \in I_x$ and $v_j \in I_y$

architecture. (2) **Poor locality**. A simple operation on one vertex may require access to all its neighbor vertices. Such poor locality leads to irregular global data access and inefficient bandwidth utilization (e.g., 57% bandwidth degradation on 2-hop [11]) on multi-processing units architecture. (3) **Unbalanced workloads**. The graph is highly unstructured, thus the computation workloads of various processing units can be quite unbalanced. (4) **Heavy conflicts**. Simultaneous updating to the same vertex by different processing units causes heavy conflicts. All these four challenges need to be addressed from both architecture and algorithm perspectives.

Processing-in-memory (PIM) has been put forward as a promising solution to providing high bandwidth for big data problems. PIM achieves speedup at several orders of magnitude over many applications [1], [3], [6], [12], [13]. Various processing units can be put into the memory and attached to a part of the memory in PIM. The total theoretical bandwidth of all units under the PIM architecture can be 10x to 100x larger than that under conventional computing architectures [1]. Moreover, PIM scales well to large-scale problems because it can provide proportional bandwidth to the memory capacity.

Although PIM can provide advantages such as high bandwidth and massive processing units (Fig. 1), it still suffers from inherent disadvantages in graph processing. For example, PIM is sensitive to unbalanced workloads and conflicts as well as other multicore architectures. Moreover, achieving high bandwidth in large-scale graph processing suffers from both the random access pattern and the poor locality, these problems remain in PIM. To tackle all these problems, previous works have proposed several solutions. Tesseract [1] introduced the prefetcher to exploit locality, but such prefetching strategy cannot avoid global random access to graph data. GraphPIM [3] proposed the instruction off-loading scheme for Hybrid Memory Cube (HMC) based graph processing, while how to make use of multiple HMCs is not presented.

Therefore, using multiple PIM devices (e.g., Hybrid Memory Cube array) can be a scalable solution for large-scale graph processing. Although Tesseract [1] has adopted the HMC array structure for graph processing, open questions remain in how to fully utilize the high bandwidth provided by PIM devices. In this paper, we propose GraphH, a Hybrid Memory Cube array architecture for large-scale graph processing problems. Compared with Tesseract [1], both hardware architecture and partitioning/scheduling algorithms are elaborately designed in GraphH. From the architecture perspective, we design specialized hardware units in the logic layer of HMC, including

Algorithm 1 Pseudo-code of Edge-Centric Model [14]

Input: $G = (V, E)$, initialization condition
Output: Updated V

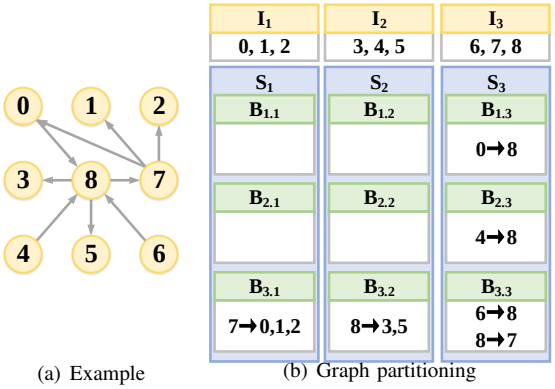
```
for each v ∈ V do
    Initialize(v, initialization condition)
end for
while (not finished) do
    for each e ∈ E do
        value(e_dst) = Update(e_src, e_dst)
    end for
end while
return V
```

SRAM-based **On-chip Vertex Buffers (OVB)**, to fully exploits the local bandwidth of a vault; We also propose the **Reconfigurable Double-Mesh Connection (RDMC)** scheme to ensure locality and provide high global access bandwidth in graph processing. From the algorithm perspective, we propose the **Index Mapping Interval-Block (IMIB)** partitioning method to balance the workloads of different processing units; Scheduling method like **Round Interval Pair (RIP)** is also introduced to avoid writing conflicts. All these four designs, which are not introduced in Tesseract [1], lead to performance improvements (detailed in Section VI-C, 4.58x using OVB, 1.29x using RDMC+RIP, 3.05x using IMIB, averagely). Contributions of this paper are concluded as follows:

- We integrate the On-chip Vertex Buffer in GraphH. Processing units can directly access OVB and do not suffer from random access pattern.
- We connect cubes in GraphH using the Reconfigurable Double Mesh Connection scheme to provide high global bandwidth and ensure locality.
- We partition graphs using Index Mapping Interval-Block method to balance workloads of cubes.
- We schedule the data transferring among cubes using Round Interval Pair scheme. Communication among cubes are organized in pairs and conflicts are avoided.
- We propose two optimization methods to reduce synchronization overhead and reuse on-chip data. Thus, we further improve the performance of GraphH.

We have also conducted extensive experiments to evaluate the performance of the GraphH system. We choose both real-world and synthetic large-scale graphs as our benchmarks and test three graph algorithms over these graphs. Our evaluation results show that GraphH remarkably outperforms DDR-based graph processing systems by up to two orders of magnitude and achieves up to 5.12x speedup compared with Tesseract [1].

The rest of this paper is organized as follows. Section II introduces the background information of graph processing models and Hybrid Memory Cubes. Section III proposes the architecture of GraphH. Then, the processing flow in GraphH is detailed in Section IV. We further propose two optimization methods to improve the performance of GraphH in Section V. Results of comprehensive experiments are shown in Section VI. Related works are introduced in Section VII and we conclude this paper in Section VIII.



(a) Example (b) Graph partitioning

Fig. 2. Intervals & blocks (Vertices and edges are divided into 3 intervals and 3×3=9 blocks).

II. Preliminary and Background

In this section, we introduce the background information of both large-scale graph processing and PIM technology. The notations of our graph abstraction used in this and following sections are shown in Table I.

A. Graph Abstraction

Given a graph $G = (V, E)$, where V and E denote vertex and edge set of G, the computation task over G is to update the value of V. We assume that the graph is directed so that each edge $e_{i.j}$ in G is associated with a source vertex v_i and a destination vertex v_j. The undirected graph can be implemented by adding an opposite edge $e_{j.i}$ to $e_{i,j}$ in G.

To build a general graph processing framework, most large-scale graph processing systems, including GraphLab [5] and Pregel [7], adopt a high-level graph algorithm abstraction model called Vertex-Centric Model (VCM). VCM is a commonly used model applied to different graph algorithms. In VCM, graph processing is divided into iterations. Each vertex needs to communicate with all its neighbor vertices to perform the updating operation. Edge-Centric Model (ECM) [14] describes VCM from another perspective.

Algorithm 1 shows a detailed example of graph processing under the Edge-Centric Model. In ECM, value of different vertices is initialized at the beginning of the program (Line 2 in Algorithm 1). Then, the algorithm is executed in the step of iterations, and each edge is traversed in an iteration (Line 4 to Line 8 in Algorithm 1). When an edge is accessed, the destination vertex is updated using the value of source vertex (Line 6 in Algorithm 1). ECM is a high-level abstraction model for graph processing, different algorithms only differ in the Initialize() and Update() function. For example, in Breadth-First Search (BFS), all vertices value is set to infinity while the value of root vertex is set to zero using Initialize(). In the Update(), a destination vertex will be updated if the depth of a source vertex is smaller than its depth.

B. Graph Partitioning

Graph partitioning is commonly used to ensure the locality of graph data access. Many previous works proposed graph partitioning strategies. Distributed/Multi-core systems like Gemini [4] and Polymer [11] mainly focused on issues like locality, cache coherence, low-overhead scaling out designs, etc. NXgraph [8] proposed an *interval-block* based graph partitioning method. After preprocessing, all vertices in the graph are divided into P disjointed intervals. Edges are divided into P shards according to their destination vertices. Furthermore, edges in a shard are divided into P blocks according to their source vertices.

Fig. 2 shows an example of this *interval-block* graph partitioning method. Vertices in the graph are divided into 3 intervals, 3 shards, and 3×3=9 blocks. Each edge is assigned to a block according to it source and destination vertices. For example, interval I_2 contains vertices v_3, v_4, v_5, interval I_3 contains vertices v_6, v_7, v_8, thus block $B_{2.3}$ contains edges $e_{3.8}, e_{4.8}, e_{5.8}$. In this graph partitioning method, when one interval I_x updates another interval I_y using corresponding block $B_{x.y}$, other intervals and blocks will not be accessed.

C. Hybrid Memory Cube

The architecture of processing-in-memory (PIM) has been proposed to provide a higher bandwidth from the perspective of hardware. By allocating processing units inside memory, PIM achieves *memory-capacity-proportional* bandwidth so that can scale to large-scale problems.

3D die-stacking memory devices, such as Hybrid Memory Cube (HMC) [15], allow us to implement PIM with high bandwidth memory. For example, an HMC device in a single package contains multiple memory layers and one logic layer. These layers are stacked together and use through-silicon via (TSV) technology as connections to achieve a high bandwidth. In an HMC, memory and logic layers are organized into vertical vaults, which can perform computation independently. The latest HMC devices can provide at most 8 GB memory space, and up to 480 GB/s external bandwidth (4 multiple serial links, each with a default of 16 input lanes and 16 output lanes for full duplex operation) referred to the Micron HMC 2.1 specification [16]. Inside an HMC, there are 32 vaults. Each vault consists of a logic layer and several memory layers which can provide up to 256 MB of memory space and 10 GB/s of bandwidth (16 GB/s in Tesseract [1]). Thus, the maximum aggregate internal bandwidth of an HMC device can be up to 320 GB/s according to the HMC 2.1 specification (512 GB/s in Tesseract). Because of advantages like high internal bandwidth and low cost of moving data, the HMC has been taken into account as an effective solution for some data-intensive applications[1], [3], [12]. For example, Gao *et al.* [12] used HMCs for neural computing and achieved 4.1x improvement compared with conventional architectures.

