Degree-aware Hybrid Graph Traversal on FPGA-HMC Platform

Jialiang Zhang and Jing Li
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
jialiang.zhang@ece.wisc.edu,jli@ece.wisc.edu

ABSTRACT

Graph traversal is a core primitive for graph analytics and a basis for many higher-level graph analysis methods. However, irregularities in the structure of scale-free graphs (e.g., social network) limit our ability to analyze these important and growing datasets. A key challenge is the redundant graph computations caused by the presence of high-degree vertices which not only increase the total amount of computations but also incur unnecessary random data access.

In this paper, we present a graph processing system on an FPGA-HMC platform, based on software/hardware co-design and co- optimization. For the first time, we leverage the inherent graph property i.e. vertex degree to co-optimize algorithm and hardware architecture. In particular, we first develop two algorithm optimization techniques: *degree-aware adjacency list reordering* and *degree-aware vertex index sorting*. The former can reduce the number of redundant graph computations, while the latter can create a strong correlation between vertex index and data access frequency, which can be effectively applied to guide the hardware design. We further implement the optimized hybrid graph traversal algorithm on an FPGA-HMC platform. By leveraging the strong correlation between vertex index and data access frequency made by degree-aware vertex index sorting, we develop two platform-dependent hardware optimization techniques, namely *degree-aware data placement* and *degree-aware adjacency list compression*. These two techniques together substantially reduce the amount of access to external memory. Finally, we conduct extensive experiments on an FPGA-HMC platform to verify the effectiveness of the proposed techniques. To the best of our knowledge, our implementation achieves the highest performance (45.8 billion traversed edges per second) among existing FPGA-based graph processing systems.

ACM Reference Format:
Jialiang Zhang and Jing Li. 2018. Degree-aware Hybrid Graph Traversal on FPGA-HMC Platform. In *Proceedings of 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3174243.3174245

1 INTRODUCTION

In response to the increasingly larger and more diverse graphs in social science[21], machine learning[8], search engine[11], and the critical need of analyzing them, graph analytics, an essential class of big data analysis, has emerged as a new fundamental computing methodology to explore the comprehensive relationship among a vast collection of interconnected entities. Among all graph primitives in graph analytics, graph traversal has served as a basis for many higher-level graph analysis algorithms.

Unfortunately, graph traversal is notoriously inefficient due to the low computational intensity and irregular data access [2]. The problem is further aggravated in processing scale-free graphs — an essential class of real-world graphs where the distribution of vertex degrees (the number of edge connections per vertex) asymptotically follows a power law distribution. Scale-free graphs have been widely used in a number of important application domains including social science, computer network, finance, and biology. However, despite of their popularity, the unique topology of scale-free graphs creates additional challenges in processing. The reason is that the presence of high-degree vertices in scale-free graphs can cause a large number of redundant edge checks during the traversal, as reported by a number of prior work [2, 3, 14, 22]. The redundant edge checks not only increase the total number of graph computations but also incur unnecessary random data access, becoming the key performance bottleneck of existing graph processing systems.

To tackle such challenge, several existing work focus on algorithm optimization on conventional CPU- or GPU-based systems. For instance, Beamer *et al.* [3] proposes a *bottom-up* method, which takes an opposite direction of visiting the adjacency list of each vertex, compared to the traditional *top-down* approach. Experiment results confirm its effectiveness in reducing the number of redundant edge checks and achieving high throughput on large scale-free graphs on CPU-based systems. Gunrock *et al* [14, 22] further applies directional optimization to derive a *hybrid* graph traversal method by combining the best advantages of both *top-down* and *bottom-up* approaches. In their hybrid method, traversal direction (either top-down or bottom-up) can be optimally selected at each step during the traversal. The implementation on GPU-based systems with GPU specific optimizations has shown that the hybrid method is more effective than either of the prior methods (*bottom-up* or *top-down*) in reducing redundant graph computations and thus achieves better performance.

Although algorithm optimization has been demonstrated as an important approach to improve the efficiency of graph traversal, high-performance graph processing system could further benefit from careful optimization of the underlying hardware architectures, as often times the performance bottleneck of existing graph processing systems has shown to be bounded by the external memory [5]. Several recent works propose that customizing the hardware using FPGA can effectively alleviate the memory bottleneck [6, 10, 12, 26]. However, despite different system architectures, most of these works [6, 10, 12] are based on one common scheme: By

FPGA '18, February 25–27, 2018, Monterey, CA, USA

ACM ISBN 978-1-4503-5614-5/18/02…$15.00
https://doi.org/10.1145/3174243.3174245

placing hot data, which is used for synchronization between parallel kernels, on the on-chip block ram (BRAM) to alleviate the pressure on accessing external DRAM, the efficiency of processing scale-free graphs can be significantly improved. In addition to conventional DRAM-based FPGA graph processing systems, Zhang *et al* [25] propose to leverage the exceptional random access performance of the emerging hybrid memory cube (HMC) to further improve the external memory access. However, this body of research all implements the conventional *top-down* graph traversal algorithm and does not leverage the state-of-the-art direction optimization techniques, resulting in limited performance gain.

In this work, we leverage the inherent graph property i.e. *vertex degree* to co-optimize algorithm and hardware architecture to achieve a workload-optimized graph processing system. We will show that *vertex degree* contains rich information of graph topology in scale-free graphs and thus provides another key dimension in optimization space in both algorithm and hardware. To the best of our knowledge, we are the *first* to leverage the degree information to optimize graph traversal algorithm to reduce the redundant graph computation and thus improve memory access. Our work also differs from prior work on hardware customization, as we not only implement the state-of-the-art hybrid graph traversal but also optimize the system design by leveraging the essential graph property.

Specifically, we made the following contributions.

- We performed a comprehensive study on the properties of real-world scale-free graphs and found that the degree distribution of the graph is highly non-uniform. This non-uniform degree distribution forms the basis of the various optimization techniques in this paper.
- Based on these insights from graph analysis, we developed two algorithm optimization techniques: *degree-aware adjacency list reordering* and *degree-aware vertex index sorting*. The former can reduce the number of redundant edge checks in the *bottom-up* method, while the latter can create a strong correlation between vertex index and data access frequency, which can be effectively applied to guide the hardware design.
- We developed a new graph processing system to implement the optimized hybrid graph traversal algorithm on an FPGA-HMC platform. By leveraging the strong correlation between vertex index and data access frequency made by degree-aware vertex index sorting, we further developed two platform-dependent hardware optimization techniques, namely *degree-aware data placement* and *degree-aware adjacency list compression*. These two techniques together substantially reduce the amount of access to external memory.
- We conducted extensive experiments on an FPGA-HMC platform to verify the effectiveness of the proposed techniques. To the best of our knowledge, our implementation achieves the highest performance (45.8 billion traversed edges per second) among existing FPGA-based graph processing systems.

