GraphIA: An In-situ Accelerator for Large-scale Graph Processing

Gushu Li* , Guohao Dai**, Shuangchen Li*, Yu Wang**, Yuan Xie*
*Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, 93106, USA
**Department of Electronic Engineering, BNRist, Tsinghua University, Beijing, 100084, China
{gushuli,yuanxie}@ece.ucsb.edu,dgh14@mails.tsinghua.edu.cn

ABSTRACT

Graph processing is widely used in various domains, while processing large-scale graphs has always been memory-bound. In-situ processing is a promising solution to overcome the "memory wall" challenges in such memory-intensive applications. Previous accelerator designs for graph processing only focused on integrating more computing units inside memories or using more memory layers, rather than exploiting the huge parallelism lying in memory banks. In this paper, we present GraphIA, an **I**n-situ **A**ccelerator for large-scale graph processing based on DRAM technology. GraphIA couples large-capacity memory and computing resource in DRAM by connecting multiple chips with computation circuits inside. GraphIA chips are organized into a scaling ring interconnection, which is able to maximize the individual bandwidth with minimal connection overheads and scale to larger graphs by using more chips. Banks in DRAM are organized into heterogeneous edge and vertex banks, cooperating with customized peripheral circuits. Data duplication and scheduling schemes in heterogeneous banks are further introduced to overcome the performance loss caused by the irregular local and remote memory access in our multi-chip ring structure, achieving 1.63× and 1.16× speedup respectively. According to our extensive experiments, by adopting GraphIA design, our in-situ accelerator achieves 217× speedup CPU-DRAM designs.

CCS CONCEPTS

• **Computer systems organization** → *Processors and memory architectures*; • **Hardware** → **Memory and dense storage**;

KEYWORDS

Large-scale Graph Processing, DRAM, In-situ Accelerator

ACM Reference Format:
Gushu Li* , Guohao Dai**, Shuangchen Li*, Yu Wang**, Yuan Xie*. 2018. GraphIA: An In-situ Accelerator for Large-scale Graph Processing . In *The International Symposium on Memory Systems (MEMSYS), October 1–4, 2018, Old Town Alexandria, VA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3240302.3240312

1 INTRODUCTION

The explosive data analysis requirements for large-scale data and relationships facilitate the evolving of both algorithms and data models. Graph, a kind of data structures which can naturally represent both data and relationships, has been widely used in various domains, including social network analysis, user behavior recommendation, etc. With the coming of "big-data" era, the size of graphs continues to scale and it has been even more challenging to achieve high graph processing performance on general purposed architectures. Many customized graph processing accelerators [1, 5–7, 11, 14, 23, 24, 26, 27] have been put forward.

The essential way to improve the performance of large-scale graph processing and overcome the "memory wall" for big data problems is to provide a high bandwidth of data access. Thus, these graph processing accelerators usually integrated multiple processing units "closer" to the memory, including using on-chip eDRAM [11], on-chip SRAM [6, 23], emerging 3D-stacked memory [1, 7, 27], etc. Such designs can achieve magnitudes of speedup against graph processing systems on conventional architectures [4, 10, 18, 21, 22, 25, 28–30].

However, integrating processing units "closer" to the memory still treats the memory as a whole storage part, without exploring the parallelism and bandwidth inside the memory. Recently, in-situ processing [8, 20] has been put forward to tackle this problem. For example, DRISA [20] modified the DRAM into a reconfigurable in-situ accelerator, and achieved 8.8× speedup against ASIC based accelerator design [3] on neural network computation [9, 17]. The main idea of in-situ accelerators is to modify circuits in the memory to perform computation functions, rather than simply putting computation units "closer" to the whole memory part.

The idea of in-situ accelerators can be a promising solution to accelerating large-scale graph processing by utilizing parallelism inside the memory. However, using in-situ computation for large-scale graph processing faces two problems: (1) Scaling to larger graphs is hard due to the poor locality when real-word graphs (e.g., Table 1) requires to be stored in multiple chips. (2) Intensive irregular memory access introduces significant overhead. Graph processing is notoriously known for the intensive irregular memory access. Processing data in the memory chip may be interfered by other processing units, leading to efficient bandwidth loss.

In this paper, we present GraphIA, to tackle these two challenges for large-scale graph processing accelerators. GraphIA follows the approach of Automata [8] and DRISA [20], which use the DRAM technology to build an in-situ accelerator with large-capacity memory and computing resource highly coupled.We propose the **scaling ring** interconnection topology and communication scheme for GraphIA chips to make the accelerator scalable to larger graphs.