Although Ahn *et al.* [1] proposed Tesseract to implement graph processing using PIM device, they didn't fully utilize the high bandwidth of PIM. In Tesseract, only one prefetcher is integrated as a queue for messages without giving much thought to graph data access patterns. As result, Tesseract involves unpredictable global access, leading to low bandwidth utilization (e.g. <3000GB/s bandwidth usage while 16 HMCs can provide up to 16GB/s×32×16=8192GB/s bandwidth, <37% bandwidth utilization).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2821565, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

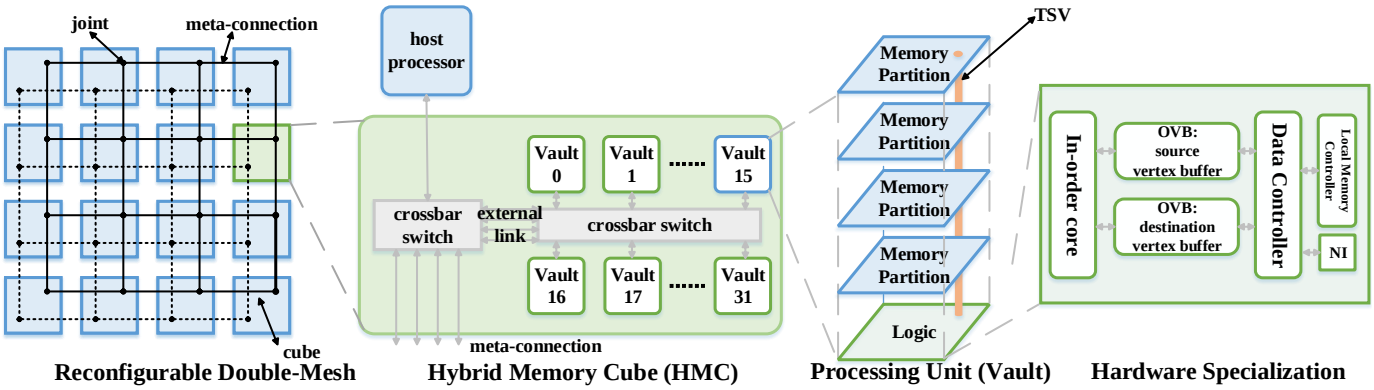


Fig. 3. Detailed architecture of GraphH (From left to right: the cube array and the reconfigurable double-mesh connection controlled by the host processor through the crossbar switch, the HMC architecture, the 3D-stacking architecture in a vault, the detailed implementation of the logic layer).

III. GraphH Architecture

Based on background information in Section II, we design the GraphH. We use the Hybrid Memory Cube as an example implementation of GraphH.

A. Overall Architecture

Our GraphH implementation consists of a 16-cube array (same as Tesseract [1]), shown in Fig. 3. Inside each HMC, there are 32 vaults and a crossbar switch to connect them to external links. A vault consists of both logic layer and memory layers to execute graph processing operation. A host processor is responsible for data allocation and interconnection configuration.

HMC and double-mesh. GraphH physically arranges 16 cubes in a 2-D array structure. To connect 16 cubes, GraphH uses a double-mesh structure as interconnections (represented by the solid line and the dotted line in Fig. 3). The connectivity of this double-mesh can be dynamically reconfigured at runtime, which is detailed in Section III-C.

Vault and crossbar. The vault is the basic unit of HMC. An HMC is composed of 32 vaults and provides high bandwidth to each vault. According to the HMC 2.1 specification, such a collective internally available bandwidth from all these 32 vaults is made accessible to the I/O links using a crossbar switch.

Layer and TSV. Inside each vault, there are several memory layers and a logic layer. These layers are stacked together using through-silicon via (TSV) [17]. According to the HMC 2.1 specification, the bandwidth between memory layers and the logic layer of each vault can be up to 10 GB/s. We implement a simple in-order core, two specific on-chip vertex buffers, the data controller, and the network logic in the logic layer.

B. On-chip Vertex Buffer

Data access pattern in graph processing is highly randomized, leading to unpredictable latency (e.g., 2x∼3x latency difference between global and local access [11]) of accessing different vertices/edges and efficient bandwidth degradation (e.g., 57% bandwidth degradation on 2-hop [11]). To overcome the unpredictable latency and bandwidth loss, we introduce On-chip Vertex Buffers (OVB) as the bridge between the in-order core and the memory.

In the Edge-Centric Model, updating is propagated from the source vertex to the destination vertex. Both source vertices and destination vertices are randomly accessed. Two types of OVB, the source vertex buffer and the destination vertex buffer, are integrated into the logic layer of a vault using SRAM. Instead of directly access vertices stored in DRAM layers when processing, GraphH firstly loads vertices data into OVB. The in-order core can directly access vertices value stored in OVB in the random pattern without bandwidth degradation. After all vertices in the destination vertex buffer have been updated (all corresponding in-edges have been accessed by the in-order core), GraphH flushes data in the destination vertex buffer back to DRAM layers and starts to process other vertices in the same way.

By adopting OVB in the logic layer, the random access pattern of DRAM layers is avoided. Data are sequentially loaded from and written back to the stacked memory. We assume that the logic layer has the same size as dram layer. According to Micron HMC 2.1 specification [16], each DRAM layer is 8 Gb. We use Cacti 6.5 [18] to evaluate the area properties of both SRAMs and DRAMs for the sake of consistency. The area of an 8 Gb DRAM with 64-bit input/output bus (data+address) under 32 nm process is 257.02 mm^2 [1], the area consumption is 1.65 mm^2 under 32 nm process. By allocating two OVBs (source vertex buffer and destination vertex buffer) to each buffer, the total area consumption of OVBs is 105.6 mm^2, which is less than half of the area of the logic layer.

C. Reconfigurable Double-Mesh Connection

According to Micron HMC 2.1 specification [16], each cube has 8 high-speed serial links providing up to 480 GB/s aggregate link bandwidth. Many previous researchers have studied the inter-cube connection scheme, and nearly most of them are static based on routers, which means all connections are pre-established. Sethia *et al.* [19] proposed and analyzed the interconnection, the topologies can be mesh, flattened butterfly, dragonfly, etc. Under the static connection scheme,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2821565, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

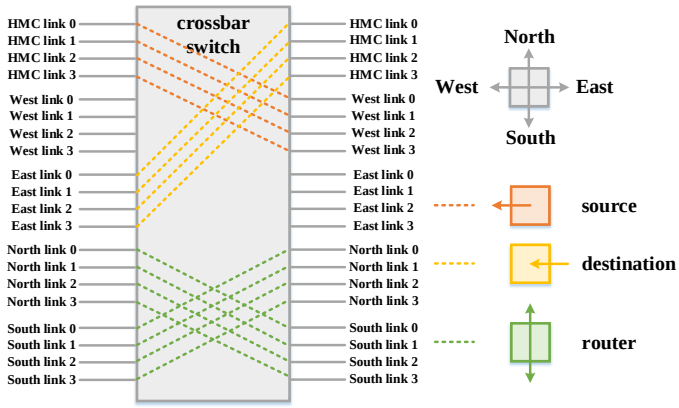


Fig. 4. Examples of reconfiguring double mesh connection using the crossbar switch. Lines in orange, yellow and green represent the way of configuring the corresponding cube into a source, destination, and router cube respectively.

a cube is often connected to 3 to 4 adjacent cubes, and such 480 GB/s external link will be partitioned by these cubes with the bandwidth around 120 GB/s to 160 GB/s. However, the aggregate internal bandwidth of a cube is 320 GB/s (10 GB/s per vault), thus such static interconnection scheme cannot fully utilize the internal bandwidth of HMC when transferring data among cubes.