The rest of the paper is organized as follows. In section 2, we present the background of hybrid graph traversal algorithm and the FPGA-based graph processing system. In section 3, we present the insights gained on analyzing the scale-free graphs followed by two algorithm optimization techniques, namely degree-aware adjacency list reordering and degree-aware vertex index sorting. In section 5, we present the software/hardware implementations including the key data structure, system architecture and two platform-dependent hardware optimization techniques, namely degree-aware data placement and degree-aware pointer compression. In section 6, we present the evaluation methodology and experimental results. In section 7, we conclude the paper.

2 BACKGROUND

In this section, we will first introduce the state-of-the-art hybrid graph traversal algorithm. Specifically, we will discuss the *top-down* method, the *bottom-up* method and the *inter-step* direction switching. Then, we will review existing work on the FPGA-based graph traversal system.

2.1 Hybrid graph traversal

Graph traversal is one of the most important kernels of many graph applications, and it is typically used to test the connectivity or to find the single-source shortest paths. As shown in figure 1, graph traversal starts from one source vertex, and the frontier expands to the neighbors of the source vertices during each step. All of the vertices at the same depth will be visited before visiting any vertices at a greater depth. The major portion of the computation in graph traversal is to find the unvisited neighbors of the frontier. The process ends when all vertices are visited, and yields a spanning tree, which contains all the connected source vertices. The number of edges in the spanning tree indicates the theoretical minimum number of edge check in a graph traversal. In the best case, if the spanning tree is known, we can achieve this minimum number. Here, we define the **redundant edge check**: the edge check which does not add an edge to the final spanning tree.

The conventional *top-down* method (figure 1) starts from the frontier, each vertex in the frontier checks all of its neighbors to see if any of them are unvisited. Each unvisited neighbor is marked as visited, added to the frontier of next step. The total number of edge checks with the *top-down* method is equal to the number of edges in the connected component containing the source vertex, as in each step every edge in the frontier is checked. When the number of edges to be checked from frontier is large, the *top-down* method becomes inefficient, as it checks all the edge connect to the same vertices from different vertices in the frontier, and only one check will update the final spanning tree. As a result, it performs a large number of redundant edge check.

To address this issue Beamer [3] proposes the *bottom-up* method for implementing the graph traversal algorithm. In this method, instead of iterating through the frontier, it iterate through the unvisited vertices array. For each, we test to see if any of its neighbors is in the frontier. When a neighbor is found in the frontier, *bottom-up* method terminates the edge check earilier, mark the vertex as visited and add it to the frontier of next step (figure 1(c)). Therefore, the *bottom-up* method can reduce the number of redundant edge check. This technique is advantageous when the frontier size is large, and disadvantageous when the frontier is small.

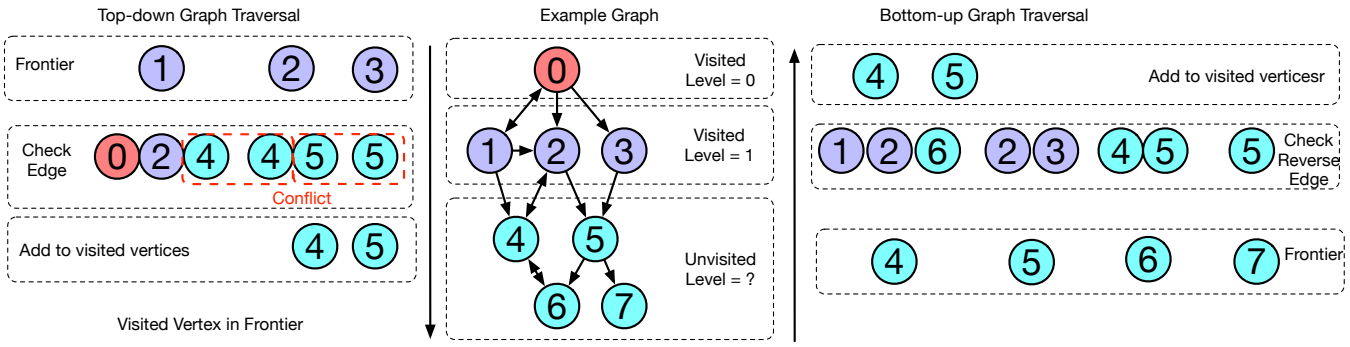

Figure 1: Example of *top-down* and *bottom-up* method in graph traversal

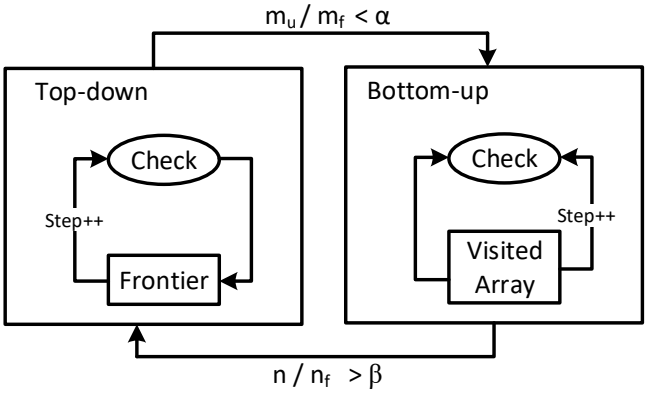

Figure 2: Hybrid (direction-optimizing) BFS

The *top-down* and *bottom-up* algorithms are complementary, since the *bottom-up* method performs well, when the frontier size is large, whereas the *top-down* method performs worse, and vice versa. To determine the graph traversal direction, Beamer et al. further introduce the Hybrid traversal algorithm. Figure 2 presents the flow of hybrid graph traversal. The hybrid graph traversal always starts from using *top-down* method as the frontier size is 1. As in figure 2, two thresholds(α and β) are used to control the direction switching, m_u represents the unexplored edge count, m_f is the number of edges to be checked from the frontier, and α determines when the number of edges to be checked fro the frontier is large enough to switch from *top-down* to *bottom-up*; n represents the number of vertices in the graph and n_f is the number of vertices in the frontier, and β determines when the frontier is small enough to use *top-down* method. Both of the two thresholds are heuristically determined. In this paper, we will show that there is still substantial to further improve the performance of hybrid graph traversal, and building an efficient hybrid graph traversal system on FPGA-HMC system will require a number of optimizations, including both platform-independent and platform-dependent techniques.

