High Performance Graph Analytics with Productivity on Hybrid CPU-GPU Platforms

Haoduo Yang[1, 2], Huayou Su[1, 2], Qiang Lan[1, 2], Mei Wen[1, 2] and Chunyuan Zhang[1, 2]
[1] Department of Computer, National University of Defense Technology, Changsha 410000, China
[2] National Key Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha 410000, China
{yhd1801105910, huayousu, lanqiang_nudt}@163.com,{meiwen,cyzhang}@nudt.edu.cn

ABSTRACT
In recent years, the rapid-growing scales of graphs have sparked a lot of parallel graph analysis frameworks to leverage the massive hardware resources on CPUs or GPUs. Existing CPU implementations are time-consuming, while GPU implementations are restricted by the memory space and the complexity of programming. In this paper, we present a high performance hybrid CPU-GPU parallel graph analytics framework with good productivity based on GraphMat. We map vertex programs to generalized sparse matrix vector multiplication on GPUs to deliver high performance, and propose a high-level abstraction for developers to implement various graph algorithms with relatively little efforts. Meanwhile, several optimizations have been adopted for reducing the communication cost and leveraging hardware resources, especially the memory hierarchy. We evaluate the proposed framework on three graph primitives (PageRank, BFS and SSSP) with large-scale graphs. The experimental results show that, our implementation achieves an average speedup of 7.0X than GraphMat on two 6-core Intel Xeon CPUs. It also has the capability to process larger datasets but achieves comparable performance than MapGraph, a state-of-the-art GPU-based framework.

CCS Concepts
• Theory of computation → Models of computation → Concurrency → Parallel computing models • Theory of computation → Design and analysis of algorithms → Graph algorithms analysis.

Keywords
Parallel Computing; Graph Analytics; Hybrid CPU-GPU

1. INTRODUCTION
Graph computing has become important for analyzing data in many application domains, such as bio-informatics, social networking, and simulations. During the last decade, for dealing with large-scale graphs, various parallel graph computing frameworks have been proposed to leverage modern massively parallel processors, e.g. CPUs or GPUs. CPU-based frameworks get the productivity benefits of programming. While GPU-based implementations usually have better performance. However, they are limited by the GPU's memory size for processing larger graph. It is also a challenge to write efficient programs on GPUs for general graph algorithms.

To overcome the limitations of existing frameworks, we propose a high performance graph computing framework on hybrid CPU-GPU platforms. Based on GraphMat [1], an advanced CPU-based graph framework, we map vertex programs to sparse matrix vector multiplication (SPMV) by CUDA kernel, and incorporate several optimization techniques for the GPU architecture. Specifically, we provide compression and decompression methods to reduce the communication time and storage requirement effectively. Meanwhile, we adopt memory access optimizations to reduce the memory access latency. In addition, our high-level abstraction brings good productivity for programming. Our contributions are as follows:

1. Based on GraphMat [1], we implement an efficient hybrid CPU-GPU computing framework which maps vertex programs to SPMV on GPUs. With high-level abstraction, this framework is designed for developers to implement various graph algorithms with good productivity.

2. We perform several optimization techniques (e.g. compression, decompression) to reduce the communication cost and improve the memory accessing efficiency by taking the advantage of the hierarchical GPU memory.

3. We evaluate the proposed framework for several graph algorithms with large scale datasets. The experimental results show that our framework achieves higher performance than both advanced CPU-based and GPU-based framework (GraphMat and MapGraph respectively).

The rest of the paper is organized as follows. Section 2 presents the existing graph frameworks and the motivation of our work. Section 3 describes our implementation and optimizations in detail. Section 4 gives the results of measuring performance with graphs, while Section 5 concludes the paper and discusses future research directions.