For each GraphIA chip, we modify the DRAM chip by designing **heterogeneous banks** with peripheral circuits in the DRAM to fully exploit the huge parallelism lying in memory banks. Moreover, GraphIA design can be easily applied to emerging devices (e.g., HMC [1, 7, 27]) by adopting same designs in memory banks and interconnections. The main contributions of GraphIA are summarized as follows:

- We propose the first in-situ architecture, GraphIA, for large-scale graph processing, which highly couples computing elements and large capacity DRAM memory. Customized circuits are integrated into DRAM to reduce the latency for graph data access.
- We propose the **scaling ring** interconnection together with our data scheduling scheme to deal with the scaling out problem facing by larger graph problems. GraphIA chips are connected using a ring structured interconnection. In this way, GraphIA can scale to larger graphs by using more chips without significant performance loss.
- We design **heterogeneous banks** with **data duplication** and **scheduling** schemes to further leverage the intrinsic parallelism in DRAM banks and overcome the performance loss caused by the irregular local and remote memory access in our multi-chip ring structure. Simulation result shows that data duplication and scheduling can achieve 1.63× and 1.16× speedup on average.
- We evaluate the performance of GraphIA through comprehensive experiments, and the results show that GraphIA outperforms CPU-DRAM based designs by 217×.

2 BACKGROUND

In this section, we will introduce the background of graph processing, including two processing models and the partitioning models.

2.1 Graph Processing Model

Gather-Apply-Scatter (GAS) model is widely used to represent various graph algorithms. There are two main ways to implement the GAS model, vertex-centric model and edge-centric model:

- **Vertex-centric model.** For each source vertex, it updates all destination vertices of its outgoing edges.
- **Edge-centric model.** For each edge, the source vertex updates the destination vertex.

The edge-centric model is proposed in X-stream [25]. Algorithm 1 shows the pseudo-code of the edge-centric model. When processing an edge $e_{i,j}$ from source vertex v_i to destination vertex v_j, different graph algorithms only differ in the "Update()" function. By restrict the range of both source and destination vertices, the edge-centric model can and to ensure the locality for data access. Thus, we adopt this edge-centric model in our GraphIA design.

Algorithm 1 Pseudo-code of the edge-centric model [25]

```
Require: $G = (V, E)$, initialization
Ensure: Updated $V$
1: for each $v_i \in V$ do
2:     Initialize($v_i$, initialization)
3: end for
4: while not finished do
5:     for each $e_{i.j} \in G$ do
6:         $value(v_i)$ = Update($v_i$, $v_j$, $e_{i,j}$)
7:     end for
8: end while
9: return $V$
```

2.2 Graph Partitioning Model

In order to ensure the locality and fit the graph into the memory, graph partitioning methods are required. The interval-block partitioning method is widely adopted in previous systems [4, 29], in which all vertices are divided into P disjointed intervals, and then edges are divided into P^2 disjointed blocks according to source and destination vertices. We show an example of the interval-block partitioning method in Figure 1(a).

Based on the edge-centric processing model and the interval-block partitioning model, different graph algorithms can be executed in the form of iterations. During each iteration, all blocks are sequentially processed, and edges within each block are sequentially accessed to update the destination vertices.

3 GRAPHIA ARCHITECTURE

In this section, we introduce our GraphIA architecture and the entire processing flow in detail. In general, our architecture contains N GraphIA chips. The vertex and edge data are divided into N partitions and distributed equally to all N chips. We will first introduce our graph partitioning method, followed by the details of our chip architecture. Then we will show how GraphIA accelerates graph computation, optimizes memory access, and hides the long latency of inter-chip data transfer in the processing flow.

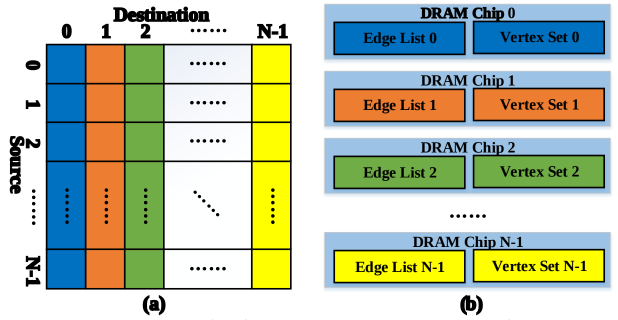


Figure 1: Vertex and edge partitioning in GraphIA, one interval and N blocks are stored in a GraphIA chip.