2.2 FPGA-based Graph Processing framework

There are several existing works on implementing graph accelerators using FPGAs. GRAPHGEN[18] proposed an FPGA-based graph processing system using a vertex-centric model. However, it stores the whole graph in the on-board DDR DRAM, which severely limits the performance due to the bandwidth bottleneck of the memory. Also, the design does not provide any platform-aware software and hardware optimizations for implementing BFS. TorusBFS [13] proposed a 2-D message passing structure to reduce the latency between parallel BFS kernels, but its performance is also limited by the poor random access performance of DRAM and the available on-chip resources. FPGP [6] employed an interval-shared structure to maximize off-chip memory bandwidth and to exploit the parallelism of graph processing fully. However, its performance is still bounded by the capacity and bandwidth of FPGA's on-chip memory. ForeGraph[7] proposes to improve the scalability on multi-FPGA architectures. [25] implements the push method of graph traversal algorithms on an FPGA-HMC platform. All of these works are proposing new algorithms, data structures and hardware architectures to increase locality and utilization of memory bandwidth. However, none of these framework attempt to reduce the number of computations (number of edge checks). To the best of our knowledge, we are the first work to provide architectural support on FPGA for the hybrid graph traversal to reduce the number of computations.

3 ALGORITHM OPTIMIZATION

In this section, we will first present our observation on the non-uniform degree distribution of scale-free graphs. Then, we will propose two optimization techniques to reduce the redundancy in hybrid graph traversal based on the degree information. In particular, we effectively leverage the degree information of scale-free graphs to reduce the redundancy in the algorithm and provide insights to its data access frequency.

3.1 Non-uniform degree distribution in scale-free graphs

The performance of graph traversal is not only determined by the algorithm, but also by the topology of the graphs. In this work, we focus on the scale-free graph, which is one of the most important categories of the large-scale graph. A few examples of scale-free graphs include social networks, computer networks, financial networks, and protein-protein interaction networks. In a scale-free

Table 1: Comparisons of the number of edge checks with different method using *bottom-up* method. We also list the number of using *top-down* for reference

Step	Top-down	Bottom-up(Random)	Bottom-up(Desc)	Bottom-up(Asec)
2	346918235(25x)	52677691(3.9x)	**13455687**(1x)	91365756(6.79x)
3	1727195615(195x)	10568751(1.19x)	**8820854**(1x)	11065150(1.25x)
4	29557400(286x)	153245(1.48x)	**103184**(1x)	203844(1.97x)
5	82357(3.83x)	21467(1x)	**21467**(1x)	21467(1x)
Total	2103753607 (92.3x)	63421157 (2.79x)	**22701186**(1x)	102656217(4.52x)

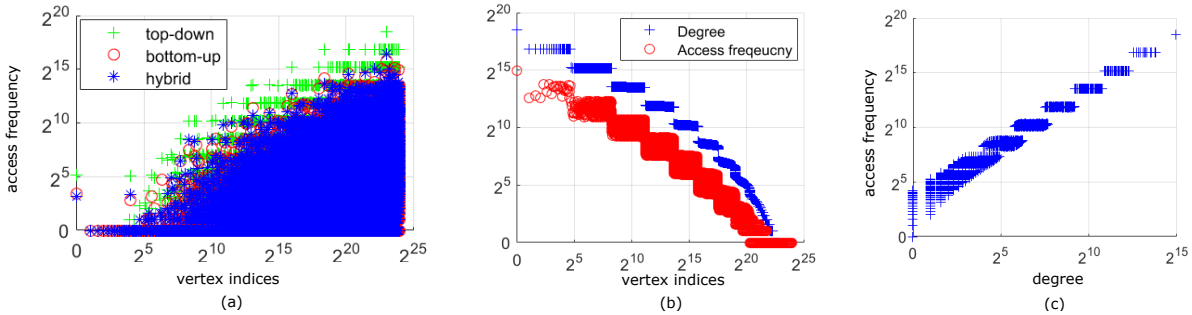

Figure 3: (a)Degree Access frequency distribution on unsorted graph; (b) Degree and access frequency of sorted vertex; (c) Strong correlation between access frequency and degrees

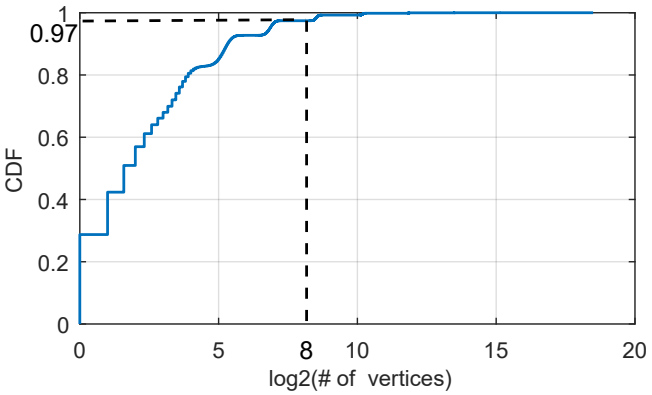

Figure 4: Cumulative distribution of vertex degree for rgg_s24_e16

graph, most vertices have only a small number of neighbors, and can reach others with a small number of hops. A scale-free graph typically has a degree distribution which follows a power-law, at least asymptotically [1, 24]. To show such power-law distribution, we plot the cumulative distribution of vertex degree of a Kronecker generated graph, which is a scale-free graph, as defined in GRAPH500 benchmark suite [16]. We can clearly see that 0.002% vertices with the largest degree contribute more than 97 % of the total edge connections.

Such a non-uniform degree distribution of scale-free graph is the basis of our optimizations. The degree distribution provides important information about the graph topology on which vertices have more connection to others. The more connection a vertex has, it is more likely that its status will be checked during the graph traversal. Moreover, it is trivial to obtain the degree information. For that, we can subtract the two nearby elements of the vertex indices list, which is the data structure to store the edge information and will be shown in section 4. Leveraging the degree information, we can reduce the redundant calculation, as well as the memory access. In section 3.2, we show the degree-aware adjacency list reordering technique, which can reduce redundant edge checks by terminating the *bottom-up* edge checks at an earlier stage. In section 3.3, we show the strong correlation between access frequency and degree of vertices, which can be used to guide the data placement in the hardware design (section 5).

