Things to Consider to Enable Dynamic Graphs in Processing-in-Memory

Euna Kim
euna.kim@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia

Hyesoon Kim
hyesoon@cc.gatech.edu
Georgia Institute of Technology
Atlanta, Georgia

ABSTRACT

With the ubiquity of graphs to represent large, sparse data sets, a recent focus has been placed on making graph processing more efficient. Processing-in-Memory (PIM) is an alternative solution to reduce the data movement between memory and processors, and it results in better performance and reduced energy consumption. However, prior PIM-based graph processing work operated only on static graphs. In fact, real-world graphs are constantly changing as their data is updated. In this paper, we will discuss design considerations for adapting dynamic graph processing to PIM.

CCS CONCEPTS

• **Computer systems organization → Real-time system architecture**.

KEYWORDS

Processing-in-memory, near memory, graph processing, dynamic graphs

ACM Reference Format:
Euna Kim and Hyesoon Kim. 2020. Things to Consider to Enable Dynamic Graphs in Processing-in-Memory. In *MEMSYS '20: The International Symposium on Memory Systems,Sept. 29 - Oct. 1, 2020, Virtual Conference.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Graph data structures represent relationships among entities and can be used to obtain insightful information with the incorporation of appropriate graph algorithms. Graph processing has been widely used in various domains in data science and has expanded to machine learning applications. Despite its essential functionality, graph processing is challenging due to the massive size of graph data sets and poor locality by the irregular memory access patterns of sparse graphs. The increased gap between memory and processors creates the "memory wall"[69] issue in which data movements are the performance bottleneck. Furthermore, most of the data transferred to cache is not reused in the conventional memory hierarchy [54] and massive energy is consumed for data transfer from DRAM to processors[17]. Data movement increases system latency and energy consumption. The emerging technology of 3D stacked memory plus a logic layer such as Hybrid Memory Cube (HMC)[3] and High Bandwidth Memory (HBM)[44] enabled the idea of Processing-in-Memory (PIM) to resolve these problems.

There are benefits, optimization opportunities as well as difficulties in PIM adoption as a graph processing accelerator. The general discussion is out of the scope of our work. Further details can be found in recent surveys[50][58][32].

Recently, several PIM-based graph processing accelerator architectures and runtimes have been designed [29][71][5][6][51][25][72][76]. Existing works provided diverse aspects of software and hardware co-design, programming models, hardware configurations to improve graph processing performance, scalability, programmability, concurrency, and energy efficiency. Tesseract[5] is the first architecture for graph processing in PIM that uses remote procedure calls (RPC). GraphPIM[51] showed the impact of offloading of atomic operations to PIM. GraphP[72] used source-cut with replica to reduce the communication volume. GraphH[25] improved the inter-cube communication overhead by implementing it in hardware and vertex re-indexing for compaction. GraphQ[76] is most advanced PIM-based graph processing hardware and software co-design solution with various optimizations in concurrent computation and batched communication. They each have a distinct design approach, but have common focus on the kinds of problems they are trying to solve. E.g., providing efficient graph data placement in memory, designing micro architectures for parallel computation units and communication units, and providing efficient communication methods for both intra-cube and inter-cube.

However, the primary assumption of existing work is that graph data sets are immutable. For example, in the preprocessing phase, graph data is partitioned and organized and no modification of the data is considered after this phase is complete. Real-world graph data sets are constantly updated. In order to process graphs with changes in PIM, there are several modifications and additions required such as data structures for dynamic graph and data layout with re-partitioning, which the remainder of this paper discusses. We outline the different classes of dynamic graphs in section 2, and we describe the system design for processing static graphs, as well as the modifications and considerations for dynamic graphs in PIM in section 3.

2 DYNAMIC GRAPHS

2.1 Dynamic Graph Classification

Real world data sets are continuously evolving and changing. Dynamic graphs are graphs with sequences of such updates. Dynamic graphs can be referred to as "fully dynamic" or "partially dynamic"

MEMSYS '20, Sept. 29 - Oct. 1, 2020, Virtual Conference

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

depending on the restrictions on insertion or deletion operation. Similarly, there are incremental graphs and decremental graphs where only the addition or deletion operation is allowed.

Yin, et al.[70] classified the characteristics of dynamic graphs into four types depending on the perspective of graph data sets and application behavior. These types are as follows: the Classic dynamic graph model[26], the Data Stream model[31], the Evolving graph model[7], and the Streamed graph model. The Classic dynamic graph model assumes that specialized data structures for efficient update for analysis are maintained to avoid recomputing of algorithms. In the Data stream model, graphs are streams of edges in the memory. Often the stream algorithms compute approximate results. In the Evolving graph model, the changes in the graph happen concurrently but not while the computation is occurring. The Evolving graph model is the only model that assumes concurrent changes in graphs, and approximate update algorithms are often applied to this model.

