A Locality-Aware Energy-Efficient Accelerator for Graph Mining Applications

Pengcheng Yao[†], Long Zheng[†], Zhen Zeng[†], Yu Huang[†], Chuangyi Gui[†], Xiaofei Liao[†], Hai Jin[†], Jingling Xue[‡]
[†]National Engineering Research Center for Big Data Technology and System/Services Computing Technology and System Lab/Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
[‡]School of Computer Science and Engineering, UNSW Sydney, Australia
{pcyao, longzh, zzeng, yuh, chygui, xfliao, hjin}@hust.edu.cn, jingling@cse.unsw.edu.au

***Abstract*—Graph mining is becoming increasingly important due to the ever-increasing demands on analyzing complex structures in graphs. Existing graph accelerators typically hold most of the randomly-accessed data in an on-chip memory to avoid off-chip communications. However, graph mining exhibits substantial random accesses from not only vertex dimension but also *edge dimension* (with the latter being excessively more complex than the former), leading to significant degradations in terms of both performance and energy efficiency.**

We observe that the most random memory requests arising in graph mining come from accessing a small fraction of valuable (vertex and edge) data when handling real-world graphs. To exploit this extension locality with maximum parallelism, we architect GRAMER, the first graph mining accelerator. GRAMER contains a specialized memory hierarchy, where the valuable data (precisely identified through a cost-efficient heuristic) is permanently resident in a high-priority memory while others are maintained in a cache-like memory under a lightweight replacement policy. The specific pipelined processing units are carefully designed to maximize computational parallelism. GRAMER is also equipped with a work-stealing mechanism to reduce load imbalance. We have implemented GRAMER on a Xilinx Alveo U250 accelerator card. Compared with two state-of-the-art CPU-based graph mining systems, Fractal and RStream, running on a 14-core Intel E5-2680 v4 processor, GRAMER achieves not only considerable speedups ($1.11\times \sim 129.95\times$) but also significant energy savings ($5.79\times \sim 678.34\times$).

***Index Terms*—graph mining, extension locality, accelerator**

I. Introduction

Graphs have been widely used for analyzing the complex relationships between entities. The massive explosion in the data volume has made graph analytics increasingly attractive in a variety of domains, including social networks [29], advertisement [26], and bioinformatics [4]. Because of graph irregularity [27], graph analytics often suffers from severe random access problems, making its acceleration extremely difficult on traditional architectures, even by making the best use of software optimizations from a domain expert [5], [6].

It has been demonstrated that hardware acceleration is one of the most promising ways for overcoming the irregular memory access challenges in graph analytics applications, by offering at least a tenfold performance improvement over general-purpose processors [11], [17], [31], [44], [46]. However, most of these earlier accelerators are typically designed for *graph processing* applications (such as BFS, CC, and PageRank), which focus on performing value computations for a graph. To parallelize graph processing applications, these graph accelerators typically use a vertex-centric programming model [14], [28], [29], [35], where each vertex is considered as an individual processing element and communicates with its neighbors. As a result, graph processing generally exhibits random accesses mostly in terms of a graph's vertices [11], [17], [44], [46], which are usually small enough to fit directly into the on-chip memory of an accelerator, avoiding off-chip communications effectively.

Graph mining, which represents another important but different class of graph analytics applications, aims at discovering structural patterns of a graph rather than value computation. Fitting a graph mining application into the vertex-centric model often needs substantial reformulation. A great deal of recent work such as Arabesque [38], RStream [42], and G-thinker [43] has made advances to fill this gap by developing a declarative subgraph-centric programming model for solving the programmability challenge faced by the general-purpose graph mining problems. In this model, graph mining needs to iteratively extend each subgraph (initialized to be a vertex of an input graph) by accessing its irregular neighboring vertices and their associated edges [38], inducing substantial random accesses to not only vertices but also *edges*.

In this context, existing graph processing accelerators are not expected to handle graph mining applications efficiently. First, the size of an input graph can easily be over giga-bytes. It is impractical to fit all randomly-accessed vertex and edge data into a limited on-chip memory (as is currently available in existing graph accelerators). Second, as more than trillion-scale subgraphs may be generated, graph mining can have more random accesses than graph processing by several orders-of-magnitude. Even if we can slice graph data to fit each slice into the on-chip memory, the overwhelming number of off-chip communications will cause undue performance degradation. In this paper, we focus on building an energy-efficient accelerator in which the most random-access-induced off-chip communications in graph mining can be transformed into fast on-chip memory accesses.

In this research, we observe that graph mining on real-world graphs exhibits significant *extension locality* in the sense that most random memory requests come from accessing a small fraction of valuable vertex and edge data. Real-world graphs

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00077

follow a power-law degree distribution, where a few vertices have many edges while many have a few [14]. In this case, when adding a vertex and its associated edges to a subgraph being extended, some high-degree vertices will have a higher chance to be explored. Therefore, these vertices can be included in the most intermediate subgraphs, further amplifying the skewness of memory accesses. Consider motif counting on Mico (as detailed in Section II-D), most memory accesses (94.57%) are from only 5% of the graph data. This has motivated us with inventing a differential on-chip memory management, in which a handful of valuable data is always resident in the on-chip memory while the remaining data is dynamically maintained. However, realizing this idea for exploiting the extension locality remains challenging. First, it is non-trivial to accurately identify the valuable data, which can be complexly associated with the degrees of a vertex and its multi-hop neighbors. Second, substantial intermediate results in graph mining also exert tremendous pressure on memory bandwidth and significantly limit the computational parallelism.

In this paper, we present GRAMER, the first graph mining accelerator, which consists of a brand new memory subsystem and specialized processing units. The whole on-chip memory in GRAMER is divided into a high-priority memory and a low-priority memory. The high-priority memory is used to permanently store the frequently accessed data, which is heuristically predicted through a fast yet accurate graph preprocessing. The low-priority memory is organized as a standard set-associative cache but dynamically maintained under a new replacement policy for exploiting the extension locality. GRAMER also features a pipeline inspired from a depth-first-search execution model [12], where the most intermediate results obtained from each subgraph being extended can be periodically reused and ultimately discarded, enabling a flexible design for maximizing the computational parallelism achieved. Finally, each processing unit of GRAMER contains a work stealing mechanism to reduce the potential load imbalance.

In this paper, we make the following contributions:

- We characterize the irregular memory access patterns in graph mining and discover their potential extension locality opportunities for solving the memory bottleneck.
- We introduce a graph mining accelerator that contains a locality-aware on-chip memory hierarchy to exploit the extension locality and a set of pipelined processing units to maximize the computational parallelism.
- We have implemented GRAMER in RTL and evaluated it on a Xilinx Alveo U250 accelerator card. Our results show that GRAMER outperforms Fractal [12] and RStream [42] by $1.11\times \sim 129.95\times$ in terms of performance speedups and $5.79\times \sim 678.34\times$ in terms of energy savings.

The rest of this paper is organized as follows. Section II introduces the background and motivation. Section III presents the architecture of GRAMER. Sections IV and V introduce the memory and pipeline specialization, respectively. Section VI describes and analyzes our results. We survey the related work in Section VII and conclude the paper in Section VIII.

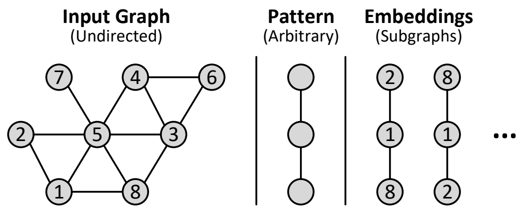


Fig. 1. A sample graph for illustrating graph mining concepts, including a pattern and two of its instantial embeddings. The number inside a given vertex represents its vertex ID. In this example, all attributes are NULL.

II. BACKGROUND AND MOTIVATION

In this section, we first introduce some requisite preliminaries of graph mining, which will be further characterized through a motivating study in order to facilitate understanding its salient properties from the architectural perspective.

A. Graph Mining

Terms. In graph mining, a *graph* G consists of vertices and edges, which can be labeled with some attributes and unique identifiers (IDs). The *degree* of a vertex represents the number of its associated edges. An *embedding* is a subgraph with a subset of vertices and edges of G. A *pattern* is an arbitrary graph, which is independent of G. An embedding e is *isomorphic* to a pattern p iff we can find a one-to-one mapping of the vertices and edges between e and p. In particular, the two embeddings are *automorphic* and can be considered identical if they have the same set of vertices and edges. Consider the pattern in Figure 1, the two example embeddings are isomorphic to the given pattern and are also automorphic.