3.2 Degree-aware adjacency list reordering

In this section, we propose to sort the adjacency list based on the vertices degree in the *bottom-up* method to further reduce the number of the redundant edge checks. As discussed in section 2.1, the *bottom-up* method scans the neighbors of all unvisited vertices and terminates earlier when one neighbor is found in the frontier. The timing of the early termination, which indicates how many edge checks it can save, is determined by the order of checking the status of neighbors (order in the adjacency list). In the best case, for all unvisited vertices scanned in the *bottom-up* method, the first neighbor to check is in the frontier. In this case, there are no redundant edge checks, as it checks only one edge and skips all the others. However, it is non-trivial to obtain the optimal order for the adjacency list. The reason is that we need to run the graph traversal first to get the vertices in the frontier in each step. Moreover, the optimal order may not be the same for different source vertices, as the frontier in each step is different. Instead of finding the optimal

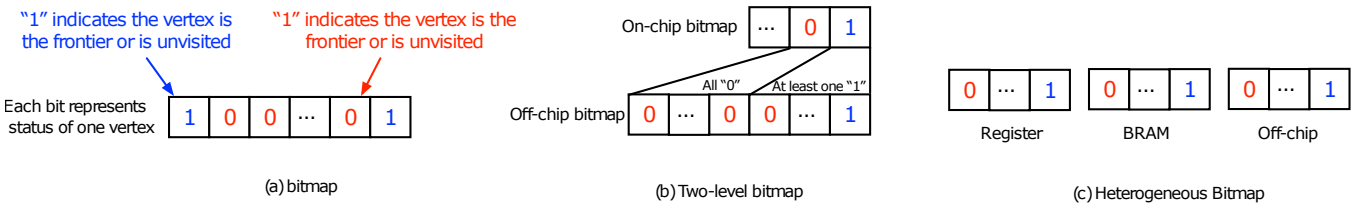

Figure 5: Illustration of bitmap

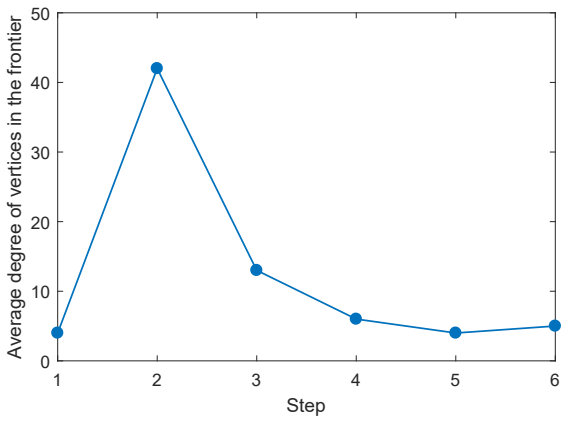

Figure 6: Average degree of vertices in the frontier of each step

order of the adjacency list , we propose a heuristic method, which sorts the adjacency list of each vertex based on the degree.

In table 1, we compare the number of checked edges for each step in both *top-down* and the *bottom-up* method using three different orders: **Descending order**, **Ascending order** and **Random order**. The table 1 shows that descending order can reduce the total number of edge checks and saves the most edge checks in step 2, which contributes most of edge checks. The reason is that veritices with higher degree tend to be visited in the first step in the *bottom-up* method. For example, as shown in figure 6, the average degree of the frontier of the step 2 is substantially larger than other steps. By sorting the adjacency list in the **descending order**, vertices in the frontier of the step 2, which have larger degree, are checked earlier than other vertices with lower degree. Though the number of reduced edge checks by using the **descending** compared to the **random** and the **ascending** order in other steps is not as large as the step 2, the adjacency list sorting still can reduce a substantial number of redundant edge checks, as the step 2 claims the most vertices.

3.3 Sorting Vertex Indices by degree

The locality in the scale-free graphs is considered weak due to its nature of sparsity and randomness[3]. In this section, we identify the relationship between degree and access frequency, which can be used to guide our hardware design.

More specifically, by sorting the vertex indices based on the degree in the descending order. We assign lower indices to vertices with high-degree and vice versa. To check how the vertices sorting affects the locality, we plot the relationship between vertex indices and access frequency on the unsorted graph and sorted graph in figure 3 (a) and (b) separately. We can see figure 3(b) shows the correlation clearly. We further plot the correlation between vertex degree and the access frequency in (figure 3 (c)). By sorting the graph, it is possible to know the degree of vertices without counting its neighbor. Moreover, we can find a strong correlation between vertex degree (indices) and access frequency, which could be used to guide the data placement. Such strong correlation is very useful to us, as it connects a run-time determined statistic (access frequency) to an off-line known property (degree distribution). Therefore, we can predict the access frequency before running the program. We can place data with different access frequency to different types of memories to maximize the memory access efficiency. We will discuss such software-hardware co-optimization in section 5.2 and section 5.3.

4 SOFTWARE IMPLEMENTATION

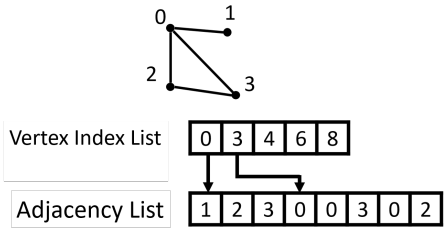

Figure 7: Illustration of adjacency list

As discussed in section 2.1, the *top-down* method needs to scan the frontier, read neighbor indices from the adjacency list ,and check the status of vertices. The *bottom-up* method follows the opposite order, which scans the status of vertices, reads the neighbor indices from the adjacency list and checks the frontier.

We first show the data structure for the frontier and the status of vertices. The key difference between scanning and checking is that the scanning reads the whole data structure *sequentially* and the checking reads the data structure *randomly* based on the indices reads from the adjacency list. The checking is considered costly, as random memory accesses have small granularities (sometimes 1 bit).

In this work, we use bitmaps, for bookkeeping both the frontier and the status of vertices, as [25] has already shown that the bitmap is a compact data structure for the frontier and can speed

up the scanning. Figure5 (a) illustrates the idea of bitmap. Each bit in the bitmap indicates whether the corresponding vertices is in the frontier or has been already visited. Different from [25], which only accelerates the scanning, this work also leverage the bitmap to accelerate the checking for both the frontier and status of vertices. Between these two types of bitmap check, the frontier bitmap check is more critical to the performance. The reason is that the hybrid algorithm tends to have more edges checked in the *bottom-up* method compared to the *top-down* method, as listed in table 1. Hence, we choose to reduce the cost of the frontier bitmap reading in the *bottom-up* step by leveraging the correlation between vertex degree and access frequency (section 2.3). We will provide more details in section 5.2.

The other data structure used in the hybrid graph traversal is the adjacency list (figure 7), which stores all the edge information. Reading the adjacency list is also costly, as it has a large size and can only be stored off-chip. The reason is that each element in the adjacency list is an vertex index, as shown in figure5, and typically has 32 bits or 64 bits. In the hybrid graph traversal, both the *top-down* and *bottom-up* method needs to read each element of the adjacency list only once. Therefore, it is impossible to reuse the data to reduce the external memory traffic. Instead, we try to reduce the size of each element. In section 5.3, we present our technique of compressing the the vertex indices.

5 HARDWARE IMPLEMENTATION

In this section, we will first introduce the implementation of the hybrid graph traversal on the FPGA-HMC platform. Then, we will provide two degree-aware optimization techniques: degree-aware data placement and degree-aware pointer compression to further accelerate the graph traversal based on the degree information.