In practice, most of these dynamic graph growth is restricted by the physical memory (or storage) capacity. The dynamic graph model can be selected depending on the purpose of the system and the characteristics of the data sets. The graph changes for certain period of time are sometimes processed as a batch form with a sequence of updates for efficiency. After updating graphs, dynamic graph algorithms can be applied to compute fast results (if they exist) and sometimes they yield approximate results depending on the algorithm.

2.2 Dynamic Graph Constraints for PIM

We define a static graph to be a *completed* graph in any format; that is, its data is fixed. A dynamic graph is a *fully dynamic* graph; that is, its vertices and edges may be updated over time.

In order to efficiently support dynamic graphs, flexible data structures are required that have fast access time and good spatial locality. Many dynamic graph data structures are often provided as a part of graph frameworks, as the frameworks are designed and implemented to maximize the performance with programming models, data structures, and application implementations tailored to specific hardware constraints.

Dynamic graph frameworks can be mainly divided two types: in-memory data structures for keeping dynamic graphs such as Stinger[30], Hornet[21], Aspen[28], and GPMA[57]; and storage-based out-of-memory approaches such as LLAMA[45], X-Stream[56], TurboGraph[33], and GraphChi[41]. Typically storage-based approaches save the graph or the changes in the graph as snapshots in storage with timestamps. In this paper, since our goal is using PIM as a graph processing accelerator, we only consider the **in-memory** dynamic graph approach.

Dynamic graph methodologies have been studied in various area such as graph theory, high performance computing, software frameworks and libraries, specialized hardware and software co-design, as well as distributed computing on clusters. Many existing technologies can be applied to dynamic graph processing in PIM directly or with minor modifications, except for optimizations at the micro-architecture level.

3 PROCESSING DYNAMIC GRAPH IN PIM

Graph processing is well-known for poor locality in traditional memory systems due to the sparsity of data sets and random memory access behavior. In addition, the vast scale of graph data sets up to trillion edges[24] contributes to excessive amounts of data movement from memory to processors. One way of mitigating this issue is by pushing the processing closer to memory, such that large transfers of sparse data are not actually necessary; this is the main idea behind PIM.

Micron's Hybrid Memory Cube (HMC)[3] is one such PIM architecture design. Using stacked memory modules, connected vertically using through-silicon-vias (TSV) atop a processing logic layer, data processing can be performed near to memory without having to transfer data to a processor and back. The memory itself is divided into partitions called *vaults*, with each vault composed of multiple banks of DRAM modules. Each one of these vaults has its own logic layer, which may be identical across all vaults.

To process graphs using PIM, the graph data should be partitioned across all cubes, so that computations can be executed concurrently. Therefore, it is key that the micro-architecture design and communication methods minimize the communication volume and cost both inter-cube and intra-cube. Several design considerations must be taken into account to process graphs using PIM shown in **Figure 1**. We further elaborate on methodologies of static graph processing in general in PIM, in addition to design considerations for adapting dynamic graphs in the following subsections.

3.1 Graph Representation

Graph data sets can be represented and stored in various formats such as sparse matrix, Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), Coordinate format (COO), ELLPACK format, and variations of the ELLPACK format such as the Sliced ELLPACK (SELL) format. Also, hybrid formats[14] exist that use a combination of more than one format such as a hybrid ELL/COO format. No special format is required to process in PIM since the graph will go through a preprocessing phase to convert the graph to the data layout native to its micro-architecture and interconnection design.

Likewise, dynamic graphs do not need special formats to be represented, but the graph changes should be maintained in memory as a sequence of edge and vertex updates. Merging these updates into an existing graph can be done using a streaming approach or with periodic batch collected updates. There exist few real world graph containing timestamps such as reddit submission time or pokec registration time as attributes.

Note the graph representation is not a data structure in memory. The data structure for a dynamic graph should be flexible to handle graph changes efficiently, and that is essential for dynamic graph processing. Graph data structure is explained in Section 3.8: Data Structures.

3.2 Preprocessing

In the preprocessing phase, a graph is converted to the appropriate format corresponding to the PIM micro-architecture design for computation and communication. A graph is divided into partitioned graphs that map to cubes (or vaults) in PIM. For communication and

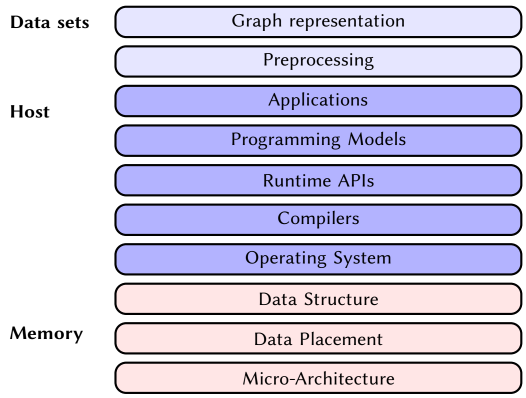

Figure 1: Design considerations for hardware and software co-design of graph processing in PIM

synchronization among the partitions, edge lists of each partition are maintained.