Problem Statement. In this paper, GRAMER focuses on accelerating graph mining [1], [12], [38], [42] which aims at exploring all patterns that satisfy some user-defined criteria. A pattern is typically evaluated by enumerating all of its isomorphic embeddings in G but excluding the redundant ones via automorphism checking. Similar to the previous work, we select the following applications as the representatives of graph mining and evaluate them throughout this paper.

Clique finding (CF) needs to find all complete subgraphs in the input graph G. A complete subgraph is defined as a connected subgraph where each vertex is connected to all other vertices. CF is a representative application where the patterns can be foreknown, and can thus be simply regarded as a subgraph matching problem [21], [32], [37].

Frequent subgraph mining (FSM) finds those patterns that occur a minimum number of times in the input graph G. The occurrence of a pattern is defined as the number of its matched embeddings. FSM is an example of more challenging applications where the patterns are unknown a priori. We have to enumerate both patterns and embeddings, instead of matching embeddings to the predefined patterns.

Motif counting (MC) needs to count the occurrences of all possible patterns in the input graph G. MC represents those applications that do not filter embeddings. As in the case of FSM, the patterns expected by MC are also unknown a priori.

Graph Mining Procedure. Graph mining follows an *embedding-centric model* (ECM) to express the programs

Algorithm 1: Embedding-centric model

```
   Input: $G$ – Input graph
   Input: $ITER$ – Maximum embedding size
   Output: $O$ – Output set
1  $I = \{v | v \in G\}$;
2  for ($iter = 0$; $iter < ITER$; $iter++$) do
3      $F = \varnothing$;
4      for ($e \in I$ and Aggregate_filter($e$)) do
5          $I' =$ extend($e$);
6          for ($e' \in I'$) do
7              if (non-automorphic($e'$) and Filter($e'$)) then
8                  $O = O \cup$ Process($e'$);
9                  $F = F \cup e'$;
10     $I = F$;
```

TABLE I
PRIMITIVE IMPLEMENTATIONS FOR CF, FSM, AND MC. $P(e)$ REPRESENTS THE PATTERN OF EMBEDDING e. Num($P(e)$) INDICATES THE OCCURRENCE NUMBER OF $P(e)$. $Thres$ IS A THRESHOLD DEFINED BY THE USER.

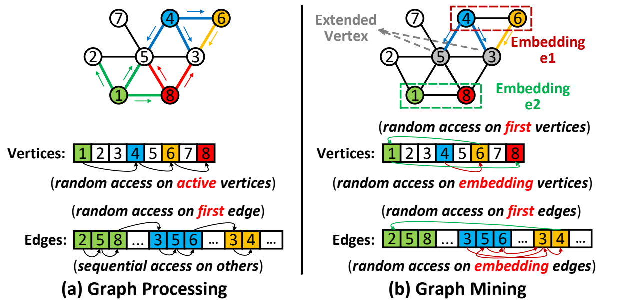
Application	**Aggregate_filter**(e)	**Filter**(e)	**Process**(e)
CF	TRUE	IsClique(e)	$(P(e), 1)$
FSM	Num($P(e)$) $\geq Thres$	TRUE	$(P(e), e)$
MC	TRUE	TRUE	$(P(e), 1)$

uniformly. Algorithm 1 shows the iterative procedure of ECM, templatized by three basic primitives: `Aggregate_filter`, `Filter`, and `Process` [38]. In the beginning, an embedding set I is initialized as consisting of each single vertex of G. For each embedding e in I, ECM first checks the validation of its pattern by invoking `Aggregate_filter` (Line 4). For example, FSM uses `Aggregate_filter` to filter embeddings whose pattern occurrences are less than the user-defined threshold. The embedding of a valid pattern is extended to generate new intermediate embeddings by adding adjacent vertices and edges (Line 5). ECM further checks the automorphism of these intermediate embeddings and invokes `Filter` to filter out embeddings unexpected by users (Line 7). Finally, the remaining embeddings are processed (Line 8) and form a new embedding set for the next iteration (Line 9).

Table I shows the typical primitive implementations for CF, FSM, and MC. These primitives receive an embedding e as input and output a boolean (`Aggregate_Filter` and `Filter`) or vector (`Process`). Let us take MC as an example. In MC, all non-automorphic embeddings must be considered for precise enumeration. Therefore, `Aggregate_Filter` and `Filter` in MC always return `TRUE`. For each non-automorphic embedding, `Process` in MC outputs its pattern and a fixed number of one, denoted as $(P(e), 1)$, to be further reduced to count the pattern occurrence.

B. Memory Accesses in Graph Mining

The memory accesses of graph mining originate from the `extend` process (Line 2 in Algorithm 1), which can lead to extraordinarily complex irregular accesses from not only vertex dimension but also edge dimension. Figure 2 exemplifies the differences in memory accesses between graph processing and graph mining for a specific iteration, where ❶, ❹, ❻, and ❽ indicate the active vertices to be processed.



Extended Vertex
Embedding e1
Embedding e2
Vertices:
Edges:
(random access on active vertices)
(random access on first edge)
(sequential access on others)
(random access on first vertices)
(random access on embedding vertices)
(random access on first edges)
(random access on embedding edges)
(a) Graph Processing
(b) Graph Mining

Fig. 2. Memory access patterns between (a) graph processing and (b) graph mining. Colored vertices represent the vertices that need to be processed in a certain iteration. In (a), the four active vertices are processed in an ascending order of vertex index. In (b), the two embeddings e_1 and e_2 will be processed.

Vertex Dimension. As shown in Figure 2(a), graph processing traverses active vertices and their neighboring vertices, which are often discontinuous, resulting in the random vertex accesses. In graph mining, the vertex access behavior is similar. For each embedding e, applying `extend` to its internal vertices will also generate random accesses on these discontinuous vertices. In Figure 2(b), extending e_1 will involve the random accesses to ❹ and ❻. Moreover, stepping into the extension for the next embedding e_2 after e_1 will generate another inter-embedding random access.

Edge Dimension. Edges are often stored together according to their source vertices. In graph processing, processing each active vertex (e.g., ❶) can therefore exhibit strong spatial locality for edge accesses. Moving to the next active vertex (e.g., ❹) can introduce a few global random accesses, which only cause a negligible impact since a graph typically has far more edges than vertices. In addition, the impact from these global random accesses can also be handled easily by edge streaming [51] or prefetching [44].

Unlike graph processing that traverses the edges of each vertex sequentially, graph mining adopts an *extend-check* edge-access model. In principle, every embedding requires to access all edges between its internal vertices and every newly-extended vertex. In this process, every time when a neighboring vertex is extended, the connectivity between this vertex and all of the embedding's vertices must be checked. As a result, substantial random edge accesses will be introduced.

Consider Figure 2(b) again, $e1$ will be first handled, starting from traversing the edges of ❹ and then selecting ❸ to extend. Afterwards, ❻ will be accessed to check its connectivity to ❸. In this process, the accesses to ❹→❸ and ❻→❸ are random. Next, ❺ will be selected until all vertices in e_1 are checked. After finishing e_1, we will proceed to handle e_2 (which has a similar procedure with e_1) until all extensions are finished. The inter-embedding switch can also introduce the random access, as shown by the green arrow drawn on the edge array.

Summary. Compared to graph processing, graph mining performs a significant number of random memory accesses on both vertex and edge data. Given N active vertices in a single iteration, graph processing makes N random vertex accesses in the worst case. However, if M embeddings (with a size of k) need to be extended in the iteration, graph mining will perform

Graphs	$\lvert V\rvert$	$\lvert E\rvert$
Citeseer [13]	3,312	4,732
P2P [24]	8,114	26,013
Astro [24]	18,772	0.2M
Mico [13]	0.1M	1.1M
Patents [16]	2.7M	14.0M

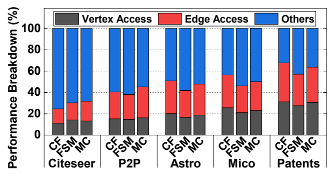


Fig. 3. Pipeline stalls due to random vertex and edge accesses. The trials run on a server: a 14-core Intel E5-2680v4 processor (with each core containing 32KB L1 cache and 256 KB L2 cache, and sharing a unified 35MB L3 cache) and a 4-channel DDR4. 'Others' represents random-access-irrelevant execution.

at least $k \times M$ random vertex accesses and $k \times M$ random edge accesses, where M is often exponential with respect to N. In graph mining, a modest graph with 0.1 million vertices can quickly generate two trillion embeddings per iteration [12], resulting in substantial random vertex and edge accesses.

Results. We have experimentally evaluated the impact of random memory accesses for CF, FSM, and MC on five real-world graphs. Intel VTune Amplifier [18] is used to count the pipeline stalls due to random vertex and edge accesses. As shown in Figure 3, graph mining applications generally suffer heavily from random accesses. When a graph (e.g., Citeseer) is so small that it can fit into the private cache, graph mining applications can take 30% of the pipeline stalls from random accesses. However, as the graph gradually exceeds the cache capacity, the number of pipeline stalls will increase quickly, by up to 67.9% for Patents as an example.