5.1 Hybrid graph traversal algorithm on FPGA-HMC platform

In this subsection, we will present details of our architecture for the optimized hybrid graph traversal. We show the architectural diagram of the proposed implementation of the hybrid graph traversal in figure 11. In contrast to the system architecture described in [25], which implements only *top-down* graph method, we add/modify several hardware components to adapt to the hybrid graph traversal algorithm. Particularly, we add a new pipeline to support the *bottom-up* method, as well as statistic counters and direction switching logic to support the optimized direction switching. To accelerate the scanning of vertex status bitmap in the *bottom-up* method, we adopt the two-level bitmap design to reduce the traffic to the external memory. Moreover, we introduce a new organization of the frontier bitmap, as the *bottom-up* method have massive random access to the frontier bitmap. Finally, we modify the *top-down* pipeline to adopt all the architectural changes above. The only component, which is from [25] is the interface design for reordering HMC requests. The details of each design component are described below:

- **Pipeline for *bottom-up* method:** We design a new pipeline for the *bottom-up* method, as it has a different data flow compared to the *top-down* method. As shown in figure 9(b), it first scans the vertices bitmap to find all the unvisited vertices.

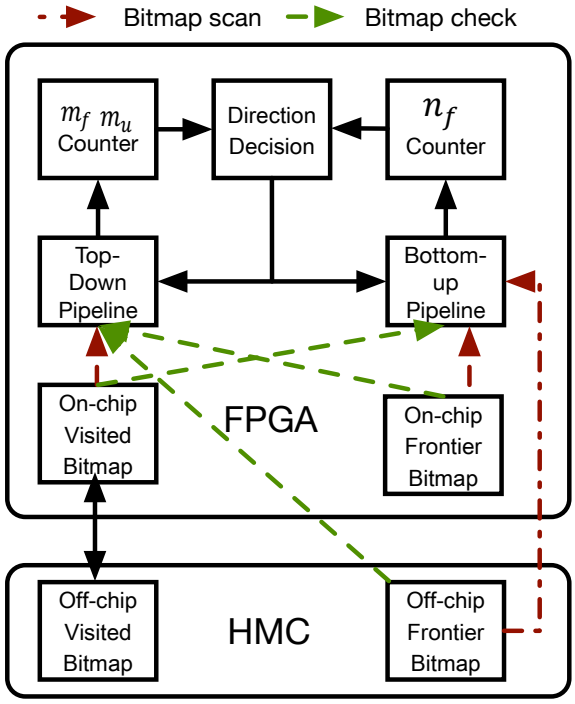

Figure 8: Architecture support for hybrid graph trveral

Then it reads the adjacency list from HMC to get the indices of neighbors to check. Finally, it checks all its neighbors to see if any of them is in the frontier. If so, the *bottom-up* pipeline marked the vertices as visited,and add it to the frontier of next step. As the *bottom-up* method only allows the child to update the visited flag by itself, the visited bitmap update doesn't need to be atomic.

- **Statistic counters:** We implement three counters to collect the statistics needed to support the direction switching: number of edges to check from the frontier(m_f), the size of the frontier(n_f) and the unexplored number of edges (m_u). These three statistics are calculated *sequentially* after each step in the CPU implementation[3] and the GPU implementation[22]. By taking advantage of the flexibility of FPGA, we implement three counters, which run in parallel to other components. These counters are updated when vertices are added to the frontier of next step. Therefore, we do not need an extra scan of the frontier after each step, which can take up to 20% of the total runtime in CPU implementation[2]. More specifically, the m_f and n_f are calculated by accumulating the degrees and the number of vertices when a vertex is added to the frontier of next step. The (m_u) is calculated by subtracting the sum of degrees of all visited vertices from the total number of edges.
- **Direction decision logic:** We implement a direction decision logic to support the optimized direction-switching. The direction decision logic compares the three statistics we collected from the last layer with the heuristic thresholds to

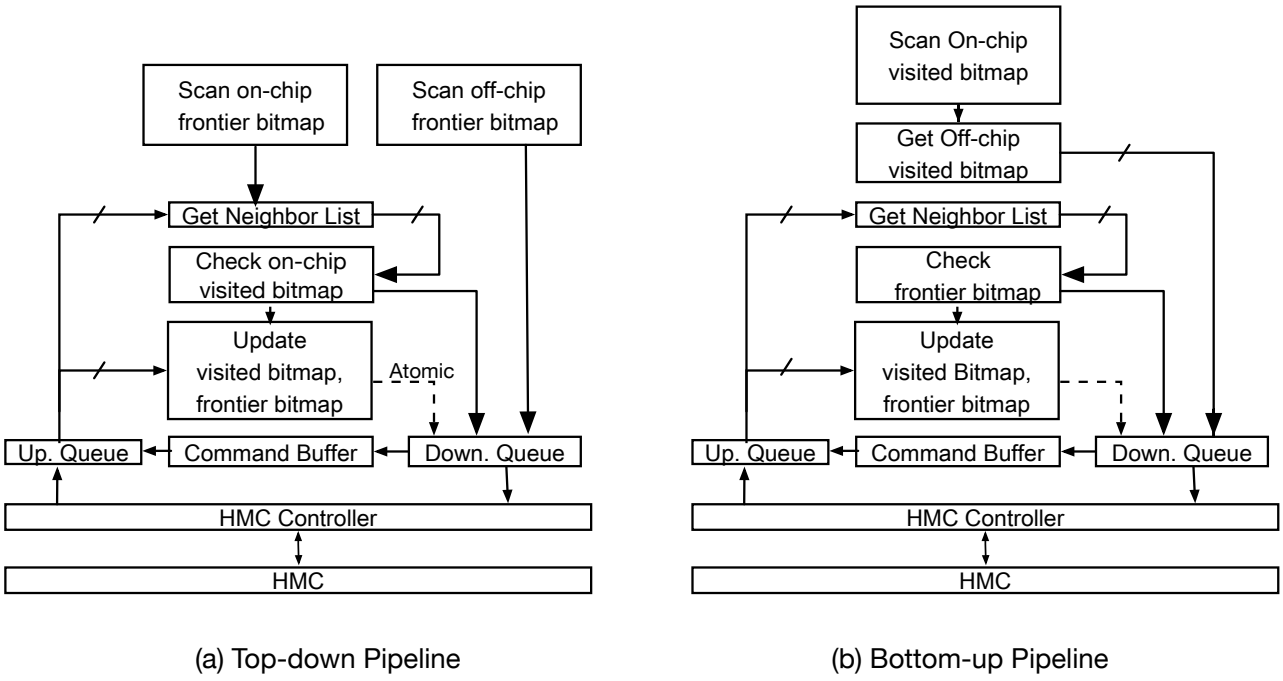

Figure 9: Data flow of the FPGA implementation of the (a) *top-down* (b) *bottom-up* graph traversal

determine whether *top-down* or *bottom-up* method should be invoked. We set these two thresholds to $\alpha = 15$ and $\beta = 20$, as in [3].