2. BACKGROUND AND MOTIVATION

2.1 Graph Analytics Frameworks
For large-scale graph analytics, several frameworks have been developed and tuned in industry and academia with different algorithm models on CPU or GPU platforms. Single-node CPU-based frameworks are in common use for graph computation. Among them, vertex programming are quite widely applied, such as GraphLab [2], Pregel [3] and GraphX [4]. High-level GPU-based frameworks for graph analytics often learn from CPU programming models. For example, Zhong *et al.* introduced

HP3C 2018, March 15–17, 2018, Hong Kong, Hong Kong

ACM ISBN 978-1-4503-6337-2/18/03…$15.00
https://doi.org/10.1145/3195612.3195614

Medusa [5] , a high-level GPU-based system for parallel graph algorithms using Pregel's [3] messaging model. MapGraph [6] adopt PowerGraph's Gather-Apply-Scatter (GAS) programming model [7]. MapGraph [6] obtains some of the best performance results among programmable models on single-node GPU.

With the increasing graph scales of over a ten million nodes and billion edges, CPU-based graph frameworks still have ability to allocate enough memory for storage and processing, but the reductions in processing performance are significant. Comparing to them, existing GPU-based graph frameworks usually gain better performance due to the strengths of hardware, the generalized load balance strategies and optimized GPU primitives. Nevertheless, almost they are programmed with relatively low productivity, since developers or users still have to design data structures and CUDA kernels carefully for a specific algorithm [6]. Meanwhile, because GPU memory capacity is limited, it is a challenge to employ it efficiently, particularly for processing large graphs. The limitations of these frameworks motivate us to design a better implementation.

2.2 GraphMat

Sparse matrix operation is the key primitive for some high performance CPU-based graph analytics frameworks, such as GraphMat [1], CombBLAS [8] and PEGASUS [9]. Actually, a lot of graph computing algorithms (such as PageRank, HITS, Random Walk with Restart) can be described as an iterative sparse matrix vector multiplication (SPMV) [10], one of the most important operations in high performance computing (HPC).

Algorithm 1. Overall framework of GraphMat

Algorithm 1 GraphMat Overview. x,y are both sparse vectors

```
1:  function RUN_GRAPH_PROGRAM(Graph G , GraphPro-
    gram P)
2:      for i = 1 → MaxIterations do
3:          for v = 1 → Vertices do
4:              if v is active then
5:                  x_v ← P.SEND_MESSAGE(v, G)
6:              end if
7:          end for
8:      end for
9:      y←SPMV(G,x,P.PROCESS_MESSAGE,P.REDUCE)
10:     Reset active for all vertives
11:     for j = 1 → y.length do
12:         v ← y.getVertex(j)
13:         old_vertexproperty ← G.getVertexProperty(v)
14:         G.setVertexProperty(v, y.getValue(j), P.APPLY)
15:         if G.getVertexProperty(v) ≠ old_vertexproperty then
16:             v set to active
17:         end if
18:     end for
19:     if Number of active vertices == 0 then
20:         break
21:     end if
22: end function
```

Narayanan *et al.* developed GraphMat [1], a single-node multicore graph framework running on CPU. As shown in Algorithm 1, GraphMat adopts an iterative process, and each iteration includes three parts: SEND_MESSAGE (lines 2-8), SPMV (line 9), and APPLY (lines 11-18). It converts a graph to an adjacency matrix G represented in a sparse matrix format. According to different algorithms, GraphMat assigns different property data to vertex. SEND_MESSAGE sends useful messages to all active nodes (those are marked to be active in the last iteration) along in-edges, out-edges or both, and creates a sparse vector x . G and x are the input of a generalized SPMV, which produces a new sparse vector y as output. It should be noticed that, GraphMat adopts PROCESS_MESSAGE and REDUCE, a series of operations defined by developers based on algorithms, to replace original multiplication and addition on each vector dot product computation during SPMV. After that, APPLY phase updates the properties of vertex using y , and marks those nodes whose properties are set to be active.

GraphMat [1] shows that, a vertex-based programming model can be established on top of a matrix backend. While GPUs, offering significantly higher memory bandwidth and very high peak computational throughput than CPUs [11], have the potential to accelerate SPMV since its memory-bound nature. Although many high quality library implementations are readily available, so far as we know, existing SPMV-based graph analytics on GPU achieves nowhere near the same performance as these optimized libraries. Basing on that abstraction proposed by GraphMat, our goal is to complete a hybrid CPU-GPU framework with high productivity (like vertex programming for users) and high performance (optimized sparse matrix backend).