3.1 Data Partitioning and Allocation

Real world graphs can be very large (millions of vertex and edges, as shown in Table 1) and require multiple DRAM chips to store the entire graph. More importantly, we hope to enable parallel graph processing on multiple chips to make our design scalable to larger problems. As a result, we adopt the interval-block partitioning method mentioned in Section 2.2. In our design, we consider more about workload balance in this graph partitioning phase while the memory access optimization will be discussed later in processing flow introduction. In order to balance the workloads of each GraphIA chip, we use hash-based method [7] to partition the vertices into N intervals and then divide the edges into N^2 blocks.

We reorder the vertices after partitioning and Figure 1(a) shows the result. The adjacent matrix is divided into N columns marked by different colors. Each column contains approximately the same amount of vertices and edges since we partition the vertices using hash function [7]. The destination vertices and the edges in one column are mapped onto one GraphIA chip (as shown in Figure 1(b)). Each chip contains one vertex interval's data and N edge blocks' data. Now our architecture is similar to a distributed graph processing system while one node in this "distributed system" is a DRAM chip rather than a processor.

3.2 Overall Architecture

Figure 2 shows the entire architecture of GraphIA . The left side shows the N GraphIA chips (modified DRAM chips) connected in a ring topology. The right side shows the inside of one chip. We use some memory banks to store edge data and vertex data. We also add some peripheral circuits to enable inner- and inter-chip communication. Our data layout and processing flow design could provide high parallelism. All chips process edges in parallel and there are simple ALUs (sALU) for basic arithmetic/logical operations on each chip. But such high parallelism requires high memory bandwidth support. In GraphIA , the memory access is also optimized for edge data access, vertex data access, and inter-chip data transfer. For edge data access and inter-chip data transfer, the memory accesses are serialized to increase row buffer hit rate. For the random vertex access in the computation, we use multiple banks and hashed vertex data layout to provide a bank-level parallelism.

Details of each component and peripheral circuits are introduced in the following sections.

3.2.1 Scaling Ring Connection: All the N chips are connected in a ring topology. During the execution, each chip will only need to transfer some vertex data to the neighbor chip and there is no data traffic conflict on the N connections. This connection makes our GraphIA scalable since we can increase the number of chips to store a larger graph and obtain more chip-level parallelism.

3.2.2 Heterogeneous Banks. The memory banks in the GraphIA are organized as edge and vertex banks. The data duplication and scheduling schemes will be introduced with the processing flow.

- **Edge Bank:** The edge data of each partition are stored in the edge banks. The edges are first sorted by the partition number of the source vertex and then sorted by the index of destination vertex. This data layout and the sequential edge-centric execution guarantee high row buffer hit rate because once a row is activated, all the data in this row will be read.
- **Vertex Bank and Controller:** The Vertex Banks are divided into 4 groups. At the different time during the execution, different groups have different functions. In general, two groups will be used to support computation. One of them stores the updated destination vertex value after computation and the other one stores the source vertex value for the computation. These two vertex bank groups will receive the vertex info of the edges from the FIFOs, read the requested data from the vertex banks, and send the data to computing components. Two remaining groups will store the vertex data transferred from the previous chip. The function of the vertex banks will change in different phases of the execution. A detailed explanation of how to use these 4 vertex bank groups is in the following section.

3.2.3 Peripheral Circuits & Computing Components. We employ a scheduler to fetch edge data from edge banks sequentially and then put the vertex data requests in FIFOs. Different FIFOs contain the addresses of vertices in different banks inside a bank group. The vertex data are stored across all the banks inside a group so that the bank-level parallelism is fully enabled during vertex data access. The Computing Components module has 4 sALUs that are able to compute 4 edges simultaneously. The destination vertex data will be updated after the computation for each edge in the vertex banks.