To avoid disadvantages of using static connections based on routers, GraphH adopts a reconfigurable double-mesh connection (RDMC) scheme shown in Fig. 3. A mesh consists of 24 meta-connections (connection between two adjacent cubes) and 48 joints. Each mesh has individual data path and owns exclusive bandwidth. Each meta-connection can provide up to 480 GB/s physical bandwidth with the full duplex operation. The joint on each side of a meta-connection can be configured to connect external links of a cube or another meta-connection. In this way, two cubes can be connected using an exclusive link with up to 480 GB/s bandwidth. GraphH refers to a look-up table (LUT) in the host processor to store all configurations required by processing scheme (detailed in Section IV). In this way, GraphH can build up an exclusive data path for any two cubes in the array. To implement the reconfigurable connection scheme according to the pre-stored LUT, GraphH uses a crossbar switch [20] connected to 4 links and 4 meta-connections (each meta-connection has 4 individual links for 4 serial links of the cube, 4×4=16 in total). Fig. 4 shows an example of how the crossbar switch works. When a cube in a mesh network is the source/destination of data transferring, the crossbar switch connects 4 output/input serial links of the cube to the corresponding meta-connection (orange and yellow lines in Figure 4). When a cube is a routing node, the crossbar switch connects meta-connections without transferring data to the cube (green lines in Figure 4). The status of the crossbar is controlled by the host processor according to a pre-stored LUT. Moreover, the interconnection reconfiguration can be executed simultaneously during processing graphs in the cube (No data are transferred among cubes when processing graphs in a cube). Thus, such a reconfigurable connection scheme will not affect the whole GraphH performance. Compared with the static connection scheme, such reconfigurable connection

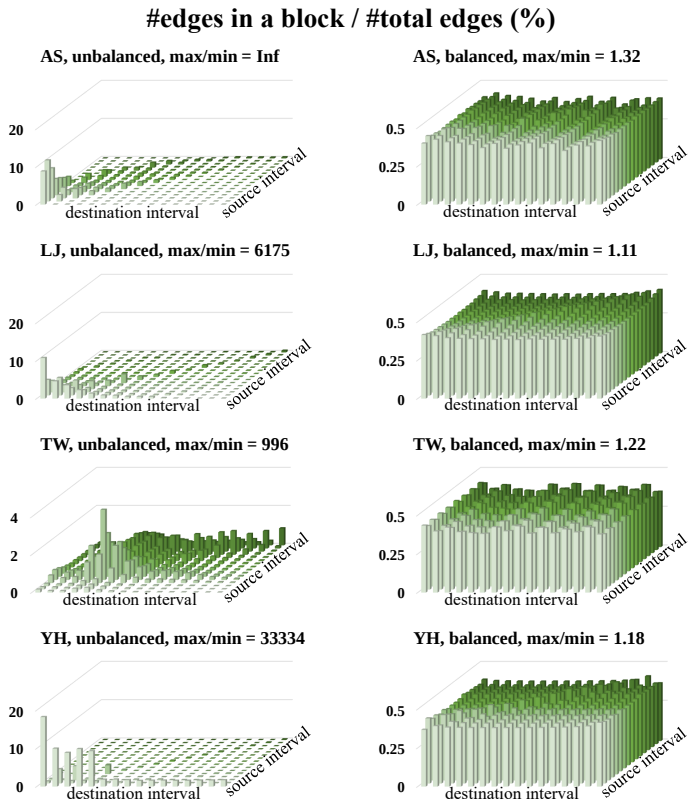


Fig. 5. Size of blocks: dividing consecutive vertices into an interval (left, unbalanced), dividing vertices using IMIB (right, balanced). Graphs are from Table III.

scheme mitigates the cost of using routers and maximizes the link bandwidth of two connected cubes.

IV. GraphH Processing

Based on the proposed GraphH architecture introduced in Section III, we introduce the graph processing flow of GraphH in this section, high level programming interface of GraphH is also introduced.

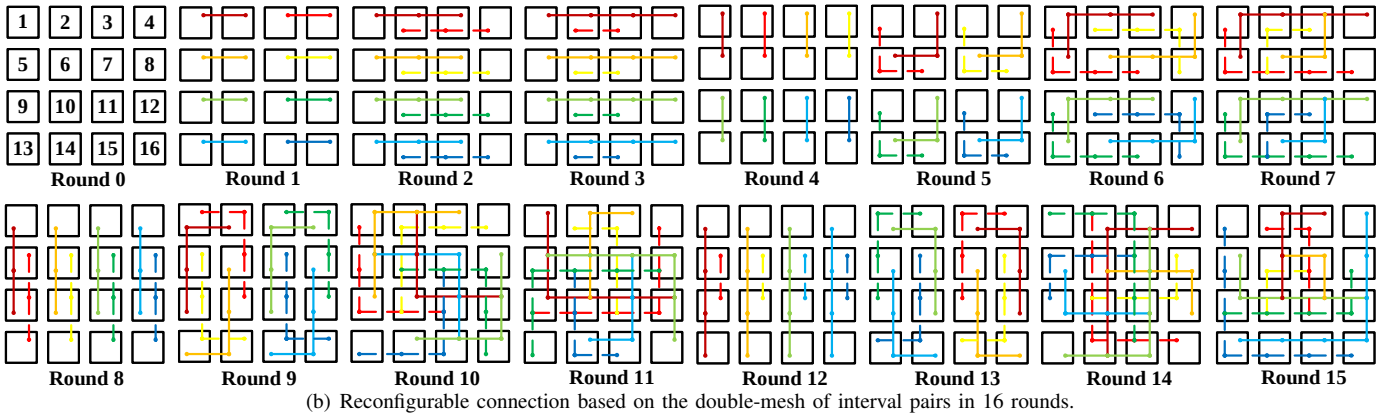
A. Overall Processing Flow

GraphH adopts Edge-Centric Model and Interval-Block partitioning method for graph processing tasks. In the pre-processing and data loading step, the vertex set V is divided into 16 equal-sized intervals and then assigned to each HMC cube; for each interval, the 16 corresponding in-edge blocks are also loaded into the same cube. In the next processing step, program execution is divided into iterations, as in Algorithm 2. Since ECM iterates over edges and requires both the source and destination vertex data of the edge to be available, each cube (Cube_x) would first receive a source interval (I_y) from another cube (Cube_y) in *Transferring Phase*. Then Cube_x performs *Updating Phase* to update I_x in parallel with the data stored in I_y and $B_{y,x}$.

To exploit vault-level parallelism, vertices in an interval are further divided into 32 disjoint small sets. Each vault is responsible for updating one small set with the corresponding in-edges. Thus, a block is also divided into 32 small sets and

Round 0: Self-updating
Round 1: 1↔2, 3↔4, 5↔6, 7↔8, 9↔10, 11↔12, 13↔14, 15↔16
Round 2: 1↔3, 2↔4, 5↔7, 6↔8, 9↔11, 10↔12, 13↔15, 14↔16
Round 3: 1↔4, 2↔3, 5↔8, 6↔7, 9↔12, 10↔11, 13↔16, 14↔15
Round 4: 1↔5, 2↔6, 3↔7, 4↔8, 9↔13, 10↔14, 11↔15, 12↔16
Round 5: 1↔6, 2↔5, 3↔8, 4↔7, 9↔14, 10↔13, 11↔16, 12↔15
Round 6: 1↔7, 2↔8, 3↔5, 4↔6, 9↔15, 10↔16, 11↔13, 12↔14
Round 7: 1↔8, 2↔7, 3↔6, 4↔5, 9↔16, 10↔15, 11↔14, 12↔13
Round 8: 1↔9, 2↔10, 3↔11, 4↔12, 5↔13, 6↔14, 7↔15, 8↔16
Round 9: 1↔10, 2↔9, 3↔12, 4↔11, 5↔14, 6↔13, 7↔16, 8↔15
Round 10: 1↔11, 2↔12, 3↔9, 4↔10, 5↔15, 6↔16, 7↔13, 8↔14
Round 11: 1↔12, 2↔11, 3↔10, 4↔9, 5↔16, 6↔15, 7↔14, 8↔13
Round 12: 1↔13, 2↔14, 3↔15, 4↔16, 5↔9, 6↔10, 7↔11, 8↔12
Round 13: 1↔14, 2↔13, 3↔16, 4↔15, 5↔10, 6↔9, 7↔12, 8↔11
Round 14: 1↔15, 2↔16, 3↔13, 4↔14, 5↔11, 6↔12, 7↔9, 8↔10
Round 15: 1↔16, 2↔15, 3↔14, 4↔13, 5↔12, 6↔11, 7↔10, 8↔9

(a) Interval pairs in 16 rounds.



(b) Reconfigurable connection based on the double-mesh of interval pairs in 16 rounds.

Fig. 6. The scheduling and connection of 16 cubes in different rounds of an updating iteration.

stored in each vault. The source interval is duplicated and stored in each vault.

B. Index Mapping Interval-Block Partition

The execution time of each cube is in positive correlation with sizes of both its vertices and edges [4]. All cubes need to be synchronized after processing. Thus, it is important to adopt a balanced partitioning method. Vertices are divided into intervals and assigned to cubes, as well as corresponding edges in blocks.

One naive way is to averagely divide consecutive vertices into an interval. For example, in a graph with 9 vertices, $v_1 \sim v_7, v_9, v_{11}$.[1] We divide all vertices into 3 intervals, the separation results are $I_1 = \{v_1 \sim v_3\}$, $I_2 = \{v_4 \sim v_6\}$, $I_3 = \{v_7, v_9, v_{11}\}$. However, when we implement this method on natural graphs (left in Fig. 5, Graphs are from Table III), sizes of different blocks are quite unbalanced, which leads to unbalanced workloads. The reason of unbalanced sizes is from the power-law [21] of natural graphs. A small fraction of vertices often pocess most of the edges in a graph. Moreover, these vertices often have consecutive indexes and are divided into one interval (e.g., $B_{2.1}$ in AS, $B_{1.1}$ in LJ, $B_{5.5}$ in TW, $B_{1.1}$ in YH, account for 10.86%, 10.73%, 3.71%, 18.19% of total edges respectively, left in Fig. 5).