Graph partitioning is a NP-hard problem in graph theory[19], and finding the de facto solution is not feasible due to distinct properties of graph data sets. In particular, scale-free social graphs obeying the power-law naturally create few very large partitions since there are some nodes with very high degree. To reduce the communications among partitions, partitioning is conducted in a manner that minimizes the number of edge-cuts across partitions. This may incur imbalanced workloads, which hampers efficient parallel processing[42][11]. However, the size of a partitioned graph is not always proportional to the amount of computations. Depending on the algorithm, only small portions of vertices are active and participating in the computation. This is why few well-known offline graph partitioners such as METIS[37](ParMetis[43]) or PULP[59](XtraPULP[60]) could not always produce good results for partitioning for distributed systems, despite their relatively long execution times in the preprocessing phase[5].

There are different strategies for reducing the communication volume such as 1D partitioning (Edge-Cut)[38][60][62][75][37], 2D partitioning (Vertex-Cut)[40][16][22], Hybrid Vertex-Cut in Power Lyra[23], and Source-Cut partitioning used in PIM-based graph processing architecture GraphP[72]. Hybrid Cut is similar to other 1D partitioning methods except for that it performs special treatments for the high degree nodes. Source-Cut with replica maintains proxy (or ghost) nodes to reduce the communication volume by that computation can be done without communicating with nodes in other partitions. This approach can produce good performance with applications such as PageRank that has heavy communications with all 1 hop neighbors. The downside of this method is the memory footprint to store the replicas.

For dynamic graphs, the partitioning strategy in the preprocessing phase does not need to be different from that of static graphs for PIM processing. However, meta information for partitioning needs to be kept somehow for later use in re-partitioning because of graph updates. One thing that might need to be kept in mind is that, unlike static graphs, the size of the partitioned graph is not fixed. If the partition size is too close to the capacity of each cube, the partition would incur substantial data movements among partitions during re-partitioning as graph updates occur. Due to the overhead of frequent re-partitioning for graph changes, offline partitioning [37][43][59][60] which would not be an option unless it is used for initial partitioning and combined with other streaming partitioning approaches. We will discuss re-partitioning in Section 3.9: Data Placement.

There might be optimization techniques for static graphs that can be applied for data reorganization, such as reordering and space efficient graph compression, that might not be able to be used for dynamic graph cases. For dynamic graphs, the overhead for maintenance of these techniques on dynamic graphs might be too high, attenuating any benefits that these techniques would otherwise provide.

3.3 Applications

There are several fundamental graph algorithms used in real world applications. These algorithms include Breadth-First Search (BFS) and Single Source Shortest Path (SSSP), which are core kernels of the Graph500[2] benchmark for exploring the performance of High Performance Computers (HPC). Other algorithms include Depth-First Search (DFS), Connected Component (CC), Betweenness Centrality (BC), Triangle Counting (TC), PageRank (PR), Katz Centrality (KC), k-Truss, Graph Coloring (GC), K-core Decomposition, and Minimal Spanning Tree (MST). Graph algorithms can be classified by their behavior, e.g, BFS, DFS, SSSP, BC traverse nodes, while PR, KC, K-core are iterative algorithms and are computationally intensive. There are optimizations that consider the characteristics of such algorithms[12], as well as communication methods among partitions that consider specific hardware configurations[13][73].

In PIM-enabled graph processing, the applications need to be implemented using parallel algorithms to exploit the parallelism in each PIM core. Several efficient parallel algorithms have been studied[48][7] and some implementations are available[27]. If parallel and distributed versions of algorithms for specific applications exist, they can be imported and used for PIM graph processing. Otherwise, design and implementation of algorithms are the application programmers' responsibility, which could create a tremendous amount of burden.

For the dynamic graph case, there are two main algorithm choices: static algorithms and dynamic/incremental algorithms. After updates are applied to a graph, computing traditional graph algorithms from scratch as is done for static graphs is typical. This requires longer execution times, but the results are more accurate. To obtain fast results and/or concurrency of computing algorithms while updating on evolving graphs, dynamic/incremental algorithms can be an alternative such as dynamic Breadth-First Search[49], approximate Triangle Counting[20], incremental PageRank[10], Connected Component for dynamic graphs[47], fast approximate Single Source Shortest Path[61], approximate centrality algorithms for dynamic graphs[34][8][9][18]. Usually, dynamic algorithms begin with the final result of a previous computation and estimate new results, which are affected by graph updates. Some dynamic algorithms produce approximate results by computing the difference between the original graph and the graph post-update. If parallel and distributed versions of dynamic algorithms of targeted graph algorithms are not accessible, a significant amount of work may be required by application programmers.