C. Pitfalls of Existing Graph Accelerators

Mostly designed for accelerating graph processing applications, existing graph accelerators [11], [17], [31], [44], [46], [49] cannot address graph mining problems efficiently. In graph processing, almost all random accesses manifest on vertex data [30], which can be relatively small (i.e., several megabytes). Therefore, a key design philosophy behind these earlier graph processing accelerators is to use a specialized on-chip memory to hold all vertex data, such that irregular vertex accesses will not result in any off-chip communications. For example, Graphicionado [17] fits all vertices into a scratchpad memory, and ForeGraph [11] employs FPGA's BRAMs.

However, graph mining problems are far more complicated due to the random accesses to both vertex data and edge data (discussed in Section II-B). Because of physical limitations, the on-chip memory is typically designed with a MB-size capacity. In this case, the on-chip memory is often too small to store all randomly-accessed data. For instance, the cache capacity of the latest Intel Xeon processors is less than 77 MB. However, today's graphs are explosively increasing and can easily have more than billions of edges [28], [29], [50]. Therefore, the design philosophy adopted in existing graph accelerators can hardly deal with the new inefficiencies arising in graph mining.

The graph partition mechanism used in previous accelerators to handle large graphs is also not helpful. In general, they partition a graph into several slices that fit in on-chip memory. However, graph mining runs on embeddings that contain several vertices with irregular IDs. Applying graph partitioning to graph mining can cause the data requested by an embedding to span

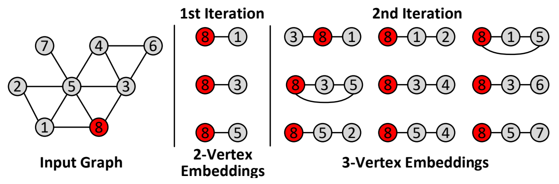
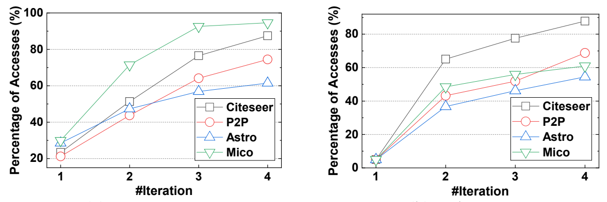


Fig. 4. An example of extension locality in graph mining: starting from exploring an initial embedding {⑧}. ⑧ can be accessed more frequently over iteration but existing graph accelerators barely exploit this due to the featured memory access pattern in graph mining.



(a) Vertex access (b) Edge access

Fig. 5. Extension locality of the memory accesses to the top 5% (a) vertices and (b) edges as the embedding size increases over iteration for MC. Patents and the results after the 5th iterations are excluded due to more than hundreds of trillions of embeddings that are too expensive to trace and analyze offline.

many slices. We have to replace on-chip slices frequently, resulting in significantly more memory accesses, compared to directly accessing the whole graph instead.

D. Overcoming Inefficiencies

In this work, we observe that not all random memory accesses in graph mining are created equally. That is, a few vertices and edges induce a large number of random accesses, while many result in a few. The underlying reason is co-determined by the skewed feature of real-world graphs and the extension nature of graph mining applications. We can develop some basic understanding here intuitively. Real-world graphs often have the hallmark property of power-law degree distribution [14]. Therefore, some high-degree vertices are more likely to be the extended candidates of an embedding. As a result, these vertices can quickly be included in the most embeddings as the size of embedding increases over iterations. Extending the embeddings containing these vertices would induce a very large number of random accesses to themselves and their edges. Thus, we have the following observation:

Observation: *Graph mining shows significant **extension locality**, meaning that most random memory accesses root in accessing a small fraction of the (vertex and edge) data.*

Figure 4 illustrates an example for understanding this observation. We take one initial embedding {⑧} as an example and consider only its vertex accesses for illustration purpose. Its edge accesses are similar. On the whole, the graph has 12 edges, and ⑧ has 3 edges. Hence, it is extended in 3 out of 12 two-vertex embeddings in the first iteration. Suppose each vertex in an embedding can issue one memory request. ⑧ will be accessed 3 times with an access frequency of 12.5% ($\frac{3}{12\times2}$). When stepping into the second iteration, ⑧ will be accessed 9 times with an access frequency of 13.6% ($\frac{9}{22\times3}$). We see that memory accesses to ⑧ become more frequent.

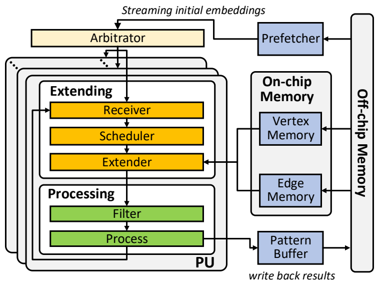


Fig. 6. Architecture of GRAMER

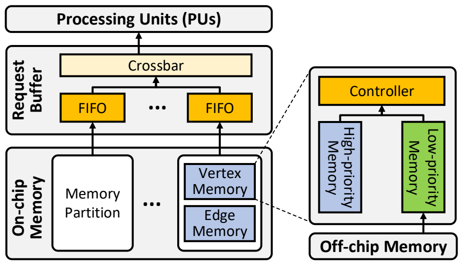


Fig. 7. On-chip memory hierarchy

Figure 5 further investigates the memory accesses[1] to the top 5% vertices and edges in each iteration for MC. The access frequency of the top 5% vertices can be less than 29.93% in the 1st iteration (with all embeddings sized by 2 vertices), but it quickly takes more than 43.73% in the 2nd iteration and reaches up to 94.57% (for Mico) in the 4th iteration. The top 5% edges start from a fixed access frequency of 5% since each edge is accessed only once for 2-vertex embeddings. As the iteration goes, the access frequency also increases significantly (up to 87.80% for Citeseer as an example). The results of CF and FSM are similar because graph mining applications generally prefer to find frequent or highly-connected patterns, where low-degree vertices are more likely to be filtered.

Nevertheless, exploiting the extension locality in existing graph accelerators remains challenging. First, accurately estimating the access frequency of every vertex and edge data is difficult. This is closely related to the degrees of not only the current vertices but also their future neighbors. A low-degree vertex connected with many high-degree vertices might also be frequently accessed. Second, the overwhelming intermediate results incur substantial off-chip communications. A modest graph with 0.1 million vertices can quickly generate trillions of embeddings because of the combinatorial explosion [12]. Iteratively accessing them in off-chip memory can cause a significant slowdown. To solve these challenges, we propose a novel accelerator GRAMER with specialized hardware designs.

III. THE Gramer ARCHITECTURE

Figure 6 depicts the architecture of GRAMER. For programmability, GRAMER follows the ECM in Algorithm 1 and transparently exploits the extension locality. The key contribution of GRAMER lies in a *locality-aware on-chip memory* that enables accurate estimation of access frequency, and *pipelined processing units* (PUs) that minimize off-chip communications and maximize computational parallelism. We detail each component of GRAMER as follows:

Prefetcher. The *Prefetcher* performs next-line prefetches to transfer the initial embeddings from the off-chip memory to the PUs. We use an *Arbitrator* to dispatch each initial embedding to a PU in a round-robin manner.

Extending. Extending module contains three core parts. The *Receiver* recursively reads initial embeddings from the off-chip memory or intermediate embeddings from the *Processing* module. The *Scheduler* validates the received embeddings and schedules one valid embedding per cycle for the extension of *Extender* that will generate memory requests. We apply a *depth-first-search* (DFS) model and propose specialized pipeline designs for embedding extension to avoid accessing intermediate embeddings from the off-chip memory (Section V).

On-Chip Memory. On-chip memory contains a vertex memory and an edge memory to isolate vertex and edge accesses. This can effectively avoid frequent data thrashing between vertices and edges. To exploit the extension locality, both vertex memory and edge memory are divided further into a high-priority memory and low-priority memory. We propose cost-efficient priority-based heuristics, which can accurately identify valuable data to store in the high-priority memory, and also specialize a new replacement policy to integrate our heuristics efficiently with hardware implementations (Section IV).

Processing. After receiving an embedding from the *Extender*, the *Filter* will check its automorphism based on a canonicality mechanism [38] and its validation by user-defined primitives. The remaining embeddings are processed by the *Process* to generate output results and also sent back to the *Receiver* for further extension.

Pattern Buffer. The user-desired output results obtained from the *Process* module will be stored in a *Pattern buffer* and further written back sequentially to the off-chip memory.