To avoid the unbalanced workloads among cubes, we adopt the Index Mapping Interval-Block (IMIB) partitioning method. IMIB consists of two steps: (1) **Compression**. We compress vertex indexes by removing blank vertices. For example, $v_1 \sim v_7, v_9, v_{11}$ are mapped to $v_1 \sim v_9$ with $v_1 \sim v_7 \rightarrow v_1 \sim v_7, v_9 \rightarrow v_8, v_{11} \rightarrow v_9$. (2) **Hashing**. After mapping vertices to compressed indexes, we divide them into different interval using modulo function. For example, $v_1 \sim v_9$ (after compression) are divided into 3 intervals with $I_1 = \{v_1, v_4, v_7\}$, $I_2 = \{v_2, v_5, v_8\}$, $I_3 = \{v_3, v_6, v_9\}$.

With IMIB, sizes of both intervals and blocks can be balanced (right in Fig. 5, the ratios of largest and smallest blocks are 1.32x, 1.11x, 1.22x, and 1.18x respectively in three graphs, much smaller than the infinity (some blocks are empty), 6175x, 996x, and 33334x which uses naive partitioning method. In this way, workloads are balanced in GraphH. Although there are also many other partitioning methods, like METIS [22], we use IMIB to minimize preprocessing overhead. The time complexity of IMIB (as well as the dividing consecutive vertices into an interval) is $O(m)$ because we only need to scan all edges without extra calculations (e.g., We do not need to get the degree of each edge to perform partitioning scheme based on the number of edges in a partition like Gemini [4]).

[1]Some vertex indexes do not appear in the original raw edge list of a graph like v_8 in this example.

Algorithm 2 Pseudo-code of updating I_x in Cube$_x$

Input: $G = (V, E)$, initialization condition
Output: Updated V

```
1: for each iteration do
2:   for each Cube_y do
3:     receive I_y from Cube_y //Transferring Phase
4:     update I_x using I_y and B_{y.x} in parallel //Updating Phase
5:   end for
6: end for
```

C. Round Interval Pair Scheduling

From the inner `for` loop in Algorithm 2 in Section IV-A we can see that interval data update in each cube can be

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2821565, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

Algorithm 3 Pseudo-code of processing in GraphH

Input: $G = (V, E)$, initialization condition
Output: Updated V

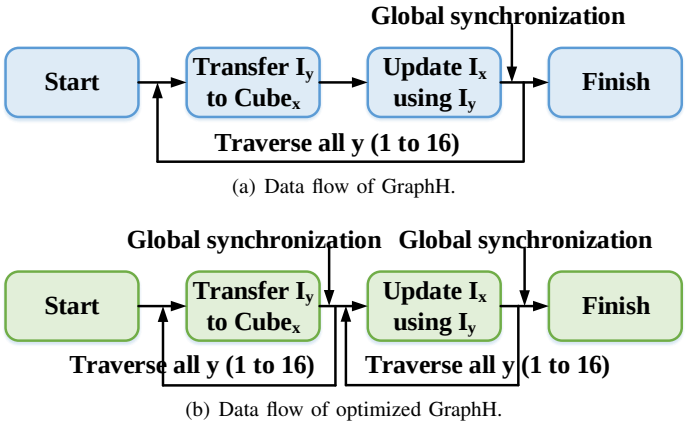
```
1: HMC_Allocate(V) & HMC_Initialize(V)
2: while (finished = false) do
3:   for each round do
4:     for each interval pair < I_x, I_y > (all interval pairs in parallel) do
5:       transfer I_y to Cube_x (update I_y in parallel)
6:       for each edge e ∈ a vault in Cube_x (all vaults in parallel) do
7:         value(e_dst)=HMC_Update(e_dst, value(e_src))
8:       end for
9:       HMC_Intracube_Barrier(I_x)
10:    end for
11:    HMC_Intercube_Barrier()
12:  end for
13:  finished = HMC_Check(V)
14: end while
```

divided into *Transferring Phase* and *Updating Phase*. During the *Transferring Phase*, a cube receives an interval from another cube. Since intervals of different cubes are disjointed, the procedure can be parallelized as follows. The 16 intervals in our GraphH implementation are organized as 8 disjoint interval pairs. Two intervals in a pair send local interval data to another cube. Thus, updating of 8 interval pairs can be executed in parallel. The operation is denoted as one *round*. A complete outer `for` loop iteration in Algorithm 2 needs 16 such rounds. Pair configurations are updated in each round so that any cube has been paired with all other cubes once when one iteration finishes.

Fig. 6(a) shows an example solution of interval pairs in 16 rounds. Round 0 refers to the round that each interval updates its value. Note that this is not the only pair configuration. Based on this scheduling scheme, an interconnection reconfiguration scheme is a prerequisite to implementing such design. We adopt the dynamic interconnection scheme instead of static ones. The host processor controls switch statuses of all meta-connections (Fig. 3). GraphH can easily configure the interconnections in all 16 rounds according to a pre-stored LUT. The targets of our interconnection scheme are: (1) implementing the Round Interval Pair scheduling scheme; (2) avoiding two or more cube pairs using the same meta-connection to transfer data. Under the double-mesh structure, Fig. 6(b) shows the interconnection implementation of the example solution in Fig. 6(a). Eight individual connections are configured to build up direct data paths for interval pairs (labeled in eight different colors) except Round 0. In this way, conflicts among cubes are eliminated because a cube is updated by only one cube at one time, using an exclusive data path with 480 GB/s bandwidth.

D. Programming Interface

Algorithm 3 shows the execution flow of GraphH. Only **HMC_Initialize()** and **HMC_Update()** need to be defined by users using high-level languages.



(a) Data flow of GraphH.

(b) Data flow of optimized GraphH.

Fig. 7. Difference in optimized GraphH.

HMC_Allocate(). The host processor allocates graph data to vaults and cubes according to IMIB.

HMC_Initialize(). This function initializes the value of graph data.

HMC_Update(). This function defines how a source vertex updates a destination vertex using an edge. We can consider **HMC_Update()** function as a particular case of the **Update()** function in Algorithm 1.

HMC_Intracube_Barrier(). This is the synchronization function in a cube. GraphH synchronizes 32 vaults to ensure all vaults have finished updating.

HMC_Intercube_Barrier(). This is the synchronization function for all interval pairs.

HMC_Check(). This function checks if the terminal condition is satisfied.

V. GraphH Optimization

A. Reduce Synchronization Overhead

As mentioned in Algorithm 3, all cubes need to be synchronized 16 times in an iteration (per *Updating Phase* and *Transferring Phase*). Such synchronization operations suffer from unbalanced workloads of different cubes because the execution time of processing a block in a cube is relative to the graph dataset. However, the time of transferring data among cubes is easy to balance because we set each interval to the same size in GraphH. Thus, instead of receiving interval data from other cubes in each round, a vault can store the value of all vertices thus all cubes can perform the updating simultaneously without transferring data among cubes.

TABLE II
Reduce Synchronization Overhead

	Original	Optimized
Sync. (*Transfer*)	16 times	16 times
Sync. (*Update*)	16 times	once
Space (in a vault)	an interval	all intervals

Fig. 7 shows the difference between original scheduling scheme and scheduling after reducing synchronization overhead. In *Updating Phase*, each cube doesn't need to communicate with others before finishing updating. After finishing processing using all vertices value, all vaults receive the updated

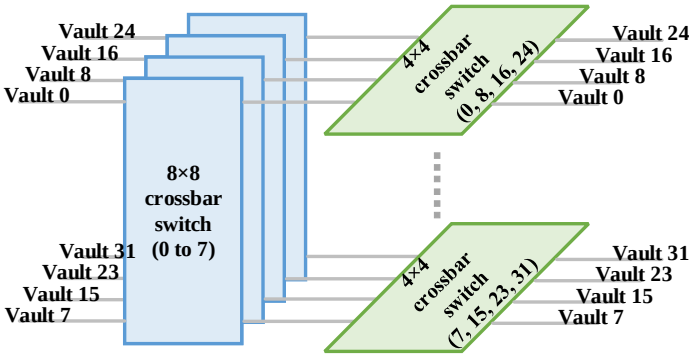


Fig. 8. Crossbar switches to reuse vertices in the logic layer.

value of all vertices from other vaults in *Transferring Phase*. In this phase, we can also use the RIP scheduling scheme shown in Fig. 6, each cube only receives updated vertices value without updating. Table II compares synchronization times and space requirement in a vault of two scheduling methods. Because all vertices need to be stored in a vault, some large graph may not adopt this method due to memory space limitation in a vault. We will solve this issue in Section V-B.

B. Reuse Data in the Logic Layer

In order to update different destination vertices in a cube in parallel, all vaults need to store a complete copy of the source interval. Such implementation has two drawbacks: (1) The memory space of a vault is limited (8GB÷32=256MB[16]), which may be not sufficient to store all intervals in our first optimization method (Section V-A); (2) All source vertices need to be loaded from memory layers to the logic layer 32 times (1 time by each vault).

In order to overcome two disadvantages mentioned above, we adopt four 8×8 crossbar switches followed by eight 4×4 crossbar switches running at 1 GHz (same as Graphicionado [2]) with standard virtual output queues [23] to share source vertices value of 32 vaults in GraphH. Source vertices from any vault can be sent to destination interval buffers in any vault in this way. Assuming that we use 8 Bytes to store an edge, the maximum throughput of DRAM in a vault is 10 GB/s÷8 Bytes = 1.25 edges/ns. Thus, the throughput of DRAMs matches that of crossbar switches. Moreover, because we do not need to modify the value of source vertices under our processing model, all crossbar switches can be pipelined without suffering from hazards (no forwarding units need to be adopted in the design). In this way, instead of duplicating intervals among vaults in a cube, GraphH can share source vertices among vaults. Such data reuse architecture is shown in Fig. 8.