3.3 Processing Flow

The processing flow of GraphIA follows the edge-centric model. In each iteration, we will traverse all edges and update the destination vertex data for each edge. As mentioned in section 3.1, all the vertex and edge data are divided into N parts and stored on N DRAM chips. Suppose the N intervals in the vertex set V are represented by $\{I_i, 0 \le i \le N-1\}$ and the N^2 blocks in the edge set E are $\{B_{i.j}, 0 \le i, j \le N-1\}$. Both I_i and $B_{\{0,1,\ldots,N-1\}.i}$ are stored on chip i (shown in Figure 1). This data partitioning guarantees that when we update the destination vertex, we only need to update it locally since all the edges with the same destination will be in the

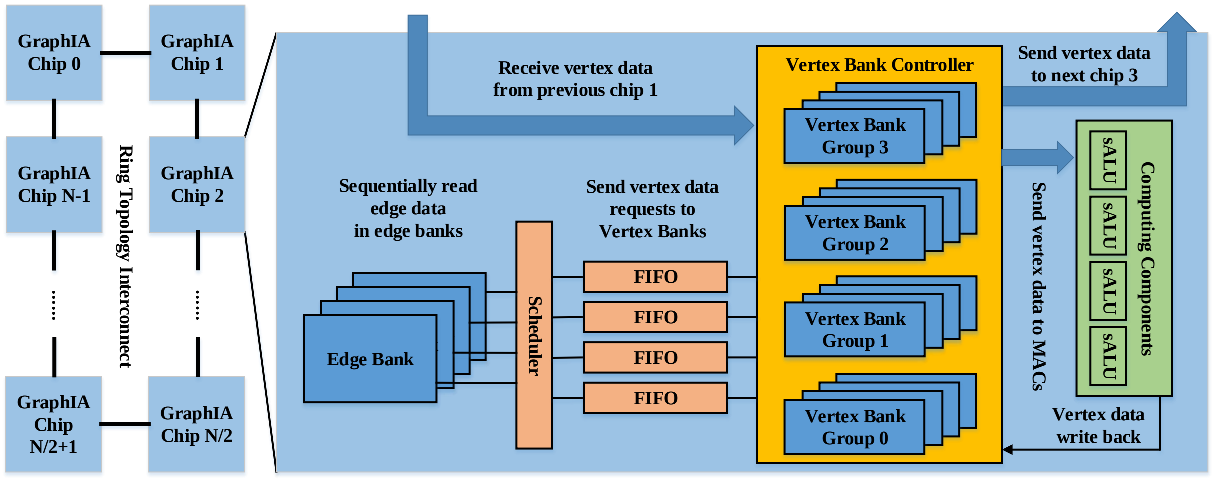
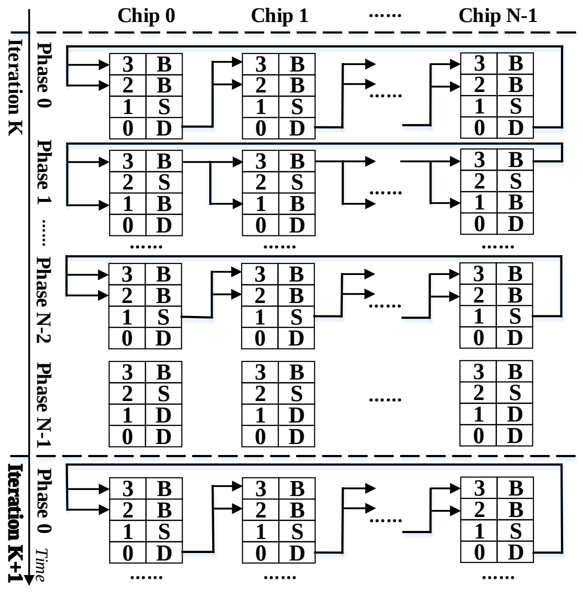


Figure 2: Overall Architecture of GraphIA. The left side shows the ring-structured interconnection of GraphIA chips. The right side shows heterogeneous banks with peripheral circuits inside a GraphIA chip.



D: Store the Destination Vertex Data in currently processed edge block
S: Store the Source Vertices Data in currently processed edge block
B: Buffer the Source Vertices Data transferred from previous chip

Figure 3: Vertex Data Transfer and Processing Flow

same chip as well as the vertex data. Thus we do not need to have a complex protocol to maintain memory coherence.

Even though the destination vertices of all the edges on chip i are limited in I_i, the source vertex set can still be the entire vertex set V under our hash-based vertex partitioning. As a result, chip i can fetch vertex data in I_i from local DRAM banks but need to access $I_j(j \neq i)$ data from DRAM banks on other chips. Such remote memory access suffers long latency which will degrade the system performance.