IV. MEMORY DESIGNS

In this section, we present our memory design to enable exploiting the extension locality in graph mining with a new memory hierarchy and hardware implementations.

A. Locality-Aware Memory Hierarchy

Figure 7 shows the hierarchy of on-chip memory, organized with eight partitions for handling multiple memory requests simultaneously. Unlike graph processing, graph mining does not change the vertex and edge data, and thus, GRAMER does not need to maintain a consistency protocol. Each memory partition is shared by all PUs and is separated with the two parts: a vertex memory and an edge memory. The isolation of vertex access and edge access can effectively avoid the potential access conflicts and data thrashing between them. When a memory request is generated by a PU, it will be assigned to a FIFO in a request buffer through an 8-radix

[1]We trace all memory requests in each iteration, and then rank each vertex and edge according to the number of their memory requests through an offline analysis.

crossbar switch, and then sent to the corresponding vertex or edge memory as per the request type.

To exploit the extension locality, GRAMER adopts a differential memory management on the graph data, which is classified into the high priority data and the low-priority data. The high priority data is often frequently accessed but typically small in quantity (e.g., the top 5% vertices and edges in Figure 5), and we propose to permanently store it in the on-chip memory such that most off-chip communications arising from it can be removed. For the most low-priority data that is less frequently accessed, it is dynamically maintained but implemented with a new replacement policy in order to maximize the value of the extension locality in graph mining. Following this design philosophy, each vertex and edge memory is further built with the components below:

- *Controller.* It is used for dispatching a memory request to a high- or low-priority memory as per its data priority. Then, the requested data will be sent back to a corresponding FIFO.
- *High-priority Memory.* It is used for permanently residing the high-priority data without data eviction. Therefore, it can be simply implemented as a fast scratchpad memory.
- *Low-priority Memory.* It is used for dynamically maintaining the low-priority data with a new replacement policy that preserves the extension locality of the low-priority data.

B. High-priority Memory

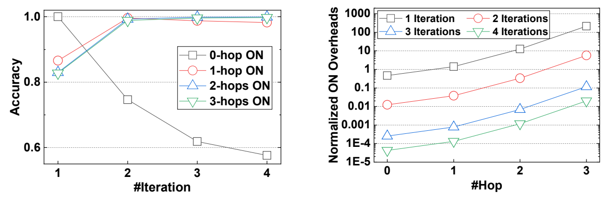
The key to the high-priority memory is to accurately identify which data has a high priority for the permanent residence.

Priority Definition. An intuition to solve this problem is to precisely count the *occurrence number* (ON) of the graph data. For illustration purposes, we take vertex access as an example. Edge access is similar. In general, all vertices of the embeddings in an iteration are accessed equally (as discussed in Section II-D). Let us assume that the number of times each vertex accessed in an embedding is c. The ON of a vertex v can be described as the number of embeddings that contain it. Without considering redundant (automorphic) embeddings, the number of embeddings in G can be simplified as a combination of all vertices. In a $(k+1)$-th iteration, where the size of input embedding is $k+1$, the ON of v can be defined as:

$$\mathrm{ON}_k = \prod_{dist=0}^{k} \sum_{v' \in nghbr(dist,v)} Deg(v') \cdot c \tag{1}$$

where $nghbr(dist, v)$ represents a set of vertices that have the distance of $dist$ away from v. $Deg(v')$ indicates the degree of a vertex v'. By ranking all vertices by their ONs, we can obtain the priority of a vertex as per its ON. The larger the ON is, the higher the priority of the corresponding vertex is.

Exhaustive Computation Is Expensive. In Equation (1), the ON of a vertex is determined by not only its degree but also the degree(s) of its k-hop neighbors. We can traverse the graph (e.g., using BFS) on every vertex to obtain their k-hop vertex sets and redundancy embeddings, but it is too expensive and needs multi-time executions on the graph. For instance, we benchmark MC on P2P on a server (described in Section II-B).



(a) Accuracy (b) ON-computation overheads

Fig. 8. Characterization on (a) Accuracy and (b) ON-computation overheads for MC on P2P with different number of hops allowed to be detected. In (a), 'Accuracy' represents the proportion of vertices that can fall in the ideal 5% top vertex set in Figure 5 for each iteration. In (b), all results are normalized to the total running time.

The ON-computation can consume 210.34 seconds, nearly the same as the normal execution. Although simply using only the vertex degree (i.e., discarding its k-hop vertex degrees) can reduce the overheads significantly, the identification accuracy can also be considerably impacted.

1-Hop Heuristic Computation. Figure 8 shows a sensitivity study on the accuracy and the ON-computation overheads as the number of hops varies for MC on P2P. As observed, *considering the degree of a vertex and its one-hop neighbors' degrees ($k=1$) is sufficient for the* ON-*computation to preserve the most accuracy while incurring low overheads.* Looking at the 1-hop ON in Figure 8(a), the accuracy is over 80% for all iterations and can be improved as k increases further. The reason is that high-degree vertices and their neighbors can connect most vertices in a graph. The degrees of other vertices contribute little to the value of ON. In addition, the 1-hop ON-computation overheads are reasonable, while a larger value of k will significantly increase the overheads by up to 8500$\times$ ($k=3$). Therefore, ON_1 (i.e., 1-hop ON) computation is expected to yield a cost-efficient trade-off.

By sorting the ON_1 of all vertices in descending order, we can determine the priority of the vertex and edge data easily according to a threshold τ (with the value of τ discussed in Section VI). The vertices with the ON_1 in the top τ range (i.e., $\{v | Rank(\mathrm{ON}_1(v)) \leq \tau \times |Vertex|\}$) are considered as the high priority ones while the rest are classified as being the low priority ones. An edge access gets started from an embedding's vertex. This allows us to compute the ON_1 of an edge based on the ON_1 value of its source vertex, i.e., $\mathrm{ON}_1(edge) = \mathrm{ON}_1(v_{src})$. Similarly, the edges with the ON_1 in the top τ range are high priority ones, while others are the low priority ones.

C. Low-priority Memory

In a dynamic environment, the data must be replaced due to the limited low-priority memory size. Typical designs use the recency of the data to determine the victims [7], [10], [19], [20], [41]. These studies predict the future access frequency of the data according to their history access information, which has shown to be effective for traditional applications with regular access patterns. However, for graph mining applications with extremely irregular features, these recency-based methods may destroy the extension locality of some low-priority data that is not frequent recently but frequent globally.

Locality-Preserved Replacement Policy. We introduce ON_1 into the recency-based method to balance the extension locality and the temporal locality in the low-priority memory. Let V_{Low} be the set of low priority vertices (i.e., $\{v|Rank(\mathrm{ON}_1(v)) > \tau \times |Vertex|\}$), and $Rec(v)$ be the recency value of the vertex v (i.e., the number of accesses after the reference to v), the low-priority memory employs a linear function of the ON_1 and the recency to replace the low-priority vertex with the largest value preferentially:

$$victim = \arg\max_{v}\{Rank(\mathrm{ON}_1(v)) + \lambda Rec(v)\}, v \in V_{Low} \quad (2)$$

where λ is a balancing factor. If $\lambda = 0$, the low-priority memory can be understood as being equivalently a high-priority memory without data eviction. If λ is large enough, the replacement policy will become a classical *least-recently-used* (LRU) policy. Finding a victim for the low-priority edges on the low-priority memory is similar.

Hardware Implementation. As shown in Equation (2), the key to implement the replacement policy is to compute $Rank(\mathrm{ON}_1(v))$. When a vertex is requested, GRAMER needs its rank value to examine the priority and further find a victim in the low-priority memory if it is the low-priority data. However, computing $Rank(\mathrm{ON}_1(v))$ is notoriously non-trivial. Performing it at runtime is too costly to implement on-chip, while storing the rank value accompanied with graph data in the off-chip memory may double memory overheads.

We propose to use the graph reordering to enable a fast computation on $Rank(\mathrm{ON}_1(v))$ at runtime. In graph mining, reading the embedding structure must be prior to its extension. That is, the vertex IDs of an embedding can be known before its random accesses get started formally. Therefore, we exploit a basic idea of using the ID to represent the rank for each data. However, the vertex ID order of a graph is often not consistent with its rank order. Thus, we propose to reorder the vertex indices in ascending order of $Rank(\mathrm{ON}_1(v))$ for all vertices in the input graph. With the graph reordering, the rank value of a vertex (or an edge) can be easily obtained at runtime by simply extracting its corresponding (source) vertex IDs.

The main overhead of this process lies in ON_1-based vertex sorting, which is similar to the offline analysis in Figure 8(b). We see that the reordering overheads for the embeddings[2] can be often less than 3% of the execution time, yielding a fast and lightweight procedure with negligible overheads.

