Balancing Memory Accesses for Energy-Efficient Graph Analytics Accelerators

Mingyu Yan*†‡, Xing Hu‡, Shuangchen Li‡, Itir Akgun‡, Han Li*†,
Xin Ma‡, Lei Deng‡, Xiaochun Ye*¶, Zhimin Zhang*, Dongrui Fan*†, and Yuan Xie‡
*SKLCA, ICT, CAS, Beijing, China, †UCAS, Beijing, China, ‡UC Santa Barbara, CA, USA
¶ Corresponding author, Email: yexiaochun@ict.ac.cn

***Abstract*—Domain-specific accelerators for graph analytics leverage a large on-chip memory in order to tackle the intensive random memory accesses, offering higher performance and energy efficiency than conventional architectures. However, limited by the inefficient usage of on-chip memory, current accelerators suffer from energy and performance bottlenecks due to the large amount of off-chip memory accesses. In this work, we introduce an online preprocessing step for the vertex-centric programming model based on our observation of imbalanced memory bandwidth utilization between two execution phases. Our scheme improves energy efficiency and performance by significantly reducing off-chip accesses in two ways. First, we sequence random off-chip memory accesses to balance memory bandwidth demands and improve the utilization of on-chip memory. Second, we prune active leaf vertices to avoid redundant memory accesses. We evaluate our method on a state-of-the-art graph analytics accelerator and achieve 1.6× speedup while reducing energy consumption by 42% on average.**

***Index Terms*—Graph Analytics, Accelerators, Memory Access**

I. Introduction

Graph analytics has emerged as an important workload in data centers due to its wide applicability to many fields such as social networks, informatics, cybersecurity, chemistry, and hence motivated efforts for acceleration. Researchers have found that the key bottleneck of graph analytics is the intensive random memory accesses which lead to high energy consumption [1].

Consequently, conventional general-purpose hardware platforms fail to meet the power requirement of data centers [2] due to the lack of an optimized memory system [1], [3], [4]. Recent studies show that domain-specific accelerators for graph analytics, such as Graphicionado [1] and HyVE [5], can offer much higher energy efficiency and performance. These accelerator designs leverage a large on-chip memory to store most of the irregularly accessed data. Their energy and performance benefits rely tremendously on the large on-chip memory, since accessing on-chip memory is less energy consuming and much faster than accessing off-chip DRAM memory.

However, the large on-chip memory becomes a new bottleneck for the accelerator in terms of energy and performance. Up to 90% of the total power is consumed by the 64MB on-chip memory in Graphicionado [1]. Furthermore, accelerators cannot utilize the on-chip memory efficiently [1], which causes off-chip memory accesses and adds to the energy consumption.

978-1-7281-2954-9/19/$31.00 ©2019 IEEE

Our evaluations show that the capacity-normalized speedup can decrease up to 78% for on-chip memory capacity ranging from 8MB to 64MB. With the explosive growth of graph sizes, even larger on-chip memories are in demand, which will further aggravate the energy efficiency of graph accelerators.

To overcome these challenges, we propose a memory access optimization technique to reduce off-chip memory accesses and improve on-chip memory utilization for energy-efficient graph analytics accelerator designs. Through in-depth studies on graph accelerators, we observe imbalanced memory bandwidth demands between two alternating execution phases (*Scatter* phase and *Apply* phase) in vertex-centric programming model. *Scatter* phase is *memory bound* with large number of irregular memory accesses, while the following *Apply* phase contains a small number of sequential memory accesses. Our analysis shows that a significant amount of bandwidth is underutilized in the Apply phase, which reveals a large and currently unexploited design space. Thus, we leverage the bandwidth slack of the Apply phase and introduce an online preprocessing step to transform irregular accesses into sequential accesses for the Scatter phase and prune the graph for vertex activations. This step helps reduce the required on-chip memory capacity and provides an opportunity to improve its utilization. Furthermore, it significantly reduces off-chip memory accesses and in turn reduces the energy consumption. Finally, we apply our optimization method to Graphicionado [1], a state-of-the-art graph analytics accelerator, as a case study to demonstrate the effectiveness of our proposed method. Our specific contributions are summarized as follows:

- We observe memory bandwidth utilization imbalance in the popular vertex-centric programming model and identify redundant memory accesses through iterations.
- We propose a memory access optimization that balances memory accesses to improve on-chip memory utilization of graph accelerators and removes redundant memory accesses by pruning active leaf vertices.
- We apply the proposed technique to Graphicionado [1] as a case study. Experimental results show 1.62× speedup while reducing energy consumption by 42% on average.