In order to solve this problem, we further divide each iteration into N phases. In one phase, each chip only computes the edges in one block. A complete processing flow is explained as follows (shown in Figure 3):

(1) Before an iteration K, each interval I_i is duplicated and stored in two Vertex Bank Groups, group 0 and group 1 on chip i.
(2) In iteration K phase 0, chip i will compute the block $B_{i.i}$. Data in Vertex Bank Group 1 will be used as the source vertex set, and Vertex Bank Group 0 will be used as the destination vertex set and store the updated vertex data during this iteration.
(3) **Data Duplication and Scheduling happens in this step.** At the same time with step (2) in iteration K phase 0, chip i will also send its vertex data (interval I_i) in Vertex Bank Group 0 to the Vertex Bank Group 2 and 3 of the next chip $i+1$ in the ring. I_i is now duplicated and both Vertex Bank Group 2 and 3 in chip $i+1$ have a copy of I_i.
(4) In iteration K phase 1, chip i now have an updated interval I_i stored in Vertex Bank Group 0 and two copied of I_{i-1} in Vertex Bank Group 2 and 3 transferred from chip $i-1$. Chip i will compute block $B_{i-1.i}$ using data from Vertex Bank Group 2 and 0 without access another chip since I_{i-1} has already been transferred in the last phase.
(5) At the same time with step (4), chip i will also send interval I_{i-1} from its Vertex Bank Group 3 to the Vertex Bank Group 3 and 1 of the next chip $i+1$ since Vertex Bank Group 2 will be used for computation in this phase.
(6) This procedure can repeat until phase $N-1$, which is the last phase in this iteration K. During this phase, chip i will compute block $B_{i+1,i}$ and duplicate the destination vertex set in both Vertex Bank Group 0 and 1. We do not need to transfer data to the next chip in the last phase. Now chip i has done the computation for all blocks $B_{0.i}, B_{1.i}, \ldots, B_{N-1.i}$ and one iteration is completed. We can go back to step 2 to start the next iteration $K+1$.

Figure 3 shows the one entire flow in iteration K. The horizontal axis represents chips and the vertical axis is time. The black arrows represent the data transfer between chips. The letter '**D**', '**S**', and '**B**' represent different states of the Vertex Bank Groups in each chip. '**D**' and '**S**' mean that the Vertex Bank Groups are being used to store the source and destination vertices for the computation respectively, and '**B**' means that it is used as a buffer to store the vertex data transferred from the previous chip.

Memory Access Optimization Analysis: To avoid the performance loss caused by remote memory access, our data scheduling will prefetch the data used in the next phase so that all computation data accesses will happen locally. However, the prefetched data need to be used by the computation and sent to the next chip at the same time. This will introduce bank conflicts which can significantly reduce the performance (discussed in 4.3.2). In our design, all prefetched data will be duplicated so that the computation and data transfer will access different Vertex Bank Groups. For edge banks, since the order of access to the edges is fixed in our processing flow, we can organize the edge data to make the subarray row buffer in the edge banks can always be hit until the next row is activated. For source and destination vertex banks access during computation, we use multiple banks and spread the vertices among all banks to provide bank-level parallelism. For inter-chip data transfer, a bulk of continuous data is transferred so that data in the subarray row buffer can be fully utilized.

4 EVALUATION

In this section, we will introduce our experiment setup and analyze the benefits coming from our design. We will also explore the design space in terms of scalability.

4.1 Experiment Setup

4.1.1 Datasets and Benchmarks. Five real-world graph datasets of different types are selected from SNAP datasets [19] to evaluate the system performance in our experiments, including web-Google (WG, web graph), wiki-Talk (WT, communication network), soc-Pokec (PK, social network), email-EuAll (EA, email network), and cit-Patents (CP, citation network). We also choose another 4 synthetic graphs [13] with different sizes to test the scalability of our design. Table 1 shows the property of the graphs with the number of vertices and edges. Three graph algorithms, PageRank (PR), Breadth-First Search (BFS), and Connected Components (CC), are selected as benchmarks.