V. PIPELINING PARALLELIZATION

This section presents the pipeline specialization of PUs for minimizing off-chip overheads and maximizing parallelism.

A. Computational Model

Existing graph mining systems typically adopt a *breadth-first-search* (BFS) style [38], [42] to enumerate the embeddings. BFS module iteratively adds one neighbor in each synchronization step. For example, when we search all embeddings with 3 vertices in an input graph, the BFS model first enumerates all 2-vertex embeddings and writes them back to the off-chip memory. In the next iteration, the BFS model accesses all 2-vertex embeddings in the off-chip memory and further extends them to get 3-vertex embeddings.

[2]The overhead for the 2-vertex embeddings is not counted in the graph reordering method since all 2-vertex embeddings always have the same pattern and only patterns with more than three vertices are of particular interest for most graph mining applications [2], [13].

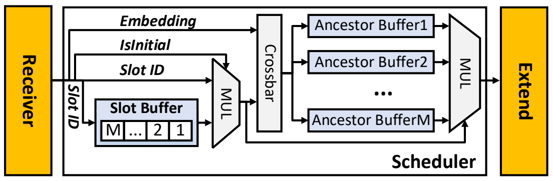


Fig. 9. Pipelined implementation of the *Scheduler*. Each slot ID obtained from the slot buffer represents an individual pipeline used for exploring the embeddings from an initial embedding.

However, the BFS model is not efficient for a graph mining accelerator. Graph mining generally suffers from the combinatorial explosion with a significant number of intermediate embeddings. Frequently accessing these intermediate embeddings will waste significant memory bandwidth, which will further reduce the computation parallelism. Moreover, storing these intermediate embeddings requires an off-chip memory capacity far beyond what an accelerator can afford.

In this work, we adopt a *depth-first-search* (DFS) model proposed in a recent study [12] to avoid writing back intermediate embeddings. In the DFS model, every initial embedding must be recursively extended until the completion. The extension of each initial embeddings will continue until a maximum depth is reached. When reaching the maximum depth, it will *traceback* (i.e., return) to the last valid embedding for further extension. For example, when searching 3-vertex embeddings, the DFS model first selects one initial embedding and iteratively extends one vertex for generating intermediate embeddings. When the embedding size reaches three, it tracebacks to the last 2-vertex embedding by removing the last vertex added and continues extension. After all embeddings having been extended, it will move to another initial embedding for further extension. By avoiding writing back the intermediate states, the DFS model significantly saves the space and memory bandwidth.

B. Maximizing Pipeline Parallelism

In GRAMER, each PU can handle multiple embeddings simultaneously for the pipelining parallelism. In the DFS model, when an extension path has reached its maximum depth, a traceback to the last valid embedding will be performed to find another branch for the subsequent extension. Therefore, the extension states of all vertices in *ancestor embeddings* (i.e., embeddings in the current extension path) must be stored to ensure accurate traceback. These intermediate vertex extension states of all PUs, often in tens of kilobytes in total, pass through all the pipeline stages. This significantly limits the computation parallelism due to substantial hardware overheads, which further decreases the overall performance.

Pipeline Specialization. In actuality, the vertex extension states of these pipelined embeddings are used only in the *Scheduler* stage (Figure 6) for deciding which candidate can

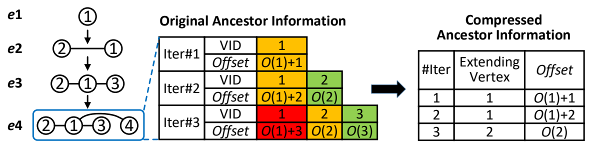


Fig. 10. Ancestor information compression. Only the information for vertex extension needs to be maintained (i.e., orange cells). The information in the green cells can be removed as their vertices have not been extended yet. The information in the red cells can be omitted due to the extension completion of ①. The *Offset* represents the offset address in the edge array for a vertex. $O(i)$ is the offset of the first edge of the vertex i.

be extended. We propose to integrate the specialized *ancestor buffers* into the *Scheduler* to store the information of the current extension path for all processing embeddings, as shown in Figure 9. However, the challenges remain. First, the scheduling order of the embeddings is non-determinate. The first scheduled embedding is not necessarily the first one finished. Accurately finding the ancestor embedding in the buffer is non-trivial. Second, the buffer may still increase the access latency.

To make a precise identification, we enforce the scheduling order to ensure each processing embedding can be easily identified by its related initial embedding. The design is based on the fact that changing the extension order of the edges has no effect on the final results. We consider all the embeddings extended from a same initial embedding as an individual pipeline, and handle them in sequence. More specifically, each PU allows only the embeddings originated from different initial embeddings to be processed simultaneously. Every time when an intermediate embedding is received, it will be scheduled to extend only on embedding. No further extension on this embedding will be made until a traceback is triggered.

We further architect a specialized pipeline structure for the *Scheduler* to enable the above index-based runtime scheduling. As shown in Figure 9, a *slot buffer* is initialized by some unique slot IDs. Whenever the *Scheduler* receives a new initial embedding, GRAMER will associate it with an ID i from the slot buffer, and store its extension states into i-th *ancestor buffer* (organized as a LIFO queue). This slot ID is shared by the initial embedding and all the embeddings extended from it. When a particular embedding e reaches the *Scheduler*, GRAMER can quickly locate the corresponding ancestor buffer by its slot ID. If e performs a normal extension, e will be enqueued into the tail. If e induces a traceback, an embedding will be dequeued out of the ancestor buffer. Note that the *Scheduler* allocates at most one slot ID per cycle. The embeddings with different slot IDs can be processed in a pipelined manner to exploit all hardware modules of a PU.

Compacting Ancestor Buffers. As discussed earlier, only one vertex of an embedding can be extended in each exploration step. Thus, the storage for the ancestor embedding of an embedding e can be greatly reduced by enforcing the exploration order of the vertices in e as per the order of vertices joined in e. Specifically, a vertex earlier joined in e will be preferentially extended compared to the later one(s) during the extension of e. This makes it possible to merely store the information of the *extending* vertex (rather than all vertices) for each ancestor embedding of e, since the extension of other vertices in e is either finished or not yet started.

Figure 10 depicts an example of storage compaction, where each original ancestor embedding is based on the CSR representation [29], [35], [51]. Assume that ① is connected with ②, ③, and ④, the initial embedding $e1$ (i.e., {①}) can gradually extend the intermediate embeddings $e2$, $e3$, and $e4$. In the original design, $e4$ needs to store all the information from three ancestors, i.e., the offsets of all vertices of three ancestor embeddings. In GRAMER, the extension order of the vertices is specified such that ① is always scheduled before ②. In the 1st and 2nd iterations, GRAMER extends ① to ② and ③, respectively. In this case, we only need to store the information (**in the orange cells**) of ① since the other vertices have not been extended yet. In the 3rd iteration, GRAMER moves to extend ② and stores its offset (i.e., $O(2)$). At this point, the information (**in the red cell**) of ① can be discarded since all of its edges have been finished and will never be explored again. ③ is not recorded (**in the green cells**) because it has not yet been extended. In summary, for an embedding, we only need to store the information on *extending* vertex for each of its ancestor embeddings, i.e., vertex ID and its offset only.

C. Improving Load Balance

Each PU has the responsibility of simultaneously handling multiple initial embeddings that may evolve a significantly different number of workloads that need to be processed, leading to the potential load imbalance. For the workloads in a PU, the power-law degree distribution of the real-world graphs can cause some assigned initial embeddings to generate more workloads than others. Therefore, the slot utilization of the PU can drop quickly as some slots finish their tasks early, while others still run as stragglers for a long time.

To reduce the load imbalance in each PU, we adopt a work-stealing mechanism to allow the idle slots to get workloads from the busy ones. To avoid costly broadcasting or re-shuffling, we specialize a *stealing buffer* in the *Scheduler* for dynamically maintaining the workloads to be stolen. Every time when the *Scheduler* receives an (initial or intermediate) embedding, the slot buffer will be checked to see if it is empty. If not, the slot ID of the received embedding will enqueue the stealing buffer, representing an embedding to be extended. When the *Scheduler* receives no embedding, a slot ID i will be popped out of the stealing buffer. If the slot i does not finish the task, the PU will steal an embedding e from the i-th ancestor buffer, and allocate an idle slot in the slot buffer to start exploring e. Compared with the linear feedback shift register that randomly selects a (probably idle) slot [8], stealing buffer can always ensure accurate stealing to a busy slot.

In essence, for the workloads across the PUs, their parallel executions can be effectively balanced using adaptive dispatching of the initial embeddings. We have simply implemented the Arbitrator (Figure 6) by dispatching in a round-robin manner.