VI. Experimental Evaluation

In this section, we first introduce the simulation setup of our GraphH design, followed by the workloads of experiments used in this section, including the graphs and algorithms.

A. Simulation Setup

All experiments are based on our in-house simulator. Trace files of graph data access patterns are first generated based on DynamoRIO [24]. Then, we apply these traces to timing model generated by Cacti 6.5 [18] and DRAMsim2 [25] to get the execution time. The overhead of reconfiguring the double mesh connection by the host has been taken into consideration in the simulator, detailed in Section III-C. We run all our experiments on a personal computer equipped with a hexa-core Intel i7 CPU running at 3.3GHz. The bandwidth of each vault is set to 10 GB/s according to HMC 2.1 specification (16 GB/s in Tesseract). On the logic layer in a cube, We implement eight 4 MB shared source vertex buffer running at 4 GHz and thirty-two 1 MB individual destination vertex buffer running at 2 GHz to perform the write-after-read operation. We use ARM Cortex-A5 with FPU (without cache) running at 1 GHz as a demo of the in-order core.

B. Workloads

Algorithms. We implement three different graph algorithms. Breadth-First Search (BFS) calculates the shortest path from a given root vertex to all other vertices in the graph. PageRank (PR) evaluates the importance of all websites in a network according to the importance of their neighbor websites. Connected Components (CC) detects all subgraphs in an arbitrary graph. The number of iterations for PR is set to 10 in our simulation, while for other two algorithms the number of iterations depends on the graph data thus we simulate them to convergence.

TABLE III
Properties of Benchmarks

Benchmarks	# Vertices	# Edges
as-skitter (AS) [8]	1.69 million	11.1 million
live-journal (LJ) [8]	4.85 million	69.0 million
twitter-2010 (TW) [8]	41.7 million	1470 million
yahoo-web (YH) [8]	720 million	6640 million
delaunay_n20 (D20) [26]	1.05 million	6.29 million
delaunay_n21 (D21) [26]	2.10 million	12.6 million
delaunay_n22 (D22) [26]	4.19 million	25.2 million
delaunay_n23 (D23) [26]	8.39 million	50.3 million
delaunay_n24 (D24) [26]	16.8 million	101 million

Graphs. Both natural graphs and synthetic graphs are used in our experiments. We conduct four natural graphs including an Internet topology graph *as-skitter* (AS) from trace routes run daily in 2005, *live-journal* (LJ) from the LiveJournal network, *twitter-2010* (TW) from the Twitter social network and *yahoo-web* (YH) from the Yahoo network which consists of billions of vertices and edges. We also conduct a set of synthetic graphs, *delaunay_n20* to *delaunay_n24*, to evaluate the scalability of GraphH. The properties of these graph benchmarks mentioned above are shown in Table III[2].

C. Benefits of GraphH Designs

Compared with Tesseract [1], four techniques architecture (OVB, RDMC) and algorithm (IMIB, RIP) perspectives are

[2]Indexes of vertices have been compressed. Thus, the number of vertices may not equal to the largest vertex index or number of vertices appeared in other papers.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2018.2821565, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

introduced in GraphH. To show the performance improvement by adopting these designs, we simulate the performance difference with/without these techniques in this section. To control the single variable, we adopt all other techniques when simulating the influence of one technique in this section. Two optimization methods in Section V are also adopted in this section.

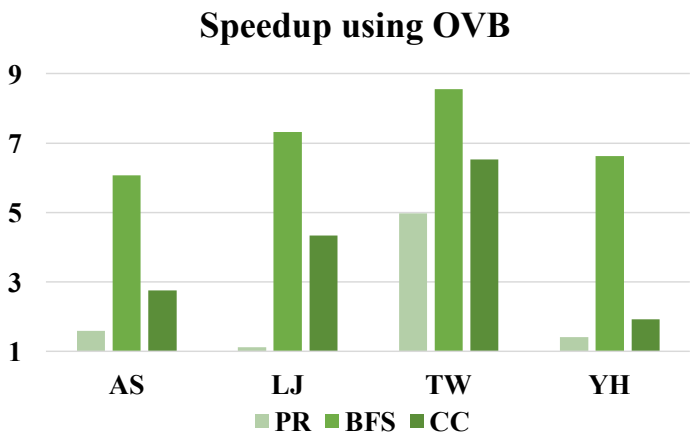


Fig. 9. Speedup using On-chip Vertex Buffer.

1) Benefits of OVB: GraphH adopts on-chip vertices buffer to avoid random access pattern to DRAM layers. We compare the performance of implementing OVB in the logic layer with the performance of directly accessing DRAM layers. The size of source/destination interval buffers is set to 1MB per vault. Techniques like crossbar switches and shared source interval buffers are adopted.

As we can see in Fig. 9. By implementing OVB in the logic layer, GraphH achieves 4.58x speedup compared with directly accessing DRAM layers on average.

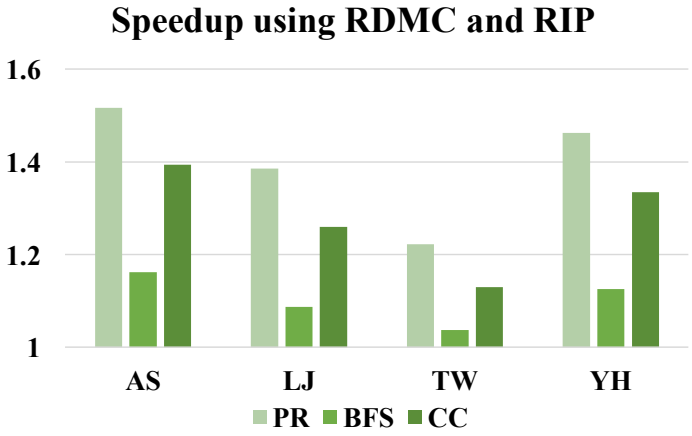


Fig. 10. Speedup using Reconfigurable Double-Mesh Connection/Round Interval Pair.

2) Benefits of RDMC and RIP: GraphH adopts reconfigurable double-mesh connection (RDMC) to maximize the interconnection bandwidth between two cubes. Moreover, to avoid conflicts among cubes, the round interval pair (RIP) scheduling scheme is implemented on RDMC. RDMC and RIP work together in GraphH. We compare the performance using RDMC+RIP with a static single mesh connection network under RIP-like routing scheme. The physical bandwidth of each connection in such static mesh network is also set to 480 GB/s, but each cube can only share a quarter of that bandwidth because only one out of four external links is connected to one meta-connection.

Using RDMC+RIP based connection and scheduling scheme, GraphH achieves 1.29x speedup compared with the static connection method on average.

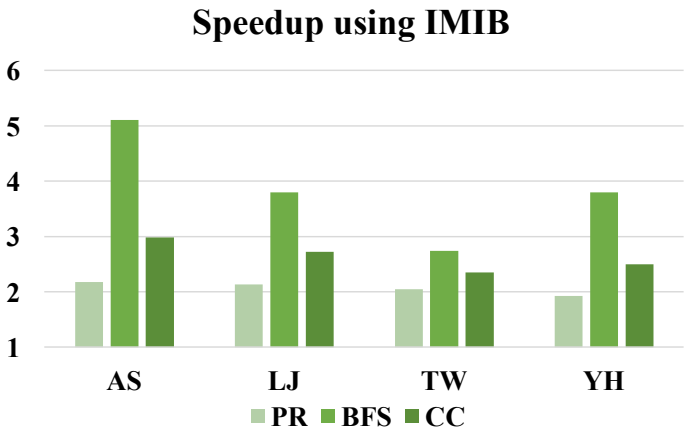


Fig. 11. Speedup using Index Mapping Interval-Block partitioning method.

3) Benefits of IMIB: Workloads of different cubes in a round can be balanced using our IMIB partitioning methods. We compare the performance of using IMIB with chunk-based method [4] (dividing vertices with consecutive indexes into a partition). These two partitioning methods introduce least preprocessing overhead (scanning all edges without other linear algebra operations).

As we can see from Fig. 11, using IMIB achieves 3.05x speedup compared with chunk-based partitioning on average. Such conclusion is in contrast to the conclusion in Gemini [4], which concludes that hash-based partitioning leads to more network traffic and worse performance. The reason is that the interconnection bandwidth in GraphH provided by HMC is two orders of magnitude as that in Gemini, thus balancing workloads is more important in this situation. We will discuss the influence of network bandwidth on the whole system performance in Section VI-E3.

D. Performance

Based on experimental results in Section VI-C, both architecture (OVB, RDMC) and algorithm (IMIB, RIP) techniques in GraphH lead to performance profits. We implement these techniques and evaluate the performance of systems under different configurations.

- **DDR-based system.** We run software graph processing code on a physical machine. The CPU is an i7-5820K core and the bandwidth of DDR memory is 12.8 GB/s.
- **Original-GraphH system.** This is the GraphH system without using optimization methods.