- **Bitmap of the status of vertices:** We use a two-level bitmap to store the unvisited vertices array instead of a single-level off-chip flag array (stores the parent of each vertices), as in Zhang *et al* [25]. As shown in Table 1, the number of unvisited vertices is relatively small, which makes the unvisited list very sparse. For every step using the *bottom-up* method, we need to scan the whole vertices status array to find all unvisited vertices. The sparsity of unvisited vertices leads a considerable amount of unnecessary external memory accesses. As discussed in Zhang *et al* [25], a two-level bitmap can accelerate the scanning of the frontier bitmap, which is also sparse in some steps. In this work, we adapt the two-level bitmap in [25] to the vertices status array. More particularly, as shown in figure 5 (b), the on-chip bitmap is initialized to all 0's at the beginning of the first *bottom-up* step. we first scan the on-chip bitmap to find the non-zero bits, which indicates there is at least 1 unvisited vertex in the corresponding vertices group. If the *bottom-up* method marks all of the vertices in this group visited, we set the on-chip bitmap to 1. During the *top-down* method, we do not check or update the on-chip bitmap. Checking whether all vertices in one group are visited in the *top-down* method has extra costs, as scanning the off-chip vertices status bitmap is not an essential step of *top-down* method. The vertices status bitmap is also used in the *top-down* method while checking whether a neighbor of the vertex in the frontier is visited or not.
- **Frontier bitmap:** In this work, we introduce a new heterogeneous bitmap organization, which can adapt to the access pattern of *bottom-up* method. Compared to the *top-down* method, which only needs to scan the frontier sequentially to find all the vertices to be checked, the *bottom-up* method access the frontier in a random parallel manner, as several unvisited vertices could have the same parent. As the two-level bitmap can only improve the efficiency of sequential bitmap scan during the *top-down* method, which only contributes a small portion of the total runtime, we choose to optimize the memory access in the *bottom-up* method. We use a heterogeneous bitmap to store the frontier, which can provide different random access performance. In the next subsection, we will further discuss the data-placement policy based on the relationship between degree and access frequency (section 3.3).
- ***Top-down* pipeline:** We also modify the *top-down* pipeline, as the memory organization of both visited veritices bitmap and the frontier bitmap has changed. Particularly, we first scan the heterogeneous bitmap to find the vertices that need to be checked. Then, we read the adjacency list from external memory and check whether neighbors are visited by reading the off-chip vertices status array. The unvisited vertices are marked through atomic updates to both the on-chip and HMC vertices status bitmaps as shown in figure 9a.

5.2 Degree-aware data placement

As discussed in section 4.1, the most performance critical operation is reading the frontier bitmap in the *bottom-up* method. In this section, we will show how to leverage the strong relationship between degree and access frequency (section 3.3) to guide the design of the frontier bitmap.

To take advantage of the correlation between vertex degree and access frequency, we introduce three types of bitmap to store the status of the vertices: register file, BRAMs and external HMC. The register file is bit addressable and is used to store the vertices with the highest degree. In our implementation, the size of the bitmap register is 4096 bits and has 64 read ports to support multiple simultaneous reads in one cycle. The second type of bitmap storage is stored in the on-chip BRAM, which stores the sorted vertices with indices from 2049 to 65536. From figure 3, we can see that the first 256 (0.002%) vertices contribute more than 97% of the edge checks in the rgg_s24_e16 graph. The rest of the visited vertices bitmap, which mainly consists of the vertices whose degrees are smaller than 4 (in the rmat graph dataset), can be stored off-chip.

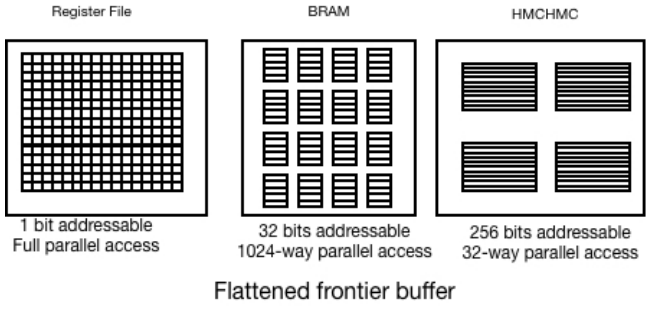

Figure 10: Flattened frontier buffer

5.3 Degree-aware adjacency list compression

To further leverage from non-uniform degree distribution, we propose to apply the coding technique to compress the adjacency list and reduce the runtime of neighbor accesses. The adjacency list stores the edge information on the external HMC, and will be accessed in both *top-down* and *bottom-up method*. Each element in the adjacency list is the index of a neighbor vertices. Vertices with higher degree will occurs more frequently in the adjacency list, and vice versa. For example , as shown in figure 7, vertex 0 occurs three time, and vertex 1 only occurs one time. However, prior works treat these vertices equally and use the same data format (e.g. uint32 or uint64), which is a waste of the storage and external memory bandwidth.

In this works, we apply a coding scheme to compress the adjacency list. By sorting vertices based on the degree in the descending order, as discussed in section 3.3, the vertices with lower indices have higher access frequency. In this case, Exp-Golomb coding [9], which is widely used in video compression [17] , can effectively compress the adjacency list to reduce the data access to the external memory. More specifically, It has been shown that Exp-Golomb coding has compression efficiency close to the more complex arithmetic coding, and comparable to Huffman coding, if the input integer follows the assumption below: the larger the integer, the lower its probability of occurrence[9], which is exactly the the vertices list after sorting. Also, we choose the Exp-Golomb coding as its complexity is much lower than Huffman coding as it does not need to construct and store the Huffman tree.

In our design, we enable Exp-Golomb for a scale-free graph, such as indochina and rmat. As a uniformly generated graph, such as rgg, does not have enough variation of access frequency, the coding will essentially increase the length of the adjacency list. Our hardware implementation follows the parallel GR decoder design in [15].

6 PERFORMANCE EVALUATION

In this section, we will first present our evaluation methodology, which includes the experimental platform setup and the choice of dataset. Then, we will present the experimental results of the proposed design and compare it with the baseline design. Finally, we will project the performance with full HMC bandwidth and compare with the latest GPU implementation.

6.1 Hardware Platform

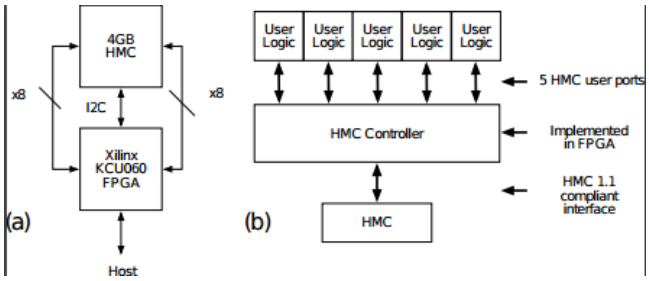

Figure 11: Diagram of experimental platform

We implement the proposed graph processor on an AC-510[20] and AC-520 FPGA-HMC platform from Micron. As shown in figure 11, the AC-510 platform has a Xilinx KCU060 FPGA and an 8GB HMC module. The HMC uses two half-width (8 lanes) 15G high-speed serial links, which provide a two-way bandwidth of 30GB/s, to communicate with the FPGA. The AC-520 platform has an Intel Arria 10 GX 1150 FPGA and a 4GB HMC module. The HMC interfaces with FPGA with four half-width links, which provide a two-way bandwidth of 60GB/s. The AC-520 also exposes the interface for power consumption measurement.