VI. Evaluation

This section evaluates the efficiency and effectiveness of GRAMER against the state-of-the-art.

TABLE II
RESOURCE UTILIZATION AND CLOCK RATE

	CF	FSM	MC
LUT	25.39%	25.53%	25.43%
Register	13.06%	13.13%	13.10%
BRAM	65.69%	65.70%	65.70%
Clock Rate	213MHz	207MHz	207MHz

A. *Experimental Setup*

GRAMER Settings. We have implemented GRAMER on a Xilinx Alveo U250 accelerator card, which includes a XCU250-2LFIGD2104E FPGA chip and four 16GB DDR4 memories. The FPGA chip provides 1.68M LUTs, 3.37M registers, and 11.8MB BRAM resources. GRAMER contains 8 PUs, each of which has a slot buffer, a stealing buffer, and 16 ancestor buffers. Each slot buffer can allocate 16 slot IDs. Thus, GRAMER can simultaneously process a maximum of 128 (8×16) embeddings. The stealing buffer has the same size as the slot buffer, and can support 16 stealing candidates at most. Each ancestor buffer can store 16 ancestor embeddings of an embedding, and thus can support the extension with a maximum depth of 16.

The low-priority memory is implemented as a four-way set-associative cache, which has the same capacity of high-priority memory used for storing all high-priority data. As for τ, which is used for classifying the priority of graph data (Section IV-B), GRAMER uses a simple equation $MIN(50\%, \frac{|Memory|}{2\times(|V|+|E|)})$ to calculate it for different input graphs uniformly. For example, τ is set to 50% to reap an optimal performance if the whole graph can fit into the on-chip memory. A sensitivity study of τ is also made in Section VI-D. λ is set as 1 by default.

Table II shows the clock rate and resource utilization obtained by Xilinx SDAccel 2019.2. In our evaluation, we conservatively use 200MHz to preserve the correctness. The on-chip memory is implemented by using the BRAM resources. For different graph mining applications, we use the same capacity of on-chip memory to manage the vertex and edge data. MC and FSM consume slightly more resources because they need to enumerate both patterns and embeddings.

CPU Baselines. We compare GRAMER with two state-of-the-art CPU-based graph mining frameworks – RStream [42] and Fractal [12]. RStream is a disk-based single machine system that adopts the BFS model. Fractal is a DFS-based distributed system, and we benchmark it with its single machine version. Both RStream and Fractal run on a server with a 14-core Intel E5-2680v4 processor, 128GB DRAM, and 1TB SSD.

Benchmarks and Datasets. We consider three representative graph mining applications (i.e., CF, FSM, and MC) described in Section II-A, and evaluate them on different scales of graphs. In addition to the five real-world graphs (shown in Figure 3), we also use YT [9] (4.58M vertices and 43.96M edges) and LJ [24] (4.85M vertices and 69.0M edges) in our evaluation. Citeseer and P2P are the representatives of small-sized graphs. Astro and Mico represent medium-size graphs. Patents, YT, and LJ are selected as large-sized graphs. All graphs are considered undirected and stored in the CSR [35].

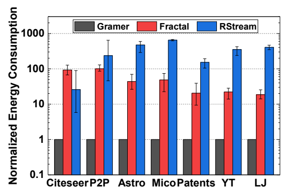
TABLE III
RUNNING TIME (IN SECONDS) OF GRAMER AGAINST FRACTAL AND RSTREAM. k-CF INDICATES TO FIND k-VERTEX COMPLETE GRAPHS. k-MC COUNTS THE OCCURRENCE TIMES OF k-VERTEX PATTERNS. FSM-k FINDS THE 3-VERTEX PATTERNS THAT HAVE OCCURRED AT LEAST k TIMES. 'N/A' DENOTES THAT THE SYSTEM RUNS OUT OF THE DISK. '-' SHOWS THAT THE APPLICATION DOES NOT BE FINISHED WITHIN 1 HOURS.

		Gramer	Fractal	RStream
3-CF	Citeseer	**0.0099**	0.15	0.011
	P2P	**0.010**	0.19	0.088
	Astro	**0.028**	0.35	1.56
	Mico	**0.11**	1.24	13.07
	Patents	**3.09**	5.56	62.34
	YT	**13.01**	34.71	598.10
	LJ	**17.81**	48.44	1188.86
4-CF	Citeseer	**0.010**	0.16	0.020
	P2P	**0.011**	0.21	0.10
	Astro	**0.27**	1.55	21.99
	Mico	**6.86**	30.64	891.44
	Patents	**3.74**	7.81	114.78
	YT	**17.30**	65.14	1301.97
	LJ	**30.89**	102.87	2761.38
5-CF	Citeseer	**0.011**	0.17	0.023
	P2P	**0.012**	0.23	0.129
	Astro	**1.46**	7.37	138.57
	Mico	**270.41**	1171.47	N/A
	Patents	**4.06**	9.63	150.53
	YT	**24.27**	97.86	1970.34
	LJ	**52.89**	179.40	-
3-MC	Citeseer	**0.031**	0.72	0.094
	P2P	**0.033**	0.82	1.90
	Astro	**0.11**	1.48	11.87
	Mico	**0.36**	4.40	-
	Patents	**4.17**	24.9	-
	YT	**16.25**	87.98	N/A
	LJ	**29.68**	144.74	N/A
4-MC	Citeseer	**0.039**	0.95	0.17
	P2P	**0.093**	1.57	5.83
	Astro	**8.00**	47.28	-
	Mico	**45.22**	641.89	-
	Patents	**103.82**	778.02	-
	YT	**931.11**	-	N/A
	LJ	**1553.87**	-	N/A
FSM 2K	Citeseer	**0.021**	0.27	0.36
	P2P	**0.045**	0.74	5.56
	Astro	**2.27**	17.52	260.13
	Mico	**132.52**	1258.70	-
FSM 20K	Patents	**1079.90**	-	N/A
FSM 250K	YT	**297.64**	1617.56	N/A
	LJ	**913.73**	-	N/A

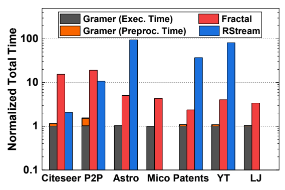
B. *Compared with Graph Mining Systems*

Performance. Table III depicts the running time of GRAMER against RStream and Fractal by benchmarking eight graph mining application variants. All graph datasets are evaluated with the same setting for different applications, except for FSM. This is because that small thresholds will take too long running time for large graphs due to a huge number of intermediate results, while large thresholds run too fast for small graphs with few expected patterns.

In our evaluation, we start to collect the running time once the input graph is loaded to the memory of server. The running time of GRAMER includes the FPGA setup time and data transfer

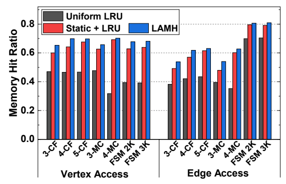


(a) Energy consumption

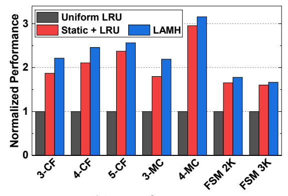


(b) Total time

Fig. 11. Energy and total time of GRAMER (with the preprocessing overheads included) against state-of-the-art. All results are normalized to (a) the energy consumption and (b) the execution time of GRAMER.



(a) On-chip memory hit ratio



(b) Performance

Fig. 12. The hit ratio and performance of the LAMH against two baselines (on P2P): a LRU-based uniform cache hierarchy and a LAMH-based hierarchy with the low-priority memory using an LRU replacement policy

overheads between CPU and FPGA, because we need to load the input graph to the memory of FPGA before processing. For Fractal, since it is built upon Spark that can often involve significant setup overheads, e.g., task partition and worker registration [48], we evaluate it with only the execution time.

Gramer vs. Fractal. Although these expensive setup overheads are excluded for Fractal, GRAMER can still outperform Fractal significantly by $1.80\times \sim 24.85\times$. To be specific, for small-sized graphs, Citeseer and P2P, the number of their possible embeddings is quite limited. Therefore, the initialization and multi-thread management overheads under CPU would dominate the overall performance of Fractal, leading to a significant performance gap between Fractal and GRAMER with the speedups of $12.86\times \sim 24.85\times$. When relatively-large graphs are processed, these non-execution overheads can be ignored. Our memory system and pipeline specialization become effective in reducing the impact of random accesses with better parallelism. As a result, GRAMER achieves speedups of $4.33\times \sim 14.20\times$ and $1.80\times \sim 7.50\times$ on medium-sized graph and large-sized graph, respectively.