4.1.2 Simulation Configuration. We simulate the entire system performance considering both memory access trace and circuits

Table 1: Datasets

Graph Name	No. Vertices	No. Edges	Type
web-Google (WG) [19]	875,713	5,105,039	web
soc-Pokec (PK) [19]	1,632,803	30,622,564	social
email-EuAll (EA) [19]	265,214	420,045	email
wiki-Talk (WT) [19]	2,394,385	5,021,410	communication
cit-Patents (CP) [19]	3,774,768	16,518,948	citation
delaunay_n20 [13]	1,048,576	6,291,372	synthetic
delaunay_n21 [13]	2,097,152	12,582,816	synthetic
delaunay_n22 [13]	4,194,304	25,165,738	synthetic
delaunay_n23 [13]	8,388,608	50,331,568	synthetic

model. The parameters of our DRAM components are estimated by Micron DDR3 SDRAM [15] and SDRAM System-Power Calculator [16]. For Computing Components, we use the data from [12] to obtain the latency and energy for 32-bit adder and multiplier. The single core CPU model is from Intel i7-6700 (Skylake) [2].

We choose $N = 16$ as a default configuration so that we have 16 chips connected in a ring. Each chip has 4 edge banks and 16 vertex banks. 4 vertex banks will be one vertex bank groups so that we can have 4 groups. There are 65536 rows in one bank and each row has 1024 bits. Such that the entire system could contain around 67 million edges in so that all the 5 real-world datasets we used can be placed inside the DRAM. In the scalability test, we will change the value of N to show GraphIA can support larger graph without a significant loss of performance.

4.2 Overall Performance Improvement

To evaluate the performance gained from GraphIA , we simulate the single core off-chip graph processing performance. Figure 4 shows the normalized execution time for the off-chip processing system and our GraphIA achieves 217× speedup on average. This acceleration comes from (1) the huge internal memory bandwidth of the in-situ accelerator and (2) parallel processing supported by the multiple sALUs on multiple chips. Moreover, we improve the memory access locality by graph partitioning, eliminate the bank conflicts by data duplication, and conceal the long remote data access latency by specialized data movement scheduling.

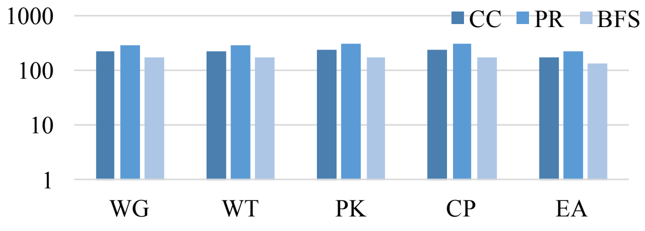


Figure 4: Normalized Off-Chip Execution Time

4.3 Evaluation of Memory Access Optimization

We evaluate the effect of the memory access optimization in GraphIA by comparing with two different designs. The first one is that we do not employ data movement scheduling (**Non-Scheduling**) in the ring architecture so that remote data access can not be overlapped with computation. The second one is that we do not use duplicate the data (**Non-Duplication**) transferred from another chip so that the computation and data transfer will access the same Vertex Bank Group and affect each other.

4.3.1 Benefits of Data Movement Scheduling. Data access onto a remote chip has longer latency and will degrade the entire system performance if we do not overlap the computation and remote data access. Figure 5 shows the normalized execution time without data movement scheduling to hide the long remote data access latency. As we can see, the system performance is improved by 1.13× on average after adopting the data movement scheduling scheme.

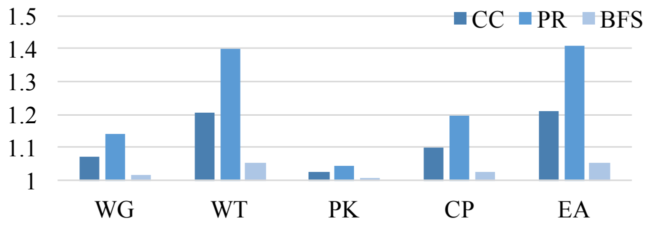


Figure 5: Normalized Execution Time without Scheduling

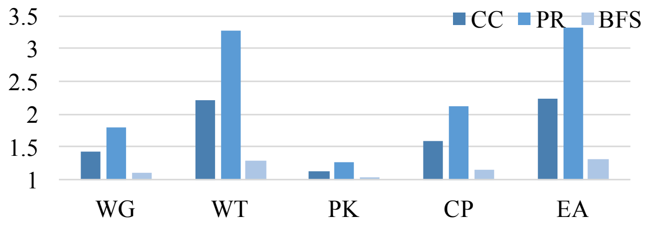


Figure 6: Normalized Execution Time without Duplication

4.3.2 Benefits of Data Duplication. If we do not use data duplication to separate the vertex data access for computation and movement, both of them will be slower due to bank conflict. Figure 6 shows the normalized execution time without data duplication compared with our default configuration, and this is a 1.63× improvement by adopting data duplication in GraphIA on average.