II. Background

A. Vertex-Centric Programming Model

Algorithms for graph analytics ranging from single-source shortest path (SSSP) to breadth-first search (BFS) can be expressed by the general Gather-Apply-Scatter (GAS) programming model [6], one that is widely used by many famous

graph software frameworks [7]–[9] and graph accelerators [1], [10]. The execution of GAS-based applications can be simplified to two phases: *Scatter* and *Apply* [1], [10], in a push-based vertex-centric programming model (PB-VCPM) as shown in Algorithm 1. In this model, the Compressed Sparse Row (CSR) format is used to express the graph as shown in Fig. 1(a) and Fig. 1(b). In every iteration, destination vertices are traversed by accessing the outgoing edges of every active vertex, followed by updates to the temporary property of destination vertices with results from Process_Edge and Reduce functions in *Scatter* phase. *Apply* phase follows *Scatter* phase and executes Apply function with the temporary property to update property of vertices and constructs the active vertex array. These two phases are executed iteratively until no active vertex remains. The detailed operations of Process_Edge, Reduce and Apply functions which vary across different graph algorithms can be referenced to prior work [1].

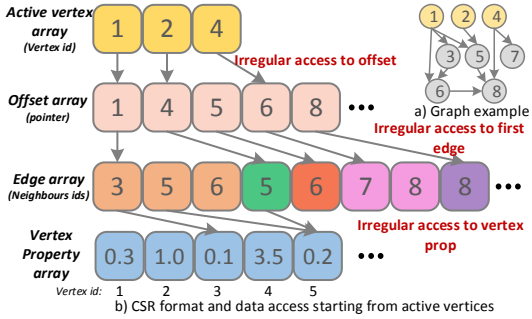

Fig. 1. **A graph example and its CSR representation**

Algorithm 1: ***Push-Based Vertex-Centric Programming Model***

```
1  ◁ Scatter phase
2  for (i = 0; i < ActiveVertexCount; i++) {
3      vid = ActiveVertexArray[i];
4      uprop = ActiveVertexPropArray[i];
5      offset = OffsetArray[vid]; //irregular access
6      for (e = EdgeArray[offset]; e.src == vid; e =
         EdgeArray[++offset]) {
7          res = Process_Edge(uprop, e.weight);
8          vtemp = VTempProperty[e.dst]; //irregular access
9          vtemp = Reduce(vtemp, res);
10         VTempProperty[e.dst] = vtemp; //irregular access
11     }
12 }
13 ◁ Apply phase
14 for (i = 0; i < TotalVertexCount; i++) {
15     vprop = VProperty[i];
16     vtemp = VTempProperty[i];
17     vconst = VConst[i];
18     temp = Apply(vprop, vtemp, vconst);
19     VProperty[i] = temp;
20     if (temp != vprop) {
21         ActiveVertexArray[ActiveVertexCount] = i;
22         ActiveVertexPropArray[ActiveVertexCount] =
             temp;
23         ActiveVertexCount++;
24     }
25 }
```

B. Graph Accelerators

Graph analytics accelerators achieve higher performance and energy efficiency over conventional hardware platforms since their datapath and memory subsystem are tailored specifically for graph analytics applications. Graphicionado [1] optimizes irregular accesses by utilizing a specialized large on-chip embedded DRAM (eDRAM) scratchpad memory. This on-chip memory stores the irregular memory accesses to temporary property and offset array. It significantly improves the performance and lowers the pipeline stall time by minimizing the latency of irregular accesses, achieving a 1.76-6.54× speedup and 50× energy efficiency compared to the software framework solution [11].

III. Motivation

Larger graphs drive the demand to increase the on-chip memory capacity. Example of such large graphs includes social networks in Facebook, where the number of users (nodes) are around 2 billion and increasing by 17% every year [12]. Enlarging the on-chip memory capacity appears to be a resulting benefit from the energy efficiency and performance gain.