Gramer vs. RStream. The speedups achieved by GRAMER vary significantly for different applications. For CF-variant applications on small-sized graphs that have small intermediate results with negligible disk accesses, GRAMER can offer speedups of $1.11\times \sim 10.75\times$. In particular, in the case of a large number of intermediate results, the speedups can be significantly improved by up to $129.95\times$ (4-CF on Mico). In this case, RStream needs to iteratively store the intermediate data with expensive disk accesses while the DFS model specialized for the PUs of GRAMER keeps consuming the intermediate data of each embedding until the extension has finished. Thus, GRAMER does not need to write back the intermediate embeddings, avoiding disk accesses.

Energy Savings. Figure 11(a) shows the average energy results when running different applications on GRAMER, RStream, and Fractal. The two error bars for each graph represent the maximal and minimal normalized energy consumption, respectively. In order to make an apples-to-apples comparison, we mainly consider the on-chip energy results of the FPGA and the CPU, exclusive of the energy consumption from DRAM accesses. GRAMER is measured using Xilinx Vivado-2019.1 under a 100% toggle rate for all on-chip logics. For the CPU baselines, we use Thermal Design Power as their energy at full capacity. Compared to Fractal and RStream, GRAMER reduces the energy consumption by $9.40\times \sim 129.72\times$ and $5.79\times \sim 678.34\times$, respectively.

Preprocessing Overheads. GRAMER relies on the graph reordering to compute $Rank(\mathrm{ON}_1(v))$ efficiently, introducing some preprocessing overheads. Figure 11(b) further depicts the performance of GRAMER with the preprocessing overheads included for 5-CF. Other applications can give rise to similar results. The preprocessing (Section IV-C) also runs on the same server as the CPU baselines. For small graphs, Citeseer and P2P, their preprocessing overheads can be as large as 55.04% of the execution time. This is because of their extremely-low execution time (e.g., 11ms for Citeseer) caused by very few intermediate results. Despite these, the preprocessing steps for these small graphs remain very fast within sub-seconds (e.g., 1.73ms for Citeseer), which are reasonable in practice. For medium and large graphs, the preprocessing proportion will become small. For instance, the preprocessing overheads of Mico can be less than 3% of the execution time. In summary, GRAMER offers significant benefits over the state-of-the-art even if the preprocessing overheads are included.

C. Effectiveness of GRAMER *Designs*

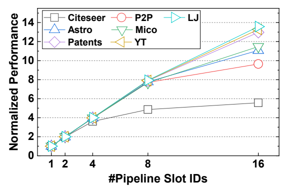
Locality-Aware Memory Hierarchy (LAMH). We investigate the effectiveness of our LAHM by comparing it with two baselines: **(1) Uniform LRU**: a uniform four-way set-associate cache hierarchy using the traditional LRU replacement policy [7], [41]. The cache has the same capacity as LAMH for making an apples-to-apples comparison, and **(2) Static + LRU:** a similar memory hierarchy with LAMH but the low-priority memory adopts the LRU (rather than the locality-preserved one in Section IV-C) replacement policy. Figure 12 shows the results in terms of the hit ratio and performance. To better understand the effectiveness of LAMH, we store only a small portion (i.e., 10%) of vertex and edge data in the on-chip memory for both baselines and LAMH.

Compared to uniform LRU, Static+LRU improves the hit ratios of vertex access and edge access by $12.96\% \sim 37.44\%$ and $8.42\% \sim 24.94\%$, respectively, showing the effectiveness of the high-priority memory in exploiting the extension locality. This therefore yields the speedups of $1.60\times \sim 2.95\times$. Compared to Static+LRU, LAMH improves the hit ratios of vertex access and edge access further by $1.01\% \sim 5.67\%$ and $1.11\% \sim 6.10\%$, respectively. This is achieved due to an awareness of the extension locality in our replacement implementation, further improving the performance by $1.06\times \sim 1.39\times$.

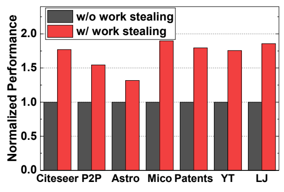
In particular, as the embedding size increases over iterations, both Static+LRU and LAMH can constantly improve the on-

TABLE IV
CLOCK RATE OF GRAMER WITH AND WITHOUT ANCESTOR BUFFERS AND COMPACTION MECHANISM

	CF	**FSM**	**MC**
w/o AB	80MHz	78MHz	78MHz
w/ AB	97MHz	96MHz	96MHz
w/ AB + Compaction	213MHz	207MHz	207MHz



(a) Varying pipeline slots



(b) Work stealing

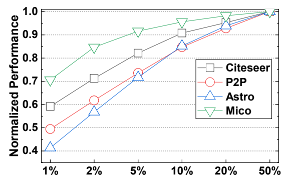
Fig. 13. Performance of GRAMER on 5-FC by (a) varying the maximum number of pipeline slots, and (b) applying the work-stealing mechanism

chip hit ratio while the LRU-based cache hierarchy improves little or even sometimes degrades the hit ratio. For instance, the hit ratios of vertex access under the LAMH for 3-MC and 4-MC can be improved from 65.74% to 70.24% while those under the uniform LRU are reduced from 47.77% to 31.81%. The reason is simple. A large embedding size often implies more extension locality opportunities that LAMH can exploit. However, the LRU-based cache is unaware of high-priority data that would be therefore frequently evicted.

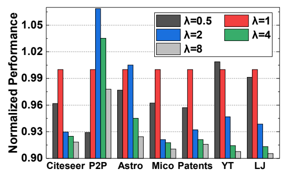
Ancestor Buffers (AB). Table IV depicts the effect of the ancestor buffers and their compaction mechanism. We see that applying the specialized ancestor buffers for storing the ancestor information of the embeddings can reduce the pipeline latency significantly by improving the maximal clock rate by 23.08%. Applying the compaction technique can further improve the clock rate by 115.64%.

PU Pipelining. We further characterize the performance of GRAMER by adjusting the maximum number of pipeline slots in the slot buffer. Figure 13(a) shows the results by varying the number of slots from 1 to 16. When the number of slot IDs is less than 8, GRAMER can achieve the nearly linear performance improvements for almost all graphs (except Citeseer), showing ideal pipelining parallelism. However, when the number of slot IDs is increased further from 8 to 16, the performance improvements decrease gradually. The reason lies in the increased memory pressure. More slot IDs represent more simultaneous memory requests to each memory partition, causing potential conflict overheads.

Work Stealing. Figure 13(b) further shows the performance of GRAMER with and without applying the work stealing (Section V-C). Compared to the PU without applying work stealing, work stealing can further offer the speedups of $1.32\times \sim 1.90\times$. In particular, Mico has the highest skewness with respect to memory accesses, and consequently, the workload amount (derived from different initial embeddings) for different pipelines can be significantly different, causing severe load imbalance. Therefore, Mico can benefit most from the work stealing over other graphs.



(a) Varying τ



(b) Varying λ

Fig. 14. Sensitivity analysis on τ and λ for 5-CF. Results are normalized to (a) $\tau = 50\%$ and (b) $\lambda = 1$.

D. Sensitivity Analysis

Sensitivity to τ. We first investigate the performance sensitivity of GRAMER with different values of τ ranging from 1% to 50%. The low-priority memory is set with the same size as the high-priority memory. Therefore, $\tau = 50\%$ represents an ideal case that all graph data can be stored on-chip. Figure 14(a) shows the results for 5-CF, where Patents, Youtube, and LiveJournal are not tested due to the limited BRAM capacity. We see that $\tau = 5\%$ can achieve $71.69\% \sim 91.59\%$ of the ideal case since most memory accesses are induced by the top 5% vertices and edges as shown in Figure 5. Increasing τ further could capture little high-priority data for reaping the benefits. However, decreasing τ will miss the high-priority data significantly, thus incurring considerable performance loss.

Sensitivity to λ. Figure 14(b) further characterizes the performance of GRAMER with different values of λ from 0.5 to 8. We see that the recency does not significantly influence the overall performance, which increases slightly from $0.91\times$ to $1.07\times$ as λ increases. This is because the data rarely-accessed but frequently-reused in a short time often incurs a few memory accesses. Thus, increasing λ to exploit its temporal locality contributes little to the overall improvements.

VII. RELATED WORK

Graph Processing Systems. Pioneered by Pregel [29], a large number of graph processing systems have been proposed to parallelize graph applications efficiently in the past decade. Many of these systems [3], [14], [15], [23], [28], [35], [50], [51] target at achieving balanced, memory-efficient computation for vertex-centric programs. However, deploying graph mining applications on these systems is non-trivial due to a lack of expressiveness of the vertex-centric model for the embedding structure. Although alternatives [36], [39], [47] adopt a subgraph-centric model for reducing the communications across the vertices, applying such a model designed for a fixed-size subgraph to graph mining is often ineffective in enumerating the embeddings with varying sizes across the iterations.