3.4 Programming Models

Programming models for graph processing have been actively studied. The vertex-centric model is popular for its intuitive approach since it was introduced in Pregel[46]. To overcome the random access nature of edge lists, an edge-centric approach was introduced in X-Stream[56]. Later, data-centric[67], graph-centric[64] and hybrid programming models[74] were developed.

These models are associated with the traditional bulk synchronous parallel (BSP) model and the GAS (Gather, Apply and Scatter) model. There are also push and pull models that depend on the direction of values in order to reduce synchronization and communication costs. The appropriate programming model can be chosen depending on the targeted graph algorithms. Programming models are tightly associated with application implementations. As mentioned in Section 3.3 Applications, if programmers had to design parallel and distributed versions of targeted algorithms on their own, selecting the proper programming model for a specific application is the programmers' responsibility, which could require substantial effort by application programmers.

Another factor to select the appropriate programming model is partitioning methods. Some partitioning algorithms are designed to minimize the cuts among partitions which yield good performance in certain applications but not all applications such as graph traversing algorithms like Breath-First-Search (BFS).

For dynamic graphs, static programming models can often be used with no modifications, unless dynamic algorithms are available and special programming models are required to execute them efficiently.

3.5 Runtime APIs

When designing a runtime for PIM, considerations must be made regarding which functions will be executed in PIM for improved performance. In addition, it must also have facilities for multi-threading, controlling PIM cores, scheduling of PIM operations, orchestrating multiple PIMs (if multi-PIM supported), and data placement of the partitioned graph.

As for runtime APIs, the features that are exposed to programmers will depend on the available application capabilities and whether a feature should have an explicit API at all. For example, the aforementioned runtime capabilities can be implicitly implemented in the runtime and at the compiler level for maximizing performance and reducing energy consumption, or they can be exposed and available to programmers as runtime APIs.

For dynamic graphs, runtime APIs for graph updates need to be added, as well as those for dynamic algorithms if dynamic algorithms are supported in PIM. Data movement resulting from graph re-partitioning that occurs as a result of graph updates can also be exposed via runtime APIs or implicitly triggered as a part of the graph update procedure. Details of re-partitioning will be explained in section 3.9 Data Placement.

3.6 Compilers

Programming models, runtimes, and compilers are tightly related when using PIM. When developing the roles and capabilities of compilers for PIM, there are trade-offs that must be considered: the compiler can assist in either mitigating the programmers' burden, or providing flexibility to the programmers.

Adapting dynamic graph processing to a compiler intended for PIM adds the functionality mentioned in Section 3.5: Runtime APIs.

3.7 Operating Systems

An operating system should provide a cache coherence protocol, virtual memory management, multi-threading capability, concurrency, and facilities for allocating memory; all of these capabilities might be affected by adoption of PIM. For example, the returned results from PIM will likely remain in the last level (e.g., L3) of the cache since there is a reduced chance of data reuse. There is the potential for optimizations of the cache coherence protocol in the operating system to mitigate high rates of cache misses.

As for dynamic graph processing, no special modifications are anticipated for the operating system.

3.8 Data Structures

Graph data structures in PIM for static graphs can be in any format mentioned in Section 3.1: Graph Representation.

For adapting dynamic graphs, to our knowledge, there are mainly two choices of existing data structure schemes. First, leaving extra space for future additions to the graph i.e, Dynamic CSR (DCSR)[39]. DCSR is an extension of hybrid graph formats. It is easy to convert from commonly used formats such as CSR or COO. The key to producing good results is properly setting the size reserved for graph updates (insertions), which is difficult to predict. If the size is too large, space will be wasted, and locality is worsened. If it is too small, scaling is limited. The second scheme is flexible data structures for data insertion and deletion. These approaches are found in Stinger[30] as linked lists, as well as tree-based approaches such as B+tree in Hornet[21], PMA[15] GPMA[57], PAM[63], and block memory of tree in Packed CSR[68], and C-tree in Aspen[28]. For faster access, tree nodes are often implemented using blocks of memory (chunks).

Technically most of aforementioned dynamic graph data structures can be used for dynamic graph processing in PIM regardless of their complexity in the implementation and data load methodology to PIM cores. The effectiveness of graph compression and reordering for dynamic graphs, which requires maintenance overhead, likely needs to be examined in the future.

3.9 Data Placement

Each partitioned graph during the preprocessing phase is mapped to available cubes in PIM according to the hardware configuration including the interconnection network design. The data structure and edge list in the partition are arranged in memory such that maximum performance can be attained by applications for a given hardware configuration.