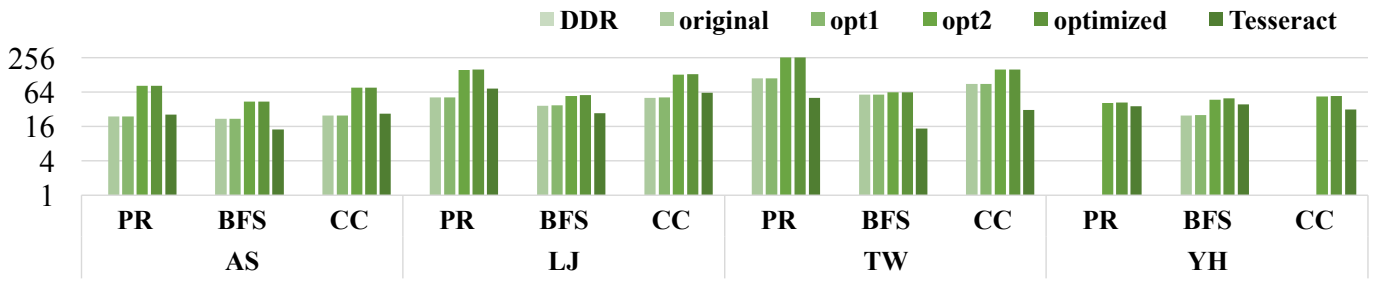


Fig. 12. Performance comparison among DDR-based system, GraphH under different configurations, and Tesseract (normalized to DDR-based system, thus bars of DDR are omitted).

- **Opt1-GraphH system.** We reduce synchronization overheads using optimization method in Section V-A.
- **Opt2-GraphH system.** We reuse vertex data in the logic layer using optimization method in Section V-B.
- **Optimized-GraphH system.** Based on two optimization methods in Section V, this is the combination of Opt1/Opt2-GraphH system.
- **Tesseract system.** We do some modifications in GraphH to simulate Tesseract [1], including: (1) The size of the prefetcher in Tesseract is set to 16 KB; (2) We assume no global access latency but the external bandwidth is set to 160 GB/s (each cube is linked to 3 cubes in Tesseract); (3) Workloads are balanced in Tesseract. In this way, we can simulate the performance upper bound of Tesseract.

1) Performance Comparison: We compare the performance of GraphH under different configurations with both DDR-based and Tesseract system. The comparison result is shown in Fig. 12. As we can see, both Tesseract and GraphH outperforms the DDR-based system by one to two orders of magnitude. GraphH achieves 1.16x $\sim$ 5.12x (2.85x on average) speedup compared with Tesseract because GraphH introduces both architecture and algorithm designs to solve challenges in graph processing.

Reducing cube synchronization overhead has limited contribution to GraphH performance, because workloads of different cubes have been balanced using IMIB. Moreover, such implementation needs to store all vertices value in a vault. Thus, it may not apply to larger graphs (e.g., PR/CC on YH, there is no bar for original and opt1). Reusing on-chip vertices data (opt2/optimized) leads to 2.28x average performance improvement compared with original/opt1 configuration, because such implementation leads to fewer data transferring between OVBs and DRAM layers (Detailed in Section VI-D2).

2) Execution Time Breakdown: Fig. 13 shows the execution time breakdown in GraphH. Note that we assume the memory space of one vault is enough to store required data, thus for larger graphs like YH, we can get results of original/opt1 configurations.

As we can see, loading vertices (including writing back updated vertices data) accounts for 73.99% of total execution time under original/opt1 configurations. By adopting shared source interval memories, GraphH significantly reduces the time of transferring vertices between OVBs and DRAM layers to 50.63% for AS, LJ, and TW. For larger graphs like YH, transferring vertices between OVBs and DRAM layers still account for over 86.21% of total execution time. Larger on-chip SRAM can relieve such bottleneck to some extent, but GraphH can only provide 1MB source vertices buffer per vault due to the area limitation.

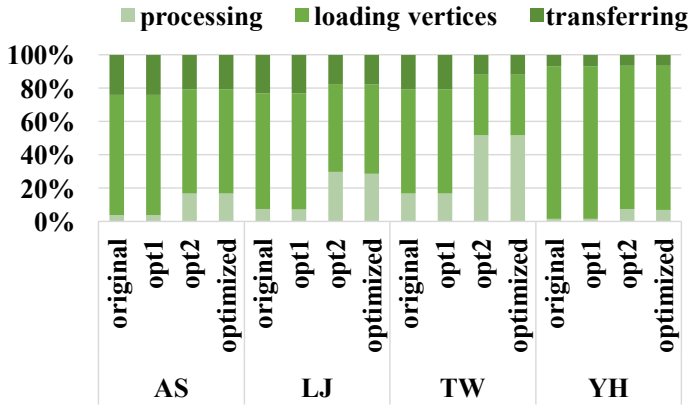


Fig. 13. Execution time breakdown when running PR (processing: processing edges in a cube; loading vertices: transferring vertices between OVBs and DRAM layers; transferring: transferring data among cubes.).

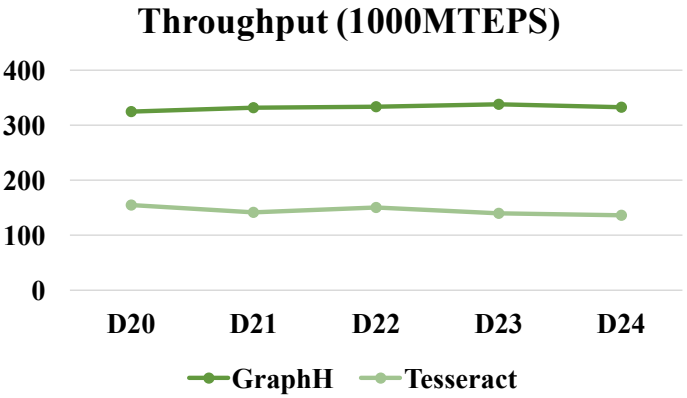


Fig. 14. Scalability running PR on synthetic graphs.

3) Scalability: PIM can achieve *memory-capacity-proportional* bandwidth, so it scales well when processing large-scale problems. We compare the scalability of GraphH with Tesseract. We run PR algorithm on five synthetic graphs,

delaunay_n20 to *delaunay_n24* [26]. We use the Million Traversed Edges Per Second (MTEPS) as the metric to evaluate the scalability of system performance.

Fig. 14 shows the scalability of both GraphH and Tesseract. As we can see, both GraphH and Tesseract scales well to larger graphs. GraphH provides 2.31x throughput than Tesseract on average.

4) Power density: We analyze the power density of GraphH in Fig. 15 using Cacti 6.5 [18] and previous HMC model [15]. Energy consumption of hardware support like OVBs is included in this figure. Logic layer accounts for 58.86% power in GraphH. By introducing extra hardware support for graph processing on HMCs, GraphH consumes more power than Tesseract (94mW/mm^2), but the power density is still under the thermal constraint (133 mW/mm^2) [27] mentioned in Tesseract [1].

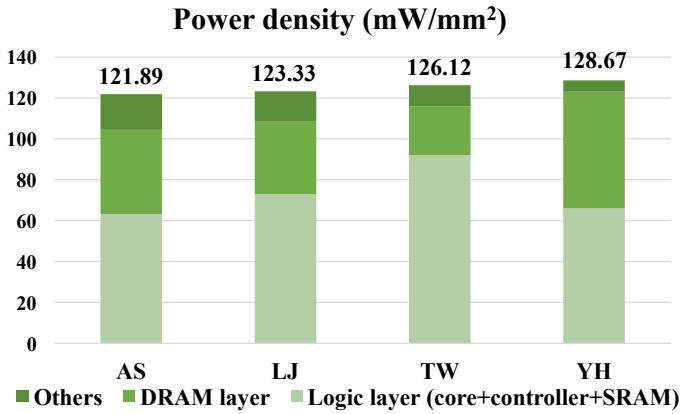


Fig. 15. Power density and its distribution in GraphH when running PR on real-world graphs.

E. Design Space Exploration

1) Different OVB Sizes: We choose 1MB for both source and destination interval buffers per vault in our GraphH implementation. We compare the performance comparison using different OVB sizes in Fig. 16. We run the PR algorithm on four graphs. Note that the size is the source interval buffer size per vault, because the shared source interval buffer is adopted. Considering one goal of HMC is to have a small footprint, we also add the performance of adopting no OVB in Fig. 16. The execution time is normalized to Tesseract.

As we can see in Fig. 16, for small graphs like AS and LJ, SRAMs of 0.25MB per vault are enough to store all vertices on the logic layer. However, for larger graphs, smaller OVB leads to significant performance degradation because more data are transferred between OVBs and DRAM layers. In some situations (e.g., YH), small OVBs cannot bring benefits compared with the no OVB configuration and Tesseract. As we can see, GraphH can still achieve 1.40x speedup against Tesseract even without OVBs.