Nevertheless, several challenges still exist and limit the scaling of on-chip memory capacity. The first challenge lies in the degradation of the utilization of a larger on-chip memory. Figure 2(a) shows our performance evaluation using Graphicionado to execute BFS and SSSP applications for different on-chip memory capacities. While continuous speedup can be achieved with larger on-chip memory, the gain from capacity-normalized speedup of BFS is decreasing, as shown in Fig. 2(b) (cross). *This indicates that the on-chip memory utilization decreases with increasing capacity, which means that off-chip memory accesses cannot be continuously or efficiently reduced*. The second challenge is related to the increased power consumption caused by such memory scaling. As shown in Fig. 2(b) (star), an on-chip memory with area of 44 mm^2 [13] contributes up to 90% of the power consumption at 64MB capacity. *This indicates that the accelerator becomes more power hungry with a larger on-chip memory capacity*.

To tackle the aforementioned challenges, we introduce an online preprocessing step for PB-VCPM to improve the utilization of on-chip memory and prune the graph based on our following observations described in the Sec. IV-A. This preprocessing step improves energy efficiency and performance as it greatly reduces redundant memory accesses.

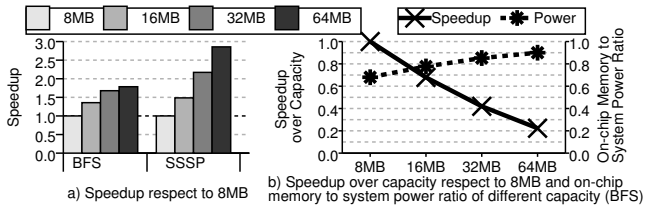

Fig. 2. **Impact of on-chip memory capacity on speedup and power**

IV. Memory Access Optimization

In this section, we first analyze the memory accesses of PB-VCPM during the execution of graph applications. We then propose memory access optimization methods to improve memory utilization and energy efficiency.

A. Observations

Observation 1: Imbalanced Memory Bandwidth Utilization. We observe that dissimilar memory access demands between the *Scatter* and the *Apply* phases lead to imbalanced memory bandwidth utilization in PB-VCPM. As shown in

Algorithm 1, most of the data structures in *Scatter* phase are accessed irregularly in memory, except for accesses to active vertices and consecutive edges. In contrast, all data structures in the *Apply* phase are accessed sequentially.

In addition to analyzing graph applications, we conduct additional experiments to quantitatively evaluate the access patterns of *Scatter* and *Apply* phases as shown in Fig. 3. We make two observations from this figure: 1) *Scatter* phase is dominated by irregular memory accesses which occupy up to 90% of the total memory accesses. Furthermore, an irregular memory access fetches an entire cacheline only to use a small portion of the fetched data, which reduces the effective bandwidth. 2) All of the data structures in the *Apply* phase are accessed sequentially in memory. Therefore, *Apply* phase utilizes memory bandwidth more effectively, resulting in fewer memory accesses. In summary, the performance of *Scatter* phase is memory-bounded while *Apply* phase is not.

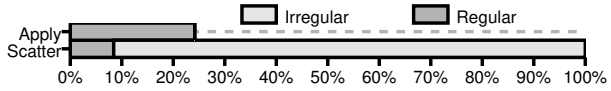

Fig. 3. **Average memory bandwidth utilization normalized to *Scatter***

Observation 2: Redundant Memory Accesses. We observe that there are large amount of redundant memory accesses in the *Scatter* phase due to active leaf vertices. Active leaf vertices introduce one type of sequential memory access and two types of irregular memory accesses in the *Scatter* phase. As shown in Algorithm 1, active leaf vertices need to access the offset array and then the edge array to determine the end of the traversal. All of these memory accesses are redundant because the leaf vertices have no edges to traverse. As shown in Fig. 4, active leaf vertices occupy 17% of active vertices on average and also cause an additional iteration step (i.e. leaf vertices occupy 100% in last iteration), resulting in even more memory accesses. For larger graphs, the situation becomes worse due to each graph being partitioned into several slices, where more active vertices become leaf vertices during the slicing process and cause even more memory accesses.

Fig. 4. **Leaf to active vertex count ratio through iterations**

B. Balancing Memory Accesses with Temporary Offset Array

Underutilized memory bandwidth in the *Apply* phase can be used to sequence irregular memory accesses of *Scatter* phase. As illustrated in Fig. 1, irregular memory accesses in *Scatter* phase include: *offset*, *first edge of an active vertex*, and *destination vertex temporary property of every outgoing edge*. The offset corresponds to the active vertex and active vertices are determined in the *Apply* phase. Thus, irregular accesses to the offset can be converted into sequential accesses by constructing a temporary offset array in the *Apply* phase.