To adapt dynamic graphs to PIM with respect to data placement, a new partitioning strategy is required that has two additional considerations. The first problem is how to map newly arrived

Table 1: Summary of graph design considerations, modifications, and the level of effort required to adapt dynamic graph processing to existing static graph processing in PIM

Static Graph		Modification	Additions for Dynamic Graphs	Level of Effort[1] Impl.Required	Import
Graph representation		Depends	Edge stream or batches	Minimal	-
Preprocessing		Depends	Some optimizations might not applicable	Minimal	-
Applications[2]		Optional	Static algorithms vs. dynamic algorithms	Substantial[3]	Minimal[3]
Programming models[2]		Optional	Considering dynamic algorithms	Substantial[3]	Minimal[3]
Runtime APIs	✓	Add-on	Adding features (re-partitioning, dynamic algorithms)	Moderate	
Operating systems		Optional	(Virtual memory and cache coherence)	None or minimal	-
Compilers	✓	Add-on	Dealing with runtime APIs for dynamic graphs	Moderate	
Data structures[2]	✓	Required	Dynamic graph data structure	Substantial[4]	Moderate[4]
Data placement[2]	✓	Required	Re-partitioning and workload balance	Substantial[4]	Moderate[4]
Micro-architectures		Optional	(Communication logic design and data load)	None or minimal	-

[1]"Impl. Required" refers to the levels of effort required to design and implement a given modification for dynamic graphs, or to tailor an optimization for PIM from an existing static graph solution. "Import" refers to whether existing algorithms or reference implementations are available to use for dynamic graph cases.
[2]These are items that require major changes for dynamic graph processing in PIM from existing static graph processing PIM software and hardware co-design architectures.
[3]Supporting dynamic algorithms is optional depending on the requirements. The levels of effort were estimated with the assumptions required for implementing dynamic algorithms.
[4]The levels of effort for data structures and data placement for dynamic graphs include the complexity of managing the data structure and communications among partitions.

graph updates such as edge additions to partitions in PIM efficiently. The second is how to deal with growing partition size and workload imbalance. The workload balance can be evaluated using the ratio of the size of partitions, i.e, the largest partition size vs. the smallest (or average) partition size in general.

The offline partition algorithms[37][43][59][60] are not suitable for dynamic graph partitioning. Typically they are modularity-based iterative algorithms on a completed graph, which means the graph partition procedure will be conducted from scratch with any graph modification. To reduce the execution time for partitioning, streaming partitioning[62] can be a good solution for new vertex/edge assignment to partitions.

In the streaming partitioning approach, the graphs are considered to be a stream of edges; the decision is made by a single scan of the graph. The graph read can be repeated for refinement purposes. The most common and simplest streaming partition policies are random, hash, and round-robin. There are heuristic algorithms to produce better quality partitions for streaming partitioning based on greedy approaches using edge-cut, such as Linear deterministic Greedy (LDG)[62], FENNEL[65], Edge-balanced Gemini[75], Leopard[36], GraSP[11], xDGP[66], and more streaming edge partitioning algorithms found [52][53][35]. Partition quality, in this case, refers to the number of edge-cuts. Some streaming partitioning algorithms need information such as the total number of vertices in the graph or in the partition. This metadata needs to be tracked if different graph partitioners are used in preprocessing and data placement of dynamic graph updates.

There are a couple of issues for dynamic graphs in PIM which are not covered by existing solutions such as how to determine the number of PIM cores to use, when the partition needs to be split if it grows close to the size of PIM cube capacity, and if node migration will be allowed to reduce communication if the connectivity of the node is affected by graph updates. The tradeoffs involved include either potential benefits or increases in system complexity.

3.10 Micro-Architectures

The micro-architecture design for the logic layer in the 3D stacked memory is the core of graph processing in PIM. Tesseract[5] eliminated a shared cache to avoid cache coherence overhead, having only private caches. The Tesseract PIM cores are identical while GraphQ[76] have heterogeneous cores of a computation (Processing) unit and a communication (Apply) unit to improve concurrency. The Processing unit has a prefetcher with no cache, and the Apply unit has a small scratchpad memory.

Adapting dynamic graph processing to those kinds of micro-architectures may be possible with minimal effort, if data placement changes and re-partitioning is handled properly in other layers such as runtimes or compilers. However, that would be difficult for certain PIM-based graph processing architectures having less flexibility for optimizations such as GraphH[25]. GraphH implements the inter-cube communication in hardware and conducts data compaction by eliminating vertices with no edges during the preprocessing phase.

3.11 Development Environments & Evaluation

Most of graph applications, data sets, as well as simulators for evaluating micro-architectures for static graph processing in PIM can be used for dynamic graphs as well.