3. IMPLEMENTATION

Since we try to avoid allocating or managing complex data structures on GPU memory with the consideration that SEND_MESSAGE and APPLY phases in GraphMat are embarrassingly parallel, we focus on implementing high performance kernels for mapping vertex programming to SPMV on GPUs and some optimization methods. In this section, we provide an adequate description of our design, including data structure, generalized SPMV, compression, decompression, optimizations for memory access, and computation pipeline.

3.1 Data Structure

We choose Compressed Sparse Row (CSR) data structure to store large-scale graphs, which is one of the most popular, general-purpose sparse matrix formats. For an *M-by-N* matrix A with N_{nz} nonzeros, the CSR formats is formed by two arrays: *row-offsets* and *column-indices*. The *column-indices* array is formed from the concatenation of the adjacency lists of the graph. The *row-offsets* array has size $M+1$ and includes the starting offset of the *i*-th row with storing N_{nz} in its last entry. Besides, CSR has an array *value* of size N_{nz} for storing all the non-zero values of matrix in row-major order.

3.2 Generalized SPMV on GPU

SPMV acceleration using GPUs has been studied extensively in the past. Efficient implementations require fine-grained parallelism to effectively utilize the computational resources of the GPU. Yongchao Liu *et al.* introduced LightSPMV in the [12], a parallelized CSR-based SPMV implementation written in CUDA C++. It assigns $32/V$ consecutive rows to all vector within a warp, and each vector contains V lanes $(V \in \{2,4,8,16,32\})$ .We refer its idea and propose modified SPMV on GPU for completing various algorithms.

Algorithm 2 illustrates our generalized SPMV kernel. The input data includes a CSR-formatted matrix, a sparse vector *x* and a bit vector *x_bitvector* . A bit vector is used to reduce the size of a sparse vector status from a 32-bit integer to a single bit per vertex, providing convenience for caching valid values effectively and reducing redundant computing. Similarly, the result of SPMV kernel stores in two arrays: y and *y_bitvector* . The whole kernel can be divided into three parts: the first part (from line 2 to 4) generates a row index *i* for the thread running the kernel currently. The second part (from line 7 to 12) gives the position

range of non-zeros in the *i*-th row. In the third part (from line 13 to 30), we judge the status of vectors by bit vectors and execute PROCESS_MESSAGE and REDUCE operations which are specified by users instead of traditional vector dot product. For example, for Single Source Shortest Path (SSSP), *yval* should be initialized to infinity. While PROCESS_MESSAGE function computes the length of a new possible path using an addition operation. REDUCE function chooses the smaller distance from two neighboring vertices. After that, the thread retrieves another new row index. This provides flexibility needed for adequately supporting general graph algorithms.

Algorithm 2. Generalized SPMV kernel in our framework

Algorithm 2 SPMV kernel

```
1: function SPMV_KERNEL(rowOffsets, colIndexValues, numeri-
   calValues, x, x_bitvector, y, y_bitvector)
2:     vecLandId ← threadIdx.x%V;
3:     vectorId ← threadIdx.x/V;
4:     row ← getRowIndex();
5:     __shared__ int space[1024/V][2];
6:     while row < R do
7:         y_row_exist ← false;
8:         if vecLandId < 2 then
9:             space[vectorId][vecLandId]←rowOffsets[row
   +vectorId];
10:        end if
11:        row_start ← space[vectorId][0];
12:        row_end ← space[vectorId][1];
13:        initialize yval;
14:        for i=row_start+vecLandId;i<row_end;i+=V do
15:            if x[colIndexValues[i]] is a non-zero then
16:                yval=PRCOCESS_MESSAGE(numericalValues[i],
   x[colIndexValues[i]]);
17:                if y_row_exist is fasle then
18:                    set index row to 1 in y_bitvector
19:                    y_row_exist ← true;
20:                end if
21:            end if
22:        end for
23:        if y_row_exist is true then
24:            for i=V>>1; i>0; i>>=1 do
25:                yval←REDUCE(__shfl_down(yval,i, V));
26:            end for
27:            if vecLandId == 0 then
28:                y[row]←yval;
29:            end if
30:        end if
31:        row←getRowIndex();
32:    end while
33: end function
```