We modify the programming model to construct a temporary offset array without breaking the fundamentals of GAS model. As illustrated in Algorithm 2, we add line 11 to read offset sequentially and line 15 to construct temporary offset array in the *Apply* phase. Next, we modify line 5 to access

Algorithm 2: ***Optimized Programming Model***

```
1  ◁ Scatter phase
2  for (i = 0; i < ActiveVertexCount; i++) {
3      vid = ActiveVertexArray[i];
4      uprop = ActiveVertexPropArray[i];
5      offset = TempOffsetArray[i];//sequential access
6      line 6 − 11 of Algorithm 1;
7  }
8  ◁ Apply phase
9  for (i = 0; i < TotalVertexCount; i++) {
10     line 15 − 19 of Algorithm 1;
11     offset = OffsetArray[i]; //sequential access
12     if ((temp != vprop) && (offset != 0)) {
13         ActiveVertexArray[ActiveVertexCount] = i;
14         ActiveVertexPropArray[ActiveVertexCount] =
             temp;
15         TempOffsetArray[ActiveVertexCount] = offset;//sequential
             access
16         ActiveVertexCount++;
17     }
18 }
```

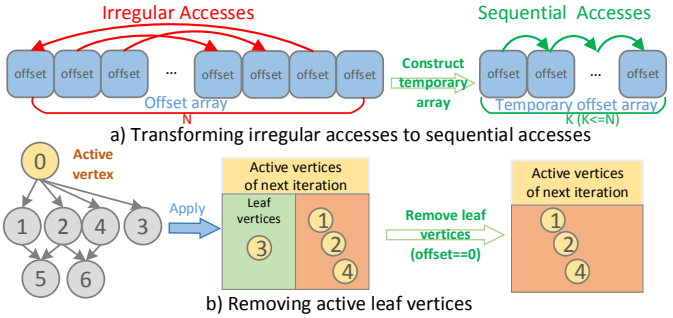

Fig. 5. **Proposed memory optimization methods**

offset by index i. As illustrated in Fig. 5(a), irregular accesses to the offset in *Scatter* phase will be sequenced by constructing the temporary offset array.

C. Pruning Active Leaf Vertices

To eliminate the redundant irregular memory accesses deriving from active leaf vertices, we add an additional condition to vertex activation to avoid leaf vertices (i.e., vertices that have no outgoing edges) from being activated. As illustrated in Fig. 5(b), in order to avoid activating leaf vertices (i.e, vertex 3), we prune them after executing the *Apply* function. We specify leaf vertices to have zero offset during offset and edge array initialization. As illustrated in Algorithm 2, we add condition *offset != 0* in line 12 to identify leaf vertices and avoid activation. This optimization reduces a significant amount of irregular memory accesses. Furthermore, this optimization can also reduce the number of iterations, since an iteration can be omitted if only leaf vertices remain in the *Apply* phase (i.e. no active vertices for the next iteration).

V. Use Case on Graphicionado

Our optimization is suitable for all graph accelerators based on the vertex-centric programming model. We take Graphicionado [1], a state-of-the-art graph analytics accelerator, as a study case to demonstrate the effectiveness of our optimization methods without loss of generality. Graphicionado decouples *Scatter* phase and *Apply* phase to several pipeline stages executed by various specified units that employs datatype and memory subsystem specializations while offering workload-specific reconfigurable blocks. In order to support our modification of the vertex programming model, we also need to modify the Graphicionado pipeline.