Specialized Algorithmic Optimizations. There are also some algorithmic optimizations for a specific graph mining application. An early study [45] presents an efficient ordering mechanism to achieve fast embedding isomorphism checks for FSM. Subsequently, the follow-up efforts introduce some heuristic optimizations to reduce intermediate embeddings [13], [22]. Specialized data structures are designed to process motifs for MC [34] efficiently. There are also some studies that attempt to use the GPU to accelerate the embedding enumeration for subgraph matching problems [25], [40]. Unlike these algorithm-by-algorithm optimizations, GRAMER emphasizes on general-purpose graph mining problems with hardware specialization.

Graph Mining Systems. Arabesque [38] and NScale [33] are the first generation of general-purpose graph mining systems in a distributed environment by adopting an embedding-centric model. To reduce the superfluous memory consumption in these earlier systems, RStream [42], a single machine graph mining system, stores the intermediate embeddings in SSD to mitigate the memory pressure. Fractal [12] further adopts a DFS execution model to avoid the substantial overheads of writing the intermediate embeddings to the main memory frequently. Unlike these software solutions built on the CPU, GRAMER identifies the extension locality opportunities in graph mining and further specializes some hardware designs to exploit them fully with massive parallelism.

Graph Processing Accelerators. The most related work to GRAMER is hardware accelerators on graph processing applications. Ham *et al.* [17] propose to utilize a large scratchpad memory efficiently. Yan *et al.* [44] represent an efficient architecture to improve data locality. There are also studies that leverage FPGA's BRAM effectively [11], [46]. Graph mining has the randomness complexity from both vertices and edges. Simply storing all irregular data on-chip as in these earlier accelerators is not applicable to graph mining since the on-chip memory is quite limited relative to the sizes of the modern graphs. To fill this gap, GRAMER distinguishes a small fraction of valuable data that will participate in the most random accesses and manages them differentially with sophisticated hardware specializations.

VIII. CONCLUSION

In this paper, we introduce GRAMER, an energy-efficient graph mining accelerator, which enables efficient memory accesses for irregular embedding enumeration in graph mining. GRAMER provides a locality-aware on-chip memory hierarchy that can process the most random memory accesses arising in graph mining on-chip to reduce their off-chip communication overheads. GRAMER also features a graph mining specific pipeline that can extract the maximum computational parallelism, and a work stealing mechanism for each PU to improve the load balance. Our evaluation on representative graph mining algorithms shows that GRAMER outperforms state-of-the-art CPU-based graph mining systems (Fractal and RStream) by $1.11\times \sim 129.95\times$ (in terms of speedups) and $5.79\times \sim 678.34\times$ (in terms of energy savings).

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful comments and valuable feedbacks. This work is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1003502, National Natural Science Foundation of China under Grant No. 61832006, 61825202, 61702201, and 61929103, and Australian Research Grants (DP170103956 and DP180104069). The correspondence of this paper should be addressed to Long Zheng.

REFERENCES

[1] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour, "Scalemine: Scalable parallel frequent subgraph mining in a single large graph," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016, pp. 716–727.

[2] C. C. Aggarwal and H. Wang, *Managing and mining graph data*, 2010, vol. 40.

[3] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng, "Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2017, pp. 125–137.

[4] T. Aittokallio and B. Schwikowski, "Graph-based methods for analysing networks in cell biology," *Briefings in bioinformatics*, vol. 7, no. 3, pp. 243–255, 2006.

[5] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 373–386.

[6] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, 2015, pp. 56–65.

[7] N. Beckmann and D. Sanchez, "Talus: A simple way to remove cliffs in cache performance," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 64–75.

[8] T. Chen, S. Srinath, C. Batten, and G. E. Suh, "An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2018, pp. 55–67.

[9] X. Cheng, C. Dale, and J. Liu, "Dataset for statistics and social network of youtube videos," http://netsg.cs.sfu.ca/youtubedata/, 2008.

[10] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Cliffhanger: Scaling performance cliffs in web memory caches," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016, pp. 379–392.

[11] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-FPGA architecture," in *Proceedings of International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 217–226.

[12] V. Dias, C. H. C. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy, "Fractal: A general-purpose graph pattern mining system," in *Proceedings of International Conference on Management of Data (SIGMOD)*, 2019, pp. 1357–1374.

[13] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "Grami: Frequent subgraph and pattern mining in a single large graph," *Proceedings of the VLDB Endowment*, vol. 7, no. 7, pp. 517–528, 2014.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 17–30.

[15] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 599–613.

[16] B. H. Hall, A. B. Jaffe, and M. Trajtenberg, "The nber patent citation data file: Lessons, insights and methodological tools," National Bureau of Economic Research, Tech. Rep., 2001.

[17] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[18] Intel, "Intel vtune amplifier," https://software.intel.com/en-us/vtune, 2020.

[19] S. Jiang and X. Zhang, "Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 1, pp. 31–42, 2002.

[20] T. Johnson and D. Shasha, "2q: a low overhead high performance buffer management replacement algorithm," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1994, pp. 439–450.

[21] H. Kim, J. Lee, S. S. Bhowmick, W.-S. Han, J. Lee, S. Ko, and M. H. Jarrah, "Dualsim: Parallel subgraph enumeration in a massive graph

on a single machine," in *Proceedings of ACM SIGMOD International Conference on Management of data (SIGMOD)*, 2016, pp. 1231–1245.

[22] M. Kuramochi and G. Karypis, "Finding frequent patterns in a large sparse graph," *Data Mining And Knowledge Discovery*, vol. 11, no. 3, pp. 243–271, 2005.

[23] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 31–46.

[24] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[25] W. Lin, X. Xiao, X. Xie, and X.-L. Li, "Network motif discovery: A GPU approach," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 3, pp. 513–528, 2017.

[26] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering," *IEEE Internet computing*, vol. 7, no. 1, pp. 76–80, 2003.

[27] A. Lumsdaine, D. P. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[28] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proceedings of European Conference on Computer Systems (EuroSys)*, 2017, pp. 527–543.

[29] 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 *Proceedings of ACM SIGMOD International Conference on Management of data (SIGMOD)*, 2010, pp. 135–146.

[30] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 457–468.

[31] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 166–177.

[32] M. Qiao, H. Zhang, and H. Cheng, "Subgraph matching: on compression and computation," *Proceedings of the VLDB Endowment*, vol. 11, no. 2, pp. 176–188, 2017.

[33] A. Quamar, A. Deshpande, and J. Lin, "Nscale: Neighborhood-centric large-scale graph analytics in the cloud," *The VLDB Journal*, vol. 25, no. 2, pp. 125–150, 2016.

[34] P. Ribeiro and F. Silva, "G-tries: a data structure for storing and finding subgraphs," *Data Mining and Knowledge Discovery*, vol. 28, no. 2, pp. 337–377, 2014.

[35] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013, pp. 135–146.

[36] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "Goffish: A sub-graph centric framework for large-scale graph analytics," in *Proceedings of European Conference on Parallel Processing (Euro-Par)*, 2014, pp. 451–462.

[42] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu, "Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 763–782.

[37] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 788–799, 2012.

[38] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015, pp. 425–440.

[39] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From think like a vertex to think like a graph," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.

[40] H.-N. Tran, J.-j. Kim, and B. He, "Fast subgraph matching on large graphs using graphics processors," in *Proceedings of International Conference on Database Systems for Advanced Applications (DASFAA)*, 2015, pp. 299–315.

[41] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache modeling and optimization using miniature simulations," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2017, pp. 487–498.

[43] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, W.-S. Ku, and J. C. Lui, "G-thinker: A distributed framework for mining subgraphs in a big graph," in *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2020, pp. 1369–1380.

[44] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2019, pp. 615–628.

[45] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *Proceedings of IEEE International Conference on Data Mining (ICDM)*, 2002, pp. 721–724.

[46] P. Yao, L. Zheng, X. Liao, H. Jin, and B. He, "An efficient graph accelerator with parallel data conflict management," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2018, pp. 8:1–8:12.

[47] P. Yuan, C. Xie, L. Liu, and H. Jin, "Pathgraph: A path centric graph processing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2998–3012, 2016.

[48] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[49] J. Zhou, S. Liu, Q. Guo, X. Zhou, T. Zhi, D. Liu, C. Wang, X. Zhou, Y. Chen, and T. Chen, "Tunao: A high-performance and energy-efficient reconfigurable accelerator for graph processing," in *2017 International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 731–734.

[50] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 301–316.

[51] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2015, pp. 375–386.