For evaluation of processing dynamic graphs, data sets for running benchmarks can be prepared in two ways. First, using dynamic graphs which have node generation information such as timestamps as attributes[1][55][4]. Second, creating batch updates by randomly choosing nodes or edges in the graph. These data items can then be removed from the graph, and added once again as a batch. Creating batch updates in this way maintains the properties of the graph.

To evaluate dynamic algorithms for dynamic graphs, the performance can be compared to the runtime of static algorithm execution. If evaluated dynamic algorithms produce approximate results, the accuracy may need to be presented as well.

4 CONCLUSION

In this paper, we discussed the design considerations for PIM-based systems to process dynamic graphs as efficiently as static graphs and explored the opportunities for future research.

The programming model, runtime APIs, and compiler affects the burden placed on software developers. The chosen data structures and their layouts in memory affect the programming model and communication methods. To reduce this burden, the operating system and compiler can be modified for efficient graph processing. In addition, existing tools can be used, such as graph frameworks and graph partitioners. A graph framework usually includes a graph data structure, programming model, and a set of algorithms. Note that there is no specialized graph partitioner for PIM to our knowledge, although data placement is key to PIM acceleration. Existing partitioners can be adapted to the PIM preprocessing phase with manual modification for communication.

Table 1 summarizes the various design considerations for both dynamic and static graphs, as well as the level of effort required to support dynamic graph processing with existing PIM-enabled static graph processing configurations.

REFERENCES

[1] [n.d.]. BigDND: Big Dynamic Network Data. http://projects.csail.mit.edu/dnd/
[2] [n.d.]. Graph500. http://www.graph500.org/.
[3] [n.d.]. Hybrid Memory Cube Consortium et al. 2015. *Hybrid memory cube specification version 2.1. Technical Report* ([n. d.]).
[4] [n.d.]. Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/#citnets
[5] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 105–117. https://doi.org/10.1145/2872887.2750386
[6] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 336–348.
[7] Aris Anagnostopoulos, Ravi Kumar, Mohammad Mahdian, Eli Upfal, and Fabio Vandin. 2012. Algorithms on Evolving Graphs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference* (Cambridge, Massachusetts) *(ITCS '12)*. Association for Computing Machinery, New York, NY, USA, 149–160. https://doi.org/10.1145/2090236.2090249
[8] David Bader, David, Kintali, Shiva, Kamesh Madduri, Kamesh, Milena Mihail, and Milena. 2007. Approximating Betweenness Centrality. https://doi.org/10.1007/978-3-540-77004-6_10
[9] Miriam Baglioni, Filippo Geraci, Marco Pellegrini, and Ernesto Lastres. 2012. Fast Exact Computation of Betweenness Centrality in Social Networks. In *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012) (ASONAM '12)*. IEEE Computer Society, USA, 450–456. https://doi.org/10.1109/ASONAM.2012.79
[10] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. 2010. Fast Incremental and Personalized PageRank over Distributed Main Memory Databases. *CoRR* abs/1006.2880 (2010). arXiv:1006.2880 http://arxiv.org/abs/1006.2880
[11] Casey Battaglino, Pienta Pienta, and Richard Vuduc. 2015. GraSP: distributed streaming graph partitioning. https://doi.org/10.5821/hpgm15.3
[12] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-Optimizing Breadth-First Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Washington, DC, USA, Article 12, 10 pages.
[13] S. Beamer, K. Asanović, and D. Patterson. 2017. Reducing Pagerank Communication via Propagation Blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 820–831.
[14] N. Bell and M. Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11.