4.4 Scalability Study

We choose 4 synthetic graphs, delaunay_n20, delaunay_n21, delaunay_n22, and delaunay_n23, from Delaunay Graphs [13]. The size of delaunay_n23 is 2× of delaunay_n22, 4× of delaunay_n21, and 8× of delaunay_n20, We use a 2-chip GraphIA configuration to process delaunay_n20 and increase the number of chips based on the size of the graph. Finally, we will use the default 16-chip GraphIA configuration for delaunay_n23. All other memory parameters remain the same. Figure 7 shows normalized execution time for different GraphIA configurations with graphs of different sizes. We can see that there is no significant performance loss as the number of chips increases. This could prove that our GraphIA is able to scale up by increasing the number of chips.

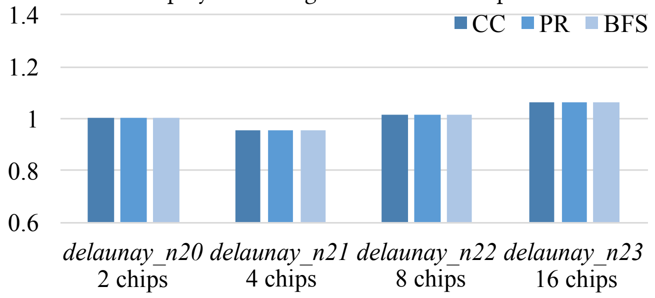


Figure 7: Normalized Memory Access Time

We can further scale up GraphIA by increasing the memory capacity inside one chip or the total number chips. Increasing single memory capacity may not affect the entire data movement scheduling. However, larger memory will cause longer latency. And the largest possible capacity of one chip is limited since we could not have a chip with too much DRAM inside within a specific area.

On the other hand, increasing the number of chips seems to be easier in term of scalability but it will increase the data movement overhead. Suppose we have N chips in our system. The execution time in one phase is proportional to the number of edges in one block, $|B_{i,j}| \approx |E|/N^2$. The data movement time in one phase is proportional to the number of vertices in one interval, $|I_i| \approx |V|/N$. As N increases, the data movement will dominate the total execution time. The inter chip data bandwidth needs to be increased, e.g. using near memory data compression to increase the effective data bandwidth. If the computation time is longer then data movement time, the computation throughput by should be increased, for example, increasing the number of banks to provide higher inner chip data bandwidth and the number of computing components.

5 RELATED WORK

The essential way to improve the performance of graph processing is to provide higher bandwidth for graph data access. One possible solution is adopting processing-in-memory (PIM) designs. By integrating processing units into the memory, PIM can achieve magnitudes of higher bandwidth compared with conventional memory systems, and provide “memory-capacity-proportional” bandwidth.

Many previous works focused on using PIM for large-scale graph processing are based on Hybrid Memory Cube (HMC) devices. HMC is a 3D stack PIM device with multiple DRAM layers for storage connected to a logic layer for computation. Tesseract [1] proposed an HMC array structure to map graph processing flow to it. GraphP [27] and GraphH [7] further improved the performance of Tesseract by carefully designing connections and data allocations among cubes in the array. All these works used the processing units on the logic layer to process graphs, and these processing units can only access one memory bank at one time.

There are also some other graph processing accelerators. Ozdal *et al.* [24] proposed an energy-efficient ASIC design by integrating specific circuits for “Gather”, “Apply” and “Scatter” operations, cooperating with customized buffers and caches for vertex and edge data supplement. Graphicionado [11] proposed an on-chip pipeline structure and used eDRAM for on-chip graph data storage. GraphOps [23] and ForeGraph [6] proposed parallel graph processing architectures with on-chip memory on FPGA.