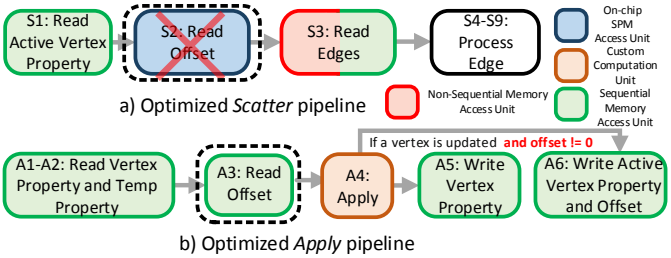

Fig. 6. **Optimized Graphicionado pipeline**

A. Balancing Memory Accesses in Graphicionado Pipeline

***Scatter* Pipeline**. As illustrated in Fig. 6(a), we remove the *Read Offset* stage, which is executed by on-chip scratchpad memory (SPM) access unit. The original accelerator design places the whole offset array and vertex temporary property array in the on-chip memory. Sizes of both arrays are equal to the total number of vertices in the graph. Thus, the offset array occupies half of the on-chip memory capacity. In contrast, we prefetch the offset in the *Read Active Vertex Property* stage since the offset array has already been sequenced. This optimization removes the offset array from on-chip memory and saves half of the on-chip memory capacity. The saved half of the on-chip memory can be utilized in the following two ways: 1) using it to store more vertex temporary property data for larger graphs, helping to reduce memory access and thus improving energy efficiency and performance; 2) reducing the on-chip memory capacity, reducing area and energy of the chip without significant performance loss.

***Apply* Pipeline**. As illustrated in Fig. 6(b), we add *Read Offset* stage to sequentially read offset and modify activation condition by adding *Offset Valid* check. If the vertex is activated, the corresponding offset will be written to temporary offset array in *Write Active Vertex Property* stage. Next, *Scatter* phase of the next iteration will prefetch active vertices along with corresponding offset, to avoid irregular memory accesses to offset. In addition, we specifically assign the value of the offset to zero if the corresponding vertex has no outgoing edges, to avoid activating a leaf vertex.

B. Pruning Active Leaf Vertices for Large Graphs

Our proposed optimization method for pruning active leaf vertices also supports graph slicing for large graphs. In the Graphicionado design, a large graph is partitioned into multiple slices when the on-chip memory cannot cache all vertex property data. As illustrated in Fig. 7(a), vertices of a graph shown on Fig. 5(b) are sliced into two and all their incoming edges will be placed into same slice. Every slice has its own offset array and the on-chip memory needs to replace in the offset array of the slice being processed.

As illustrated in Fig. 7(a), original design does not remove the active leaf vertices and uses all active vertices to traverse each slice, even though vertices 1&3 have no outgoing edges in slice 1 and vertices 3&4 have no outgoing edges in slice 2. As shown on Fig. 7(b), to prune the active leaf vertices, we identify leaf vertices(i.e, *offset[slice number] == 0*) for different slices and construct an active vertex array for each slice with active leaf vertices removed. Hence, our optimization removes redundant memory accesses for large graphs, thereby reducing the energy consumption.

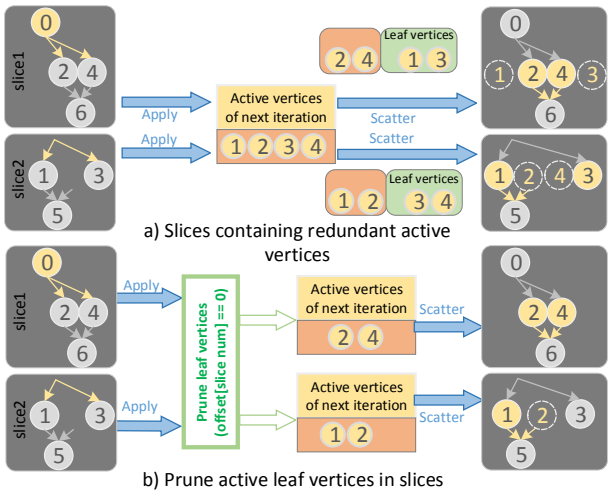

Fig. 7. **Pruning active leaf vertices in graph slices (Sliced graph in Fig. 5).**

VI. Evaluation

A. Methodology

For the evaluation, we implement an in-house cycle-accurate simulator of Graphicionado [1], and then apply our optimization methods on this simulator for comparison. In our evaluation, we adopt CACTI 6.5 [13] for on-chip memory evaluation and use Ramulator [14] and DRAMPower [15] to simulate the latency and energy consumption of DDR4. We generate energy and timing models of DDR4 using Micron System Power Calculators [16]. We execute BFS and SSSP workloads on accelerators with 3 graph datasets shown in Table I. For the SSSP evaluation on unweighted graphs, random integer weights between 1 and 256 are assigned.

We make comparisons among the baseline Graphicionado pipeline (ICI) and two optimized solutions, namely area-orientated optimization (O1) and energy-oriented optimization (O2). As mentioned in Section V-A, our optimizations can be used to reduce the on-chip area or energy consumption. Hence, to demonstrate the effectiveness of our optimizations, O1 only has half on-chip memory capacity of ICI, showing the area reduction of our optimization. O2 uses the same on-chip memory capacity as ICI to show the energy consumption reduction of our optimization. The specification for these three solutions is listed in Table II.