[15] Michael A. Bender and Haodong Hu. 2007. An Adaptive Packed-Memory Array. *ACM Trans. Database Syst.* 32, 4 (Nov. 2007), 26–es. https://doi.org/10.1145/1292609.1292616
[16] Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. 2013. Scalable Matrix Computations on Large Scale-Free Graphs Using 2D Graph Partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 50, 12 pages. https://doi.org/10.1145/2503210.2503293
[17] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. *SIGPLAN Not.* 53, 2 (March 2018), 316–331. https://doi.org/10.1145/3296957.3173177
[18] Ulrik Brandes and Christian Pich. 2007. Centrality Estimation in Large Networks. *Int. J. Bifurc. Chaos* 17 (2007), 2303–2318.
[19] Thang Nguyen Bui and Curt Jones. 1992. Finding good approximate vertex and edge partitions is NP-hard. *Inform. Process. Lett.* 42, 3 (25 May 1992), 153–159. https://doi.org/10.1016/0020-0190(92)90140-Q
[20] Laurent Bulteau, Vincent Froese, Konstantin Kutzkov, and Rasmus Pagh. 2014. Triangle counting in dynamic graph streams. *CoRR* abs/1404.4696 (2014). arXiv:1404.4696 http://arxiv.org/abs/1404.4696
[21] F. Busato, O. Green, N. Bombieri, and D. A. Bader. 2018. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7.
[22] Umit Catalyurek, Cevdet Aykanat, and Bora Uçar. 2010. On Two-Dimensional Sparse Matrix Partitioning: Models, Methods, and a Recipe. *SIAM J. Scientific Computing* 32 (07 2010), 656–683. https://doi.org/10.1137/080737770
[23] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 1, 15 pages. https://doi.org/10.1145/2741948.2741970
[24] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1804–1815. https://doi.org/10.14778/2824032.2824077
[25] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang. 2019. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (2019), 640–653.
[26] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. 2010. *Dynamic Graph Algorithms* (2 ed.). Chapman and Hall/CRC, 9.
[27] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA '18)*. Association for Computing Machinery, New York, NY, USA, 393–404. https://doi.org/10.1145/3210377.3210414
[28] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-Latency Graph Streaming Using Compressed Purely-Functional Trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 918–934. https://doi.org/10.1145/3314221.3314598
[29] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. 2002. The Architecture of the DIVA Processing-in-Memory Chip. In *Proceedings of the 16th International Conference on Supercomputing* (New York, New York, USA) *(ICS '02)*. Association for Computing Machinery, New York, NY, USA, 14âĂŞ25. https://doi.org/10.1145/514191.514197
[30] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–5.
[31] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. 2005. On Graph Problems in a Semi-Streaming Model. *Theor. Comput. Sci.* 348, 2 (Dec. 2005), 207–216. https://doi.org/10.1016/j.tcs.2005.09.013
[32] Chuangyi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xinyu Chen, Xiaofei Liao, and Hai Jin. 2019. A Survey on Graph Processing Accelerators: Challenges and Opportunities. *CoRR* abs/1902.10130 (2019). arXiv:1902.10130 http://arxiv.org/abs/1902.10130
[33] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: A Fast Parallel Graph Engine Handling Billion-Scale Graphs in a Single PC. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Chicago, Illinois, USA) *(KDD '13)*. Association for Computing Machinery, New York, NY, USA, 77–85. https://doi.org/10.1145/2487575.2487581
[34] Takanori Hayashi, Takuya Akiba, and Yuichi Yoshida. 2015. Fully Dynamic Betweenness Centrality Maintenance on Massive Networks. *Proc. VLDB Endow.* 9, 2 (Oct. 2015), 48–59. https://doi.org/10.14778/2850578.2850580
[35] L. Hoang, R. Dathathri, G. Gill, and K. Pingali. 2019. CuSP: A Customizable Streaming Edge Partitioner for Distributed Graph Analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 439–450.
[36] Jiewen Huang and Daniel Abadi. 2016. Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs. *Proceedings of the VLDB Endowment*