2) Different Cube Array Sizes: We study the influence of the different number of cubes on the performance. We run PR performance of different array size: a 2×2 array (4 cubes),

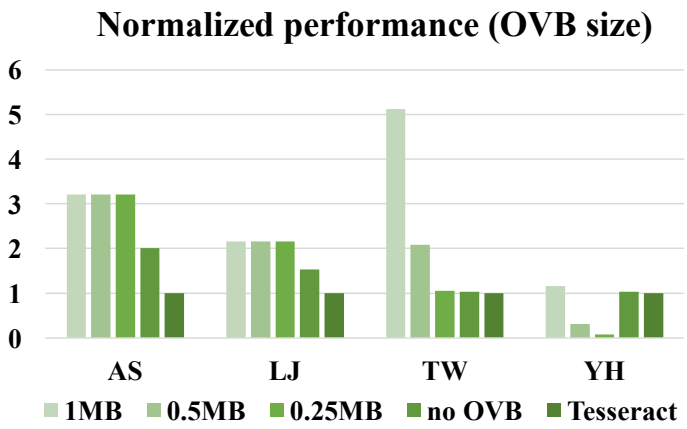


Fig. 16. Speedup using different OVB sizes (Normalized to Tesseract).

a 2×4 array (8 cubes), a 4×8 (32 cubes), and the current 16 cubes' design. The scheduling schemes are akin (e.g., we use Cube 1∼8 to perform Round 0∼7 in Fig. 6(b) as the scheduling scheme of 8 cubes). 32 cubes need a triple-mesh structure to avoid conflicts among cubes.

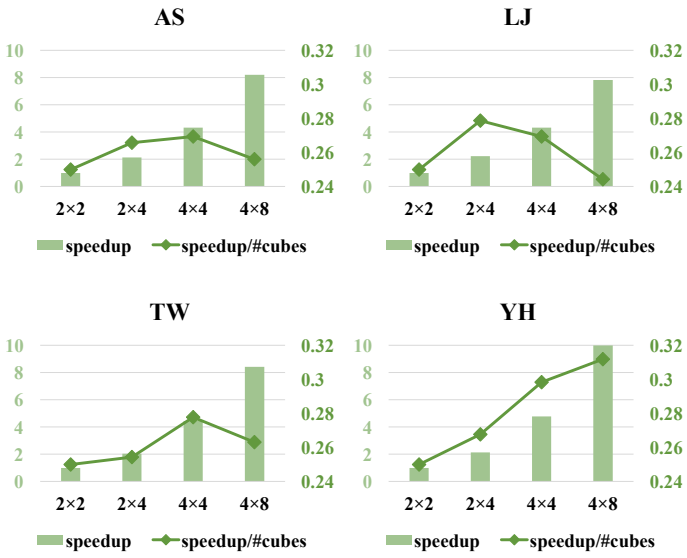


Fig. 17. Performance on different array sizes.

Fig. 17 shows the performance comparison of different array sizes. We normalize the speedup to the performance of the 2×2 array. The axis on the left (light green bar) shows that using more cubes leads to better absolute performance. However, when we normalize the speedup to the number of cubes (right axis, dark green line), we find that speedup per cube is not monotonically increasing. For small graphs like AS and LJ, the OVB of 0.25MB per vault is enough to store all vertices and using more cubes may lead to unbalanced issues. For larger graphs like YH, using more cubes provides larger OVB, thus leads to better performance (normalized to the number of cubes). Simply adding cubes to GraphH can lead to enhancement of absolute performance, but it can also cause inefficient utilization of each cube depending on the size of the graph.

3) Different Network Bandwidth: As mentioned at the end of Section VI-C3, other distributed graph processing systems may also adopt the IMIB partitioning method. However, whether the method works depends on the network bandwidth. Compared with distributed graph processing systems, like Gemini [4], the network bandwidth in GraphH is two orders of magnitude higher. We depict the proportion of transferring data among cube in the total execution time, when the total external bandwidth of a cube varies, in Fig. 18.

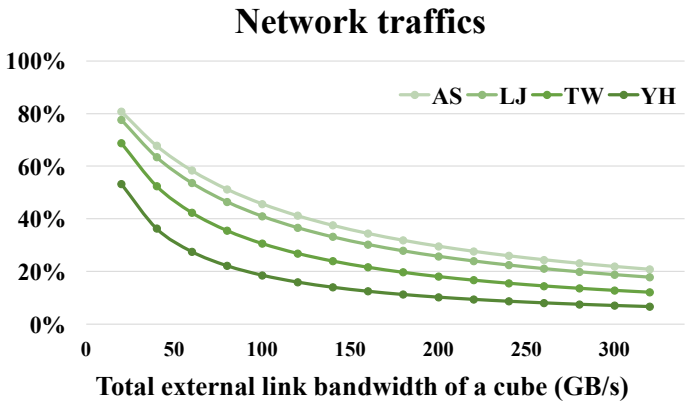


Fig. 18. Proportion of transferring data among cubes in the total execution, when the total external bandwidth of a cube varies.

As we can see in Fig. 18, if we lower the interconnection bandwidth to tens of Gigabytes (same as Gemini [4]), the network traffic accounts for up to 80.79% of the total execution time when network bandwidth is low. Transferring data among cubes using IMIB will domination the total execution time in that situation. While in GraphH, due to high bandwidth among cubes with our designs, the network traffic only accounts for less than 20% of the total execution time, so we balance workloads rather than reduce network traffics in GraphH.

VII. Related work

A. Processing-in-Memory and 3D-Stacking

The concept of Processing-in-memory (PIM) has been proposed since 1990s [28]. The original motivation of PIM is to reduce the cost of moving data and overcome the memory wall. Although adopting caches or improving the off-chip bandwidth are also solutions to such problems, PIM has its advantages like low data fetch cost. One key point in PIM devices is to place massive computation units inside memory dies with high integration density, 3D-stacking technology turns it into reality. In 3D-stacking technology, silicon dies are stacked and interconnected using through-silicon via (TSV) in a package. Intel Corporation presents its first 3D-stacking Pentium 4 processors in 2004 and after that 3D-stacking technology has raised growing attentions. Several work used 3D-stacking PIM architecture to accelerate data-intensive applications, like graph processing [1], [3], neural computation [12], and etc [29], [30].

B. Large-scale Graph Processing Systems

Many large-scale graph processing systems have been developed in recent years. These systems execute on different platforms, including distributed systems [4], [5], [7], [11], single machine systems [8], [14], [31], heterogeneous systems [32]–[35], etc [2]. Some distributed systems are based on big data processing framework [36], [37], they focus on the fault tolerance to provide a stable system. Other distributed graph processing systems focus on other issues like graph partitioning and real-time requirement. Single machine systems focus on providing an efficient system under limited resources (like a personal computer). Heterogeneous systems use other devices like GPUs [32] and FPGAs [33]–[35] to accelerate graph computation, while the capacity and bandwidth of these systems may be limited.

C. Large-scale Graph Processing on Hybrid Memory Cubes

Tesseract [1] and GraphPIM [3] are two PIM-based graph processing architecture based on Hybrid Memory Cubes. By first adopting PIM in graph processing, Tesseract [1] it is efficient and scalable to the problem with intense memory bandwidth demands. To fully exploit the potential of PIM on graph processing, GraphH proposes specific designs for graph processing, including hardware support, balancing method, and reconfigurable network, which are not discussed in Tesseract. GraphPIM [3] proposes the solution of offloading instructions in graph processing to HMC devices. Compared with GraphH and Tesseract, GraphPIM does not introduce extra logics in the logic layer of HMCs. However, without the design of the cube's interconnection, HMC in GraphPIM just performs as the substitute for the conventional memory in a graph processing system. On the other hand, GraphH and Tesseract focus on the scalability of using multiple cubes, and providing the solution of offloading operations in whole graph processing flow to HMCs.

VIII. Conclusion

In this paper, we analyze four crucial factors of improving the performance of large-scale graph processing. To provide higher bandwidth, we implement an HMC array-based graph processing system, GraphH, adopting the concept of processing-in-memory (PIM). Hardware specializations like on-chip vertex buffer (OVB) are integrated into GraphH to avoid random data access. Cubes are connected using Reconfigurable Double-Mesh Connection (RDMC) to provide high global bandwidth and ensure locality. We divide large graphs into partitions and then map them to the HMC. We balance workloads of partitions using Index Mapping Interval-Shard (IMIB). Conflicts among cubes are avoided using Round Interval Pair (RIP) scheduling method. Then, we propose two optimization methods to reduce global synchronization overhead and reuse on-chip data to further improve the performance of GraphH. According to our experimental results, GraphH scales to large graphs and outperforms DDR-based graph processing systems by up to two orders of magnitude and achieves up yo 5.12x speedup compared with Tesseract [1].

REFERENCES

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*. ACM, 2015, pp. 105–117.

[2] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *MICRO*. IEEE, 2016, pp. 1–13.

[3] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level pim offloading in graph computing frameworks," in *HPCA*. IEEE, 2017, pp. 457–468.

[4] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *OSDI*. USENIX, 2016, pp. 301–316.

[5] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[6] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *ISCA*. ACM, 2016, pp. 166–177.

[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*. ACM, 2010, pp. 135–146.

[8] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "NXgraph: An efficient graph processing system on a single machine," in *ICDE*. IEEE, 2016, pp. 409–420.

[9] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication," in *ATC*. USENIX, 2017, pp. 195–207.

[10] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *PACT*. IEEE, 2011, pp. 78–88.

[11] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *SIGPLAN Notices*, vol. 50, no. 8. ACM, 2015, pp. 183–193.

[12] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and efficient neural network acceleration with 3d memory," in *ASPLOS*. ACM, 2017, pp. 751–764.

[13] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *ISCA*. IEEE, 2016, pp. 380–392.

[14] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *SOSP*. ACM, 2013, pp. 472–488.

[15] J. Jeddeloh and B. Keeth, "Hybrid memory cube new dram architecture increases density and performance," in *VLSIT*. IEEE, 2012, pp. 87–88.