We implement our graph processing architecture using the PicoFramework, which provides an abstraction layer of the low level data transfer protocol. The kernel logic is driven by a 125 MHz clock, and on-chip memories are working at 250MHz in double-pump mode. The host machine is equipped with an Intel Xeon E5-1630V3 CPU and one DDR4 memory channel with a 16GB capacity. We use Ubuntu 14.04 as the host operating system and compile our CPU implementation using gcc with flags "-Ofast" and "-march=native." The GPU benchmark is running with a GTX Titan X graphics card with 12GB of GDDR5X VRAM. We summarize the resource utilization statistics in table 2. The high BRAM usage is

Table 2: FPGA resource utilization

	Available Resources	Proposed	Percentage
Logic	1506k	437k	64
BRAM	2713	2062	76
DSP	1518	64	4

because we hope to absorb as much off-chip memory access as possible by using a larger hierarchical frontier bitmap.

6.1.1 Datasets. We summarize the datasets used in our evaluations in table 3. Soc-orkut is a social graph; indochina-04 is a crawled hyperlink graph from indochina web domains; rmat_s22_e64, rmat_s23_e32, and rmat_s24_e16 are three generated R-MAT graphs with similar vertex counts. All five datasets are scale-free graphs with diameters less than 30 and unevenly distributed node degrees (80

Both rgg_n_24 and roadnet_usa datasets have large diameters with small and evenly distributed node degrees (most nodes have degree less than 12). soc-ork is from the Stanford Network Repository; Indochina-04 and roadnet are from the UF Sparse Matrix Collection; rmat_s22_e64 rmat_s23_e32, rmat_s24_e16, and rgg_n_24 are R-MAT and random geometric graphs we generated. For R-MAT, we use 16 as the edge factor, and the initiator parameters for the Kronecker graph generator are: a = 0.57, b = 0.19, c = 0.19, d = 0.05, which follows the Graph 500 Benchmark. For random geometric graphs, we set the threshold parameter to 0.000548.

Table 3: Test dataset (all directed graphs have been converted to un-directed graphs)

Dataset	Vertices	Edges	Max Degree	Diameter
soc-orkut	3M	212.7M	27,466	9
indochina-04	7.4M	302M	256425	26
rmat_s22_e64	4.2M	483M	451607	5
rmat_s23_e32	8.4M	505.6M	440396	6
rmat_s24_e16	16.8M	519.7M	432152	6
rgg_n_24	16.8M	265.1M	40	2622
roadnet_USA	23.9M	577.1M	9	6809

6.2 Experimental Results

We first compare the performance gain of the CPU-DRAM hybrid graph traversal algorithm and the FPGA+HMC based implementation in figure 12. The throughput gain of the proposed FPGA-HMC system is higher than the CPU-DRAM system[3] on the five scale-free graph datasets since the proposed FPGA-HMC system is optimized for the multiple random accesses of the frontier bitmap in the *bottom-up* graph traversal. Also, the five scale-free graphs with lower diameter have a large frontier size, which leads to more *bottom-up* method than *top-down* during the graph traversal and can benefit more from our optimization to the *bottom-up* method.

On the contrary, the performance gain difference on the scaled graph is relatively low, since nearly all steps use *top-down* method, which is hard for parallelization. We should note that most of the emerging graph analytics workloads, such as social networks, web, and communication networks are scale-free graphs[4].

In figure 13, we compare the performance gains of the three optimizations: degree-aware adjacency list reordering , degree aware

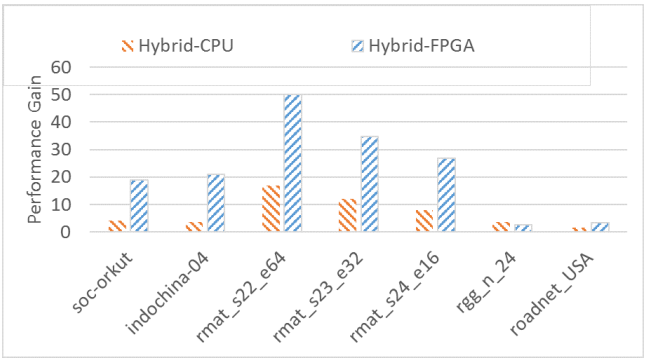

Figure 12: Performance gain comparison of CPU+DRAM and proposed FPGA+HMC implementation of hybrid graph traversal system

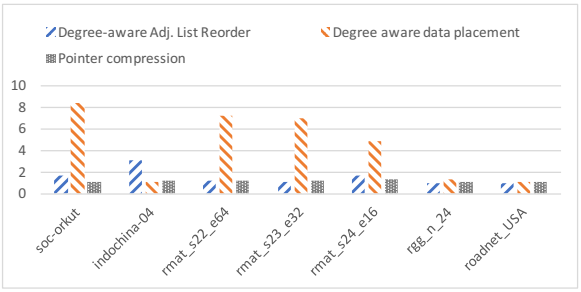

Figure 13: 8 Performance gain for degree-aware adjacency list reordering, degree-aware index sorting and degree-pointer compression

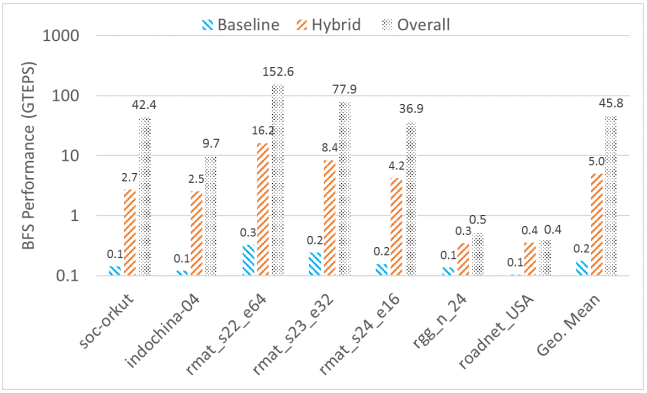

Figure 14: Graph traversal performance comparison

index sorting, and degree-aware pointer compression. We find that the degree-aware data-placement contributes to the major portion of the performance gain since the random access to the frontier bitmap is the most costly memory access in the *bottom-up* operation. Similar to the result in figure 10, the degree-aware data-placement also has fewer benefits in the rgg and roadnet graph due to its degree distribution being nearly uniform.