9 (03 2016), 540–551. https://doi.org/10.14778/2904483.2904486
[37] George Karypis and Vipin Kumar. 1995. METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0. (01 1995).
[38] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (Dec. 1998), 359–392.
[39] James King, Thomas Gilray, Robert Michael Kirby, and Matthew Might. 2016. Dynamic-CSR : A Format for Dynamic Sparse-Matrix Updates.
[40] Vipin Kumar, A. Grama, Anshul Gupta, and George Karypis. 1994. *Introduction to parallel computing. Design and analysis of algorithms.* Vol. 2.
[41] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 31–46.
[42] Kevin Lang. 2004. Finding good nearly balanced cuts in power law graphs. (12 2004).
[43] D. Lasalle and G. Karypis. 2013. Multi-threaded Graph Partitioning. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 225–236.
[44] Dong Uk Lee, Kyung Whan Kim, Kwan W. Kim, Hongjung Kim, Ju Young Kim, Young Jun Park, Jae Hwan Kim, Dae Suk Kim, Heat Bit Park, Jin Wook Shin, Jang Hwan Cho, Ki Hun Kwon, Min Jeong Kim, Jaejin Lee, Kun Woo Park, Byongtae Chung, and Sungjoo Hong. 2014. 25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV. *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)* (2014), 432–433.
[45] Peter Macko, Virendra Marathe, Daniel Margo, and Margo Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. *Proceedings - International Conference on Data Engineering* 2015 (05 2015), 363–374. https://doi.org/10.1109/ICDE.2015.7113298
[46] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184
[47] R. McColl, O. Green, and D. A. Bader. 2013. A new parallel algorithm for connected components in dynamic graphs. In *20th Annual International Conference on High Performance Computing*. 246–255.
[48] Andrew McGregor. 2014. Graph Stream Algorithms: A Survey. *SIGMOD Rec.* 43, 1 (May 2014), 9–20. https://doi.org/10.1145/2627692.2627694
[49] Ulrich Meyer. 2008. On Dynamic Breadth-First Search in External-Memory. *CoRR* abs/0802.2847 (2008). arXiv:0802.2847 http://arxiv.org/abs/0802.2847
[50] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2019. Processing Data Where It Makes Sense: Enabling In-Memory Computation. *CoRR* abs/1903.03988 (2019). arXiv:1903.03988 http://arxiv.org/abs/1903.03988
[51] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 457–468.
[52] Joel Nishimura and Johan Ugander. 2013. Restreaming Graph Partitioning: Simple Versatile Algorithms for Advanced Balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Chicago, Illinois, USA) *(KDD '13)*. Association for Computing Machinery, New York, NY, USA, 1106–1114. https://doi.org/10.1145/2487575.2487696
[53] A. S. Pope, D. R. Tauritz, and A. D. Kent. 2016. Evolving Multi-level Graph Partitioning Algorithms. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*. 1–8.
[54] Moinuddin K. Qureshi, M. Aater Suleman, and Yale N. Patt. 2007. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*. IEEE Computer Society, USA, 250–259. https://doi.org/10.1109/HPCA.2007.346202
[55] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. http://networkrepository.com
[56] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 472–488. https://doi.org/10.1145/2517349.2522740
[57] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proc. VLDB Endow.* 11, 1 (Sept. 2017), 107–120. https://doi.org/10.14778/3151113.3151122
[58] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. 2019. Near-Memory Computing: Past, Present, and Future. *Microprocess. Microsystems* 71 (2019).
[59] G. M. Slota, K. Madduri, and S. Rajamanickam. 2014. PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks. In *2014 IEEE International Conference on Big Data (Big Data)*. 481–490.
[60] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri. 2017. Partitioning Trillion-Edge Graphs in Minutes. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 646–655.
[61] S. Srinivasan, S. Riazi, B. Norris, S. K. Das, and S. Bhowmick. 2018. A Shared-Memory Parallel Algorithm for Updating Single-Source Shortest Paths in Large Dynamic Networks. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. 245–254.
[62] Isabelle Stanton and Gabriel Kliot. 2012. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Beijing, China) *(KDD '12)*. Association for Computing Machinery, New York, NY, USA, 1222–1230. https://doi.org/10.1145/2339530.2339722
[63] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. 2016. PAM: Parallel Augmented Maps. *CoRR* abs/1612.05665 (2016). arXiv:1612.05665 http://arxiv.org/abs/1612.05665
[64] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think like a Vertex" to "Think like a Graph". *Proc. VLDB Endow.* 7, 3 (Nov. 2013), 193–204. https://doi.org/10.14778/2732232.2732238
[65] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming Graph Partitioning for Massive Scale Graphs. In *WSDM*. ACM.
[66] Luis M. Vaquero, Félix Cuadrado, Dionysios Logothetis, and Claudio Martella. 2013. xDGP: A Dynamic Graph Processing System with Adaptive Partitioning. *CoRR* abs/1309.1049 (2013). arXiv:1309.1049 http://arxiv.org/abs/1309.1049
[67] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. *SIGPLAN Not.* 51, 8, Article 11 (Feb. 2016), 12 pages. https://doi.org/10.1145/3016078.2851145
[68] Brian Wheatman and Helen Xu. 2018. Packed Compressed Sparse Row: A Dynamic Graph Representation. 1–7. https://doi.org/10.1109/HPEC.2018.8547566
[69] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20âĂŞ24. https://doi.org/10.1145/216585.216588
[70] Chunxing Yin, Jason Riedy, and David A. Bader. 2018. A New Algorithmic Model for Graph Analysis of Streaming Data. In *Proceedings of the 14th International Workshop on Mining and Learning with Graphs (MLG)*. http://www.mlgworkshop.org/2018/papers/MLG2018_paper_23.pdf
[71] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-Oriented Programmable Processing in Memory. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing* (Vancouver, BC, Canada) *(HPDC '14)*. Association for Computing Machinery, New York, NY, USA, 85âĂŞ98. https://doi.org/10.1145/2600212.2600213
[72] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 544–557.
[73] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K. Prasanna. 2018. An FPGA Framework for Edge-centric Graph Processing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '18)*. ACM, New York, NY, USA, 69–77. https://doi.org/10.1145/3203217.3203233
[74] Shijie Zhou and V. Prasanna. 2017. Accelerating Graph Analytics on CPU-FPGA Heterogeneous Platform. 137–144. https://doi.org/10.1109/SBAC-PAD.2017.25
[75] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 301–316.
[76] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 712–725. https://doi.org/10.1145/3352460.3358256