[16] *Hybrid Memory Cube Specification 2.1*, Nov. 2015, hybrid Memory Cube Consortium, Tech. Rep.

[17] M. G. Smith and S. Emanuel, "Methods of making thru-connections in semiconductor wafers," Sep. 1967, uS Patent 3,343,256.

[18] H. Labs, "Cacti," http://www.hpl.hp.com/research/cacti/.

[19] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "Memory-centric system interconnect design with hybrid memory cubes," in *PACT*. IEEE, 2013, pp. 145–156.

[20] C. Cakir, R. Ho, J. Lexau, and K. Mai, "Modeling and design of high-radix on-chip crossbar switches," in *NOCS*. ACM, 2015, p. 20.

[21] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *SIGCOMM computer communication review*, vol. 29, no. 4. ACM, 1999, pp. 251–262.

[22] G. Karypis and V. Kumar, "METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0," 1995.

[23] Y. Tamir and G. L. Frazier, *High-performance multi-queue buffers for VLSI communications switches*. IEEE Computer Society Press, 1988, vol. 16, no. 2.

[24] "Dynamorio," http://www.dynamorio.org/.

[25] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[26] "10th dimacs implementation challenge," http://www.cc.gatech.edu/dimacs10/index.shtml.

[27] Y. Eckert, N. Jayasena, and G. H. Loh, "Thermal feasibility of die-stacked processing in memory," 2014.

[28] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The terasys massively parallel pim array," *Computer*, vol. 28, no. 4, pp. 23–31, 1995.

[29] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi, and F. Franchetti, "A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing," in *3DIC*. IEEE, 2013, pp. 1–7.

[30] N. Mirzadeh, O. Kocberber, B. Falsafi, and B. Grot, "Sort vs. hash join revisited for near-memory execution," in *ASBD*, no. EPFL-TALK-209111, 2015.

[31] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *ATC*. USENIX, 2015, pp. 375–386.

[32] F. Khorasani, "Scalable simd-efficient graph processing on gpus," in *PACT*. ACM, 2015, pp. 39–50.

[33] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "ForeGraph: Exploring large-scale graph processing on multi-fpga architecture," in *FPGA*. ACM, 2017, pp. 217–226.

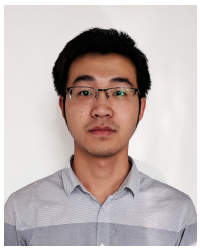
[34] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: Graph processing framework on fpga a case study of breadth-first search," in *FPGA*. ACM, 2016, pp. 105–110.

[35] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "Graphgen: An fpga framework for vertex-centric graph computation," in *FCCM*. IEEE, 2014, pp. 25–28.

[36] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

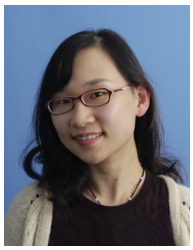
[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster computing with working sets," *HotCloud*, pp. 1–7, 2010.

Guohao Dai (S'18) received his B.S. degree in 2014 from Tsinghua University, Beijing. He is currently pursuing the Ph.D. degree with the Department of Electronic Engineering, Tsinghua University, Beijing. Now, he is visiting University of California, Berkeley. His current research interests include acceleration of large-scale graph processing on hardware and emerging devices.



Tianhao Huang is currently a senior undergraduate majoring in Electronic Engineering of Tsinghua University, Beijing. He joined the Nanoscale Integrated Circuits and Systems (NICS) Lab, Department of Electronic Engineering since 2016. His research interests include architecture support for efficient graph processing and parallel computing.

Yuze Chi received his B.S. degree in electronic engineering from Tsinghua University and started pursuing a Ph.D. degree in computer science in 2016. Being advised by Prof. Jason Cong, Yuze is looking for software/hardware optimization opportunities in many application domains, including graph processing, image processing, and genomics.



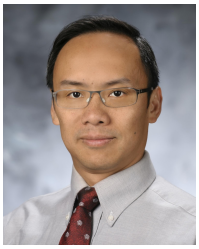
Jishen Zhao (M'10) is an Assistant Professor at Computer Science and Engineering Department of University of California, San Diego. Her research spans and stretches the boundary between computer architecture and system software, with a particular emphasis on memory and storage systems, domain-specific acceleration, and high-performance computing. She is a member of IEEE.

Guangyu Sun (M'09) received his BS and MS degrees from Tsinghua University, Beijing, in 2003 and 2006, respectively and the PhD degree in computer science from Pennsylvania State University, in 2011. He is currently an associate professor of CECA at Peking University, Beijing, China. His research interests include computer architecture, VLSI Design as well as electronic design automation (EDA). He has published more than 60 journals and refereed conference papers in these areas. He serves as an AE of ACM TECS and JETC. He has also served as a peer reviewer and technical referee for several journals, which include IEEE Micro, the IEEE Transactions on Very Large Scale Integration, the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, etc. He is a member of the IEEE, ACM, and CCF.

Yongpan Liu (M'07-SM'15) received his B.S., M.S. and Ph.D. degrees from Electronic Engineering Department, Tsinghua University in 1999, 2002 and 2007. He has been a visiting scholar at Penn State University during summer 2014. He is a key member of Tsinghua-Rohm Research Center and Research Center of Future ICs. He is now an associate professor in Dept. of Electronic Engineering Tsinghua University. His main research interests include nonvolatile computation, low power VLSI design, emerging circuits and systems and design automation. His research is supported by NSFC, 863, 973 Program and Industry Companies such as Huawei, Rohm, Intel and so on. He has published over 60 peer-reviewed conference and journal papers and led over 6 chip design projects for sensing applications, including the first nonvolatile processor (THU1010N) and has received Design Contest Awards from (ISLPED2012, ISLPED2013) and best paper award HPCA2015. He is an IEEE (ACM,IEICE) member and served on several conference technical program committees (DAC, ASP-DAC, ISLPED, A-SSCC, ICCD, VLSI-D).

Yu Wang (S'05-M'07-SM'14) received his B.S. degree in 2002 and Ph.D. degree (with honor) in 2007 from Tsinghua University, Beijing. He is currently a tenured Associate Professor with the Department of Electronic Engineering, Tsinghua University. His research interests include brain inspired computing, application specific hardware computing, parallel circuit analysis, and power/reliability aware system design methodology.

Dr. Wang has published more than 50 journals (38 IEEE/ACM journals) and more than 150 conference papers. He has received Best Paper Award in FPGA 2017, ISVLSI 2012, and Best Poster Award in HEART 2012 with 8 Best Paper Nominations (DAC 2017, ASPDAC 2016, ASPDAC 2014, ASPDAC 2012, 2 in ASPDAC 2010, ISLPED 2009, CODES 2009). He is a recipient of IBM X10 Faculty Award in 2010. He served as TPC chair for ISVLSI 2018, ICFPT 2011 and Finance Chair of ISLPED 2012-2016, Track Chair for DATE 2017-2018 and GLSVLSI 2018, and served as program committee member for leading conferences in these areas, including top EDA conferences such as DAC, DATE, ICCAD, ASP-DAC, and top FPGA conferences such as FPGA and FPT. Currently he serves as Co-Editor-in-Chief for ACM SIGDA E-Newsletter, Associate Editor for IEEE Transactions on CAD, and Journal of Circuits, Systems, and Computers. He also serves as guest editor for Integration, the VLSI Journal and IEEE Transactions on Multi-Scale Computing Systems. He has given 70 invited talks and 2 tutorials in industry/academia. He is now with ACM Distinguished Speaker Program. He is an ACM/IEEE Senior Member. Yu Wang also received The Natural Science Fund for Outstanding Youth Fund in 2016, and is the co-founder of Deephi Tech (valued over 150M USD), which is a leading deep learning processing platform provider.



Yuan Xie (SM'07-F'15) was with IBM, Armonk, NY, USA, from 2002 to 2003, and AMD Research China Lab, Beijing, China, from 2012 to 2013. He has been a Professor with Pennsylvania State University, State College, PA, USA, since 2003. He is currently a Professor with the Electrical and Computer Engineering Department, University of California at Santa Barbara, Santa Barbara, CA, USA. He has been inducted to ISCA/MICRO/HPCA Hall of Fame. His current research interests include computer architecture, Electronic Design Automation, and VLSI design.

Huazhong Yang (M'97-SM'00) was born in Ziyang, Sichuan Province, P.R.China, on Aug.18, 1967. He received B.S. degree in microelectronics in 1989, M.S. and Ph.D. degree in electronic engineering in 1993 and 1998, respectively, all from Tsinghua University, Beijing.

In 1993, he joined the Department of Electronic Engineering, Tsinghua University, Beijing, where he has been a Full Professor since 1998. Dr. Yang was awarded the Distinguished Young Researcher by NSFC in 2000 and Cheung Kong Scholar by Ministry of Education of China in 2012. He has been in charge of several projects, including projects sponsored by the national science and technology major project, 863 program, NSFC, 9th five-year national program and several international research projects. Dr. Yang has authored and co-authored over 400 technical papers, 7 books, and over 100 granted Chinese patents. His current research interests include wireless sensor networks, data converters, energy-harvesting circuits, nonvolatile processors, and brain inspired computing. Dr. Yang has also served as the chair of Northern China ACM SIGDA Chapter.