TABLE I
GRAPH DATASETS USED IN EVALUATION

Graph	Vertices	Edges	Brief Explanation
Flickr(FR) [17]	0.82M	9.84M	Flickr Crawl Graph
Pokec(PK) [18]	1.63M	30.62M	Social Network Graph
LiveJournal(LJ) [19]	4.84M	68.99M	LiveJournal Follower Graph

TABLE II
EVALUATION SPECIFICATIONS

	ICI	O1	O2
On-chip memory	1×capacity	$\frac{1}{2}$×capacity	1×capacity
Off-chip memory	1×DDR4-2133, 17GB/s channel		
Compute unit	1×1Ghz pipeline		

Note: On-chip memory capacity varies in different graph of our evaluation.

Since offsets are prefetched with active vertices, area-oriented optimization mode with half the on-chip memory capacity is able to store as many vertex temporary property data as the original design. On the other hand, energy-oriented

optimized mode is able to store twice the amount of vertex temporary property of the original design. To demonstrate that our optimization scheme has better memory access efficiency even when the graph is not accommodated with a large enough on-chip memory, we evaluate all applications for full capacity (i.e, graph is only 1 slice, namely ALGO-1), half capacity (i.e, graph is sliced into 2 slices, namely ALGO-2) and quarter capacity (i.e, graph is sliced into 4 slices, namely ALGO-4) of Graphicionado. Fig. 8 shows how data structures are stored in on-chip memory differently for different memory capacities.

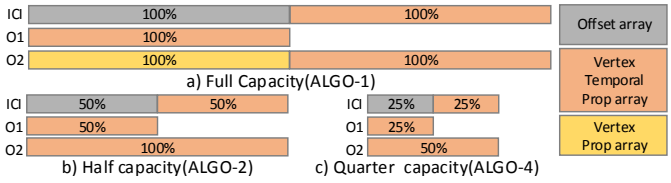

Fig. 8. **On-chip data layout for different on-chip memory capacities**

B. Memory Access Efficiency

First, we evaluate the memory access efficiency for the three scenarios across different algorithms. Fig. 9(a) shows the average memory accesses across all graphs, normalized to Graphicionado. Fig. 9(b) shows the memory accesses on PK dataset, normalized to Graphicionado. ALGO-1, ALGO-2 and ALGO-4 represent scenarios with different on-chip memory capacities. The experiment results indicate the following conclusions: 1) When the on-chip memory is not large enough to hold the entire *offset* and *vertex temporary property* arrays, the number of memory accesses in O1 and O2 are reduced by up to 22% and 39% respectively, with decreasing on-chip memory capacity. 2) When the on-chip memory is large enough, such as ALGO-1 case, the number of memory accesses increases in O1 and decreases in O2 compared to ICI. This is because, in the ALGO-1 case, ICI is able to accommodate all *vertex temporary property* data and the entire *offset* array, which is read from memory in the first iteration to its on-chip memory. O1, however, has to access offset array in each iteration, leading to extra memory accesses. The reason that the O2 mode achieves better performance in ALGO-1 is because O2 leverages the half of on-chip memory to store the entire vertex property array in on-chip memory, and then decreases the number of vertex property accesses in *Apply* phase of each iteration. Moreover, O2 reduces the number of slices of the vertex temporary property array by half since its on-chip memory can store 2× more than ICI in ALGO-2 and ALGO-4 cases. The total number of memory accesses is reduced by up to 39%, demonstrating the effectiveness of our memory optimizations.

Better memory access efficiency comes from both sequencing the irregular memory accesses and pruning leaf vertex activations. Next, we evaluate memory accesses after pruning the active leaf vertices. Fig. 10 shows the normalized active vertex and edge accesses of PK dataset in *Scatter* phase of our optimization with respect to Graphicionado when executing SSSP algorithm. Modification of activation condition results in active vertex access reduction by up to 40% and 62% for O1 and O2, respectively. Furthermore, corresponding edge accesses are reduced by up to 9% and 11%. Note that the number of memory accesses to active vertices in O1 and O2