3.3 Compression and Decompression

Although the proposed generalized SPMV kernel provides a faster matrix operation than original CPU-based implementation, and yields higher performance for some general graph algorithms, it still has several drawbacks. As mentioned above, for executing each SPMV, we have to transfer two arrays: x and $x\_bitvector$ into GPU memory as input, and copy y and $y\_bitvector$ back to CPU for the following APPLY phase. We use GraphMat to run Breadth First Search (BFS) on some graphs (refer to Section 4 for more details). For the input vector x and output vector y in each SPMV computation, we gather the number of their non-values respectively and summarize the results in Figure 1. And we observe that, all curves tend to rise up at the beginning and decline in late. Actually, it is the same thing when testing many other algorithms. This means that transferring these sparse vectors storing plenty of invalid zeros directly will render a lot of communication cost and redundant memory.

In this regard, we introduce compression and decompression. We compress a sparse vector into two arrays, storing all non-zeros and their indices respectively. After transferring them, we get the original data through decompression. Considering an extreme case, if vector x of size l has only one non-value, using the bit-vector representation mentioned above will lead to transferring an array of size l and a bit vector of size $(l >> 4)$. By contrast, after compression, we need merely to store and send two arrays of size 1, which results in significant reductions in communication time and storage space they required. In addition, we complete compression and decompression effectively with parallel computation, which greatly reduces their time overhead.

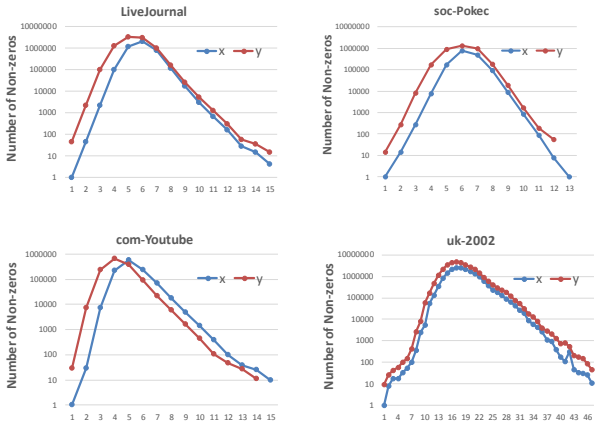


Figure 1. The number of non-values (in log-scale) of input vector *x* and output vector *y* within SPMV for computing BFS.

3.4 Memory Access Optimizations

CUDA devices are highly memory latency bound and have several kinds of addressable memory (i.e. registers, shared memory, constant memory, texture memory, local memory and global memory). We utilize the follow methods to optimize device memory access.

1. Before the first iteration, we allocate global memory for storing the CSR-formatted sparse matrix as it is read-only and constant. This memory would not be free until the last iteration is finished. And we load them via the 48KB read-only cache on Kepler, which can benefit the performance of bandwidth-limited kernels.

2. The input arrays of SPMV are read-only for the entire lifetime of kernels. Due to the frequent and random access to them in SPMV kernels, we decide to store these arrays in texture memory which can improve performance and reduce memory traffic.

3. We assign one block (containing 1024 threads) to execute kernels for compression and decompression respectively. Shared memory can be adopted within these kernels so that all threads within a block can communicate and collaborate on computations. As a result, the latency to access shared memory tends to be far lower than local or global memory for getting better performance.