We summarize the overall graph traversal performance of the proposed system in figure 14 and compare it with the hybrid-only

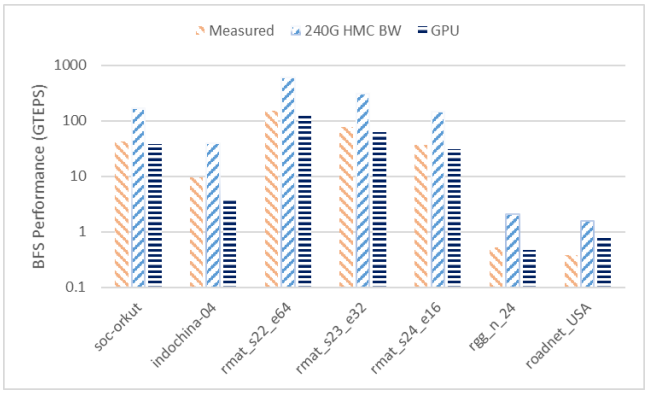

Figure 15: BFS performance projection and comparison with GPU

implementation and the baseline [3], which is the fastest existing FPGA-based graph traversal system. The geometric mean over the seven datasets is 45.8 GTEPS, which is significantly faster than the baseline design and the native implementation of hybrid graph traversal on an FPGA.

We further measure the graph traversal performance with 120GB/s HMC bandwidth on AC-520 system. We can achieve 79.8 GTEPS, which nearly doubles the performance compared to AC-510 platform. The power consumption of AC-520 board is 43.6 watts, which gives a power efficiency of 1.85 GTEPS per watt.

We further projects the performance on the system with the largest FPGA devices, XCKU115[23] and the HMC module, which can achieve the full two-way bandwidth of 240GB/s[19], and compare it with best single-GPU graph traversal performance[22]. As shown in figure 15, the proposed system achieves better performance than GPU for scale-free performance. By using the larger device with full HMC bandwidth, the proposed system can also outperform GPU on scaled graphs.

7 CONCLUSION

In this work, we present a high-performance graph traversal framework, which implements and optimizes the hybrid graph traversal on an FPGA-HMC platform. In particular, we first identify the improvement space of state-of-the-art hybrid graph traversal and provide two techniques to optimize the algorithm: degree-aware adjacency list reordering and degree-aware vertices sorting. Then, we introduce the implementation of the optimized hybrid traversal algorithm on an FPGA-HMC platform and provides two hardware optimization. Finally, we conduct experiments on eight different datasets to verify the effectiveness of the proposed techniques and provide a performance projection with higher HMC bandwidth. Our implementation on the AC-510 development board from Micron achieves 45 GTEPS, outperforming CPU and other FPGA-based large-scale graph processors, and can be compared with the latest GPU graph processing library.

ACKNOWLEDGEMENTS

We appreciate the insightful comments and feedback from the anonymous reviewers. We thank Micron for the donation of the development tool and hardware. We especially thank John Watson and Mark Hur for their support.

REFERENCES

[1] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *science* 286, 5439 (1999), 509–512.
[2] Scott Beamer, Krste Asanovic, and David A Patterson. 2011. Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500. (2011).
[3] Scott Beamer, Aydin Buluc, Krste Asanovic, and David Patterson. 2013. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 1618–1627.
[4] Béla Bollobás, Oliver Riordan, Joel Spencer, Gábor Tusnády, et al. 2001. The degree sequence of a scale-free random graph process. *Random Structures & Algorithms* 18, 3 (2001), 279–290.
[5] Aydin Buluç and Kamesh Madduri. 2011. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 65.
[6] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *ACM/SIGDA FPGA (FPGA '16)*.
[7] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 217–226. https://doi.org/10.1145/3020078.3021739
[8] P. F. Felzenszwalb and R. Zabih. 2011. Dynamic Programming and Graph Algorithms in Computer Vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 4 (April 2011), 721–740. https://doi.org/10.1109/TPAMI.2010.135
[9] Solomon Golomb. 1966. Run-length encodings (Corresp.). *IEEE transactions on information theory* 12, 3 (1966), 399–401.
[10] 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 *ACM SIGKDD*. ACM.
[11] T. H. Haveliwala. 2003. Topic-sensitive PageRank: a context-sensitive ranking algorithm for Web search. *IEEE Transactions on Knowledge and Data Engineering* 15, 4 (July 2003), 784–796. https://doi.org/10.1109/TKDE.2003.1208999
[12] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: large-scale graph computation on just a PC. In *USENIX OSDI*.
[13] Guoqing LEI, Rongchun LI, Song GUO, and Fei XIA. 2016. TorusBFS: A Novel Message-passing Parallel Breadth-First Search Architecture on FPGAs. (10 2016).
[14] Hang Liu and H Howie Huang. 2015. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 68.
[15] Roger Moussalli, Walid Najjar, Xi Luo, and Amna Khan. 2013. A high throughput no-stall golomb-rice hardware decoder. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. IEEE, 65–72.
[16] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. (2010).
[17] Savita Nargundmath and Archana Nandibewoor. 2013. Entropy coding of H. 264/AVC using Exp-Golomb coding and CAVLC coding. In *Advanced Nanomaterials and Emerging Engineering Technologies (ICANMEET), 2013 International Conference on*. IEEE, 607–612.
[18] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, José F. Martínez, and Carlos Guestrin. 2014. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *IEEE FCCM*.
[19] J Thomas Pawlowski. 2011. Hybrid memory cube (HMC). In *IEEE Hot Chips*.
[20] Picocomputing. 2016. UltraScale-based SuperProcessor with Hybrid Memory Cube. http://picocomputing.com/ac-510-superprocessor-module. (2016).
[21] John Scott. 2017. *Social network analysis*. Sage.
[22] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. 2017. Gunrock: GPU Graph Analytics. *arXiv preprint arXiv:1701.01170* (2017).
[23] Xilinx. 2011. Ultrascale Plus Fpga Product Selection Guide. https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf. (2011).
[24] Yuichiro Yasui, Katsuki Fujisawa, Eng Lim Goh, John Baron, Atsushi Sugiura, and Takashi Uchiyama. 2016. NUMA-aware scalable graph traversal on SGI UV systems. In *Proceedings of the ACM Workshop on High Performance Graph Processing*. ACM, 19–26.
[25] Jialiang Zhang, Soroosh Khoram, and Jing Li. 2017. Boosting the Performance of FPGA-based Graph Processor using Hybrid Memory Cube: A Case for Breadth First Search.. In *FPGA*. 207–216.
[26] Shijie Zhou, Charalampos Chelmis, and Viktor K Prasanna. 2016. High-throughput and energy-efficient graph processing on fpga. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*. IEEE, 103–110.