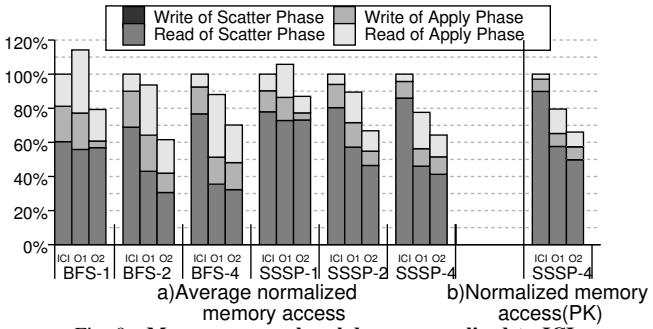

Fig. 9. **Memory access breakdown normalized to ICI**

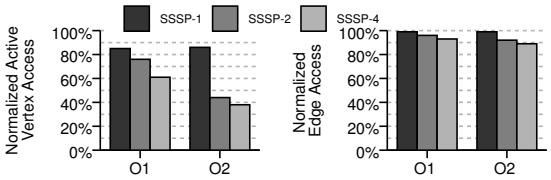

Fig. 10. **Active vertex and edge accesses of PK (SSSP) normalized to ICI**

with PK dataset running SSSP-1 case decrease by up to 13% due to the fact that PK contains many leaf vertices activated in ICI mode, but are avoided in our optimization.

C. Performance

In this subsection, we compare the overall performance of ICI, O1, and O2 solutions. Fig. 11(a) shows the average speedup across all graphs, normalized to Graphicionado for different on-chip memory capacities. When the on-chip memory capacity is decreased, O1 and O2 perform better than ICI by up to 1.3× and 1.6× respectively. Speedup improvement in O1 and O2 cases are derived from removing active leaf vertex accesses in each iteration, which grows with the number of slices. O2 case achieves higher speedup because the capacity of O2 accommodates 2× vertex temporary property, which reduces the number of slices and in turn reduces the number of the accesses to offset array by half. O1 case is slower than ICI in ALGO-1 since our work accesses *offset* array from off-chip memory while ICI accesses it from on-chip memory, and the improvement of our optimization is not enough to cover this overhead. The execution time of SSSP-4 with PK graph is shown in Fig. 11(b). The execution time of O1 and O2 in *Scatter* phase are reduced by 35% and 42% due to pruning active leaf vertices and moving offset accesses into *Apply* phase. On the other hand, respective execution time in *Apply* phase are increased by 10% and 5% due to the offset array accesses. For energy-oriented optimization, our work achieves up to 1.6× performance improvement.

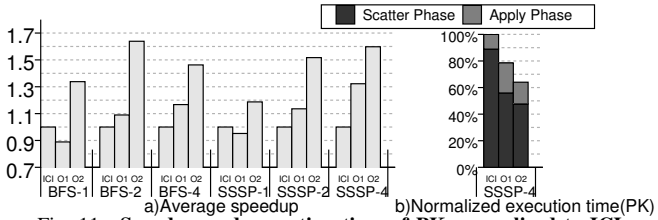

Fig. 11. **Speedup and execution time of PK normalized to ICI**

D. Energy Consumption

In this subsection, we evaluate the energy consumption of ICI, O1, and O2 solutions. The following evaluation only shows the energy consumption of on-chip memory and DRAM

since the focus in Graphicionado [1] is on optimizing the memory subsystem and other components consume only around 10% of system energy. Fig. 12(a) shows the average energy consumption across all graphs, normalized to ICI. As the on-chip capacity decreases, the average energy consumption of O1 and O2 are reduced by up to 30% and 42% respectively. The energy consumption reduction of O1 comes from the reduced execution time, on-chip memory capacity and active leaf vertex accesses. The energy consumption reduction of O2 mainly comes from reduced off-chip memory accesses. Energy consumption of SSSP-4 with PK dataset degrades as shown in Fig. 12(b). The DRAM energy of O2 in *Scatter* phase decreases due to reduced off-chip memory accesses. On the other hand, the DRAM energy in *Apply* phase increases due to accesses to the offset array. With decreasing on-chip memory capacity, our memory optimizations are beneficial for improving energy efficiency even more, by up to 42%.