6 CONCLUSIONS

In this paper, we modify conventional DRAM architectures into GraphIA to overcome the “memory wall” challenges in large-scale graph processing problems. GraphIA is an **I**n-situ **A**ccelerator, which couples both large-capacity memory and computing resources in DRAM. Unlike previous graph processing accelerator designs, GraphIA fully exploits the internal memory bandwidth in DRAM for graph processing, rather than simply puts processing units “closer” to the memory. We organize heterogeneous banks in GraphIA chips into edge banks and vertex banks, cooperating with customized peripheral circuits. For scalability consideration, GraphIA chips are connected using scaling ring interconnection topology, which makes GraphIA scalable to larger graphs by employing more GraphIA chips. To tackle the performance loss caused by the irregular memory access in our multi-chip ring structure, we propose data duplication and scheduling schemes to optimize the memory access in heterogeneous banks, leading to 1.63× and 1.13× speedup respectively against baseline designs. Both heterogeneous banks (including data duplication and scheduling) and scaling ring interconnection scheme in GraphIA can be applied to previous graph processing accelerators. Our experimental results show that, by adopting above designs in GraphIA, our accelerator achieves 217× speedup against conventional CPU-DRAM designs.

7 ACKNOWLEDGEMENT

The authors thank the anonymous reviewers for their valuable comments. This work was supported by NSF 1719160, and CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] Junwhan Ahn et al. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*. IEEE, 105–117.
[2] 7-Zip LZMA Benchmark. 2018. Intel Skylake. https://www.7-cpu.com/cpu/Skylake.html. (2018).
[3] Yunji Chen et al. 2014. Dadiannao: A machine-learning supercomputer. In *MICRO*. IEEE, 609–622.
[4] Yuze Chi et al. 2016. Nxgraph: An efficient graph processing system on a single machine. In *ICDE*. IEEE, 409–420.
[5] Guohao Dai et al. 2016. FPGP: Graph processing framework on fpga a case study of breadth-first search. In *FPGA*. ACM, 105–110.
[6] Guohao Dai et al. 2017. ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture. In *FPGA*. ACM, 217–226.
[7] Guohao Dai et al. 2018. GraphH: A Processing-in-Memory Architecture for Large-scale Graph Processing. *IEEE TCAD* (2018).
[8] Paul Dlugosch et al. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE TPDS* 25, 12 (2014), 3088–3098.
[9] Mingyu Gao et al. 2017. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *ASPLOS*. ACM, 751–764.
[10] Joseph E Gonzalez et al. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework.. In *OSDI*. USENIX, 599–613.
[11] Tae Jun Ham et al. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *MICRO*. IEEE, 1–13.
[12] Song Han et al. 2016. EIE: efficient inference engine on compressed deep neural network. In *ISCA*. IEEE, 243–254.
[13] Manuel Holtgrewe et al. 2010. Engineering a scalable high quality graph partitioner. In *IPDPS*. IEEE, 1–12.
[14] Tianhao Huang et al. 2018. HyVE: Hybrid vertex-edge memory hierarchy for energy-efficient graph processing. In *DATE*. IEEE, 973–978.
[15] Micron Technology Inc. 2009. 4Gb: x4, x8, x16 DDR3 SDRAM. https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr3/4gb_ddr3_sdram.pdf. (2009).
[16] Micron Technology Inc. 2018. System Power Calculator Information. https://www.micron.com/support/tools-and-utilities/power-calc. (2018).
[17] Duckhwan Kim et al. 2016. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *ISCA*. IEEE, 380–392.
[18] Aapo Kyrola et al. 2012. Graphchi: Large-scale graph computation on just a pc. In *OSDI*. USENIX, 31–46.
[19] Jure Leskovec et al. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).
[20] Shuangchen Li et al. 2017. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *MICRO*. ACM, 288–301.
[21] Yucheng Low et al. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *VLDB Endowment* 5, 8 (2012), 716–727.
[22] Grzegorz Malewicz et al. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*. ACM, 135–146.
[23] Tayo Oguntebi et al. 2016. Graphops: A dataflow library for graph analytics acceleration. In *FPGA*. ACM, 111–117.
[24] Muhammet Mustafa Ozdal et al. 2016. Energy efficient architecture for graph analytics accelerators. In *ISCA*. IEEE, 166–177.
[25] Amitabha Roy et al. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*. ACM, 472–488.
[26] Dan Zhang et al. 2018. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. In *ASPLOS*. ACM, 593–607.
[27] Mingxing Zhang et al. 2018. GraphP: Reducing Communication of PIM-based Graph Processing with Efficient Data Partition. In *HPCA*. IEEE, 544–557.
[28] Mingxing Zhang et al. 2018. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *ASPLOS*. ACM, 608–621.
[29] Xiaowei Zhu et al. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *ATC*. USENIX, 375–386.
[30] Xiaowei Zhu et al. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *OSDI*. USENIX, 301–316.