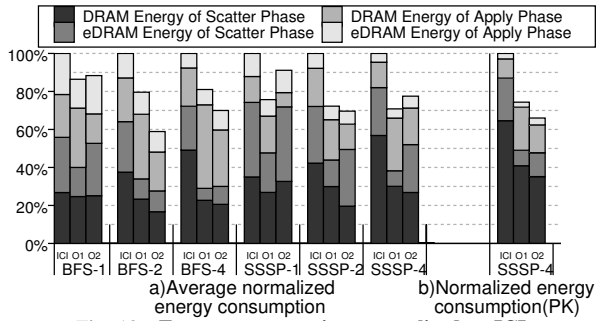

Fig. 12. **Energy consumption normalized to ICI**

E. Discussion

Although we evaluate BFS and SSSP algorithms, the proposed memory optimization can also improve the energy efficiency of other graph algorithms such as Connected Components and Single Source Widest Path. Furthermore, our optimization can also be used in software frameworks running on CPU or GPU by adopting the proposed modified graph programming model. However, most of these frameworks use offline preprocessing to improve cache efficiency, which modifies the data organization of CSR format limiting the benefits of our optimization. For applications with narrow and deep graph structures, which only have few active vertices in each iteration and exhibit large number of iterations, the proposed optimization is less effective. However, according to analysis of social and information networks [20], most of these graphs exhibit a small diameter graph structure. Thus, the proposed method can generally be effective.

VII. Conclusion

In this paper, we observe the inefficient memory utilization of the popular vertex-centric programming model that arise from the imbalanced memory bandwidth demands of two execution phases and redundant memory accesses due to active leaf vertices. Based on these observations, we present memory optimization for graph analytics accelerators, which balances memory accesses of two execution phases and prunes active leaf vertices. We apply our optimization on a state-of-the-art graph analytics accelerator in order to reduce memory accesses and improve on-chip memory utilization. Results show that the proposed method achieves up to 1.6× speedup with 42% energy reduction on average. Our observation shows a large yet unexploited design space for memory optimization along the entire graph application execution. This provides a great opportunity to improve memory bandwidth efficiency and energy efficiency in current graph analytics accelerators.

Acknowledgment

This work was supported by the National Key Research and Development Program (Grant No. 2018YFB1003501), the National Natural Science Foundation of China (Grant No. 61732018, 61872335, and 61802367), Austrian-Chinese Cooperative RD Project (FFG and CAS)(Grant No. 171111KYSB20170032), the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDA18000000), the Innovation Project Program of the State Key Laboratory of Computer Architecture (Grant No. CARCH3303, CARCH3407, CARCH3502, and CARCH3505), and the National Science Foundation (Grant No. 1500848 and 1730309).

References

[1] T. J. Ham *et al.*, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[2] L. A. Barroso and U. Hlzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, Dec 2007.

[3] A. Basak *et al.*, "Exploring core and cache hierarchy bottlenecks in graph processing workloads," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 197–200, July 2018.

[4] A.Basak *et al.*, "Analysis and optimization of the memory hierarchy for graph processing workloads partition," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[5] T. Huang *et al.*, "HyVE: Hybrid vertex-edge memory hierarchy for energy-efficient graph processing," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, pp. 973–978.

[6] J. E. Gonzalez *et al.*, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 17–30.

[7] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146.

[8] Y. Low *et al.*, "GraphLab: A new framework for parallel machine learning," *arXiv:1408.2041 [cs]*, 2014.

[9] Z. Fu *et al.*, "MapGraph: A high level API for fast development of high performance graph analytics on GPUs," in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, ser. GRADES'14. ACM, 2014, pp. 2:1–2:6.

[10] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. ACM, 2015, pp. 105–117.

[11] N. Sundaram *et al.*, "GraphMat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, 2015.

[12] Global social media ranking| statistic. [Online]. Available: https://www.statista.com/statistics

[13] Cacti. [Online]. Available: http://www.hpl.hp.com/research/cacti/

[14] Y. Kim *et al.*, "Ramulator: A fast and extensible DRAM simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[15] K. Chandrasekar *et al.* Drampower: Open-source dram power & energy estimation. [Online]. Available: http://www.drampower.info

[16] System power calculator information. [Online]. Available: https://www.micron.com/support/tools-and-utilities/power-calc

[17] T. A. Davis *et al.*, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[18] M. Z. L. Takac, "Data analysis in public social networks," *IEEE International Scientific Conference International Workshop Present Day Trends of Innovations*, 2012.

[19] L. Backstrom *et al.*, "Group formation in large social networks: Membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. ACM, 2006, pp. 44–54.

[20] J. Leskovec *et al.*, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, pp. 29–123, 2009.