3.5 Computation Pipeline

The computation pipeline of our proposed framework is shown in Figure 2. In GRAPH_PREPROCESS phase, it converts a large graph into a CSR-based matrix and stores it in GPU memory. In each iteration, similar to GraphMat, a sparse vector with messages of those active vertices and a bit vector are generated by SEND_MESSAGE operation firstly. Then, it executes a phase named INPUT_PROCESS, including compression on CPU, transferring the compressed results to GPU memory and decompression on device. After that, the framework obtains the input data of SPMV. An efficient SPMV kernel is performed

which takes most running time. In OUTPUT_PROCESS stage, the resulting vectors are compressed on GPU and decompressed on CPU. While the state of vertices are updated in APPLY phase. Those changed vertices will be marked active for the next iteration. The graph algorithm will continue until convergence or reach the maximum number of iterations defined by users. For implement a graph algorithm, developers merely need to initialize the property data of vertices, and define a fixed number of functions and *yval* in SPMV kernel (mentioned in Section 3.2). Such a high-level abstraction can raise productivity for our framework.

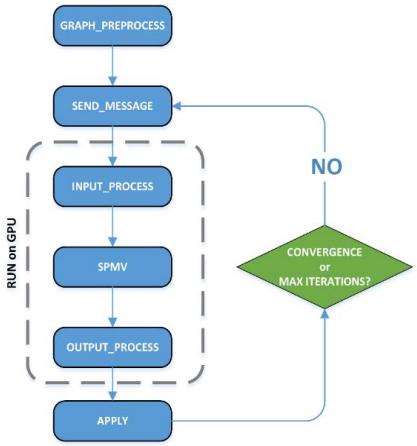


Figure 2. Computation Pipeline of Our Framework. Dotted line marks the computations on GPU.

4. EVALUATION

To show the performance and productivity of the proposed framework based on hybrid CPU-GPU platforms, we implement three common graph analytics algorithms: PageRank, Breadth First Search (BFS) and Single Source Shortest Path (SSSP). In this section, we report performance results and compare them to both advanced CPU-based and GPU-based implementations (GraphMat and MapGraph respectively).

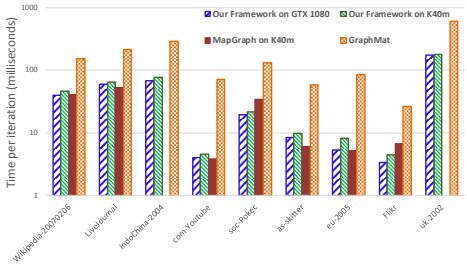
Table 1. Information of datasets

Dateset	#Vertices	#Edges	Algorithms
Wikipedia-20070206[13]	3,566,907	45,030,389	PageRank
LiveJournal[13]	4,847,571	68,993,773	PageRank, BFS, SSSP
indoChina-2004[13]	7,414,866	194,109,311	PageRank, BFS, SSSP
com-Youtube[14]	1,134,890	2,987,624	PageRank, BFS, SSSP
soc-Pokec[14]	1,632,803	30,622,564	PageRank, BFS, SSSP
sx-stackoverflow[14]	2,601,977	63,497,050	BFS, SSSP
com-Orkut[14]	3,072,441	117,185,083	BFS, SSSP
Flickr[13]	820,878	9,837,214	PageRank, BFS, SSSP
as-Skitter[13]	1,696,415	22,190,596	PageRank
eu-2005[13]	862,664	19,235,140	PageRank
uk-2002[13]	18,520,486	298,113,762	PageRank, BFS, SSSP

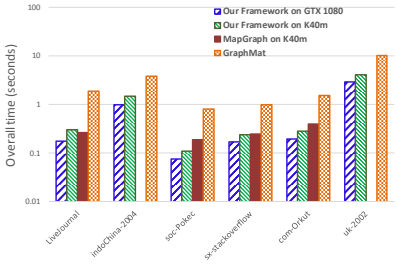
4.1 Experimental Setup

We conduct our GPU experiments, complied with CUDA Toolkit 8.0 release, on both NVIDIA Tesla K40m GPU and NVIDIA GeForce GTX 1080. K40m is equipped with 2,880 stream cores, 12 GB on-board memory, and memory bandwidth up to 288 GB/sec. While GTX 1080 has 8GB GDDR5X memory, 320 GB/sec of memory bandwidth. The CPU codes are compiled with the Intel ICPC 17.0.4 compiler, and executed on two Intel(R) Xeon(R) E5-2620 CPUs, each with 6 cores running at 2.40GHz (12 cores, 24 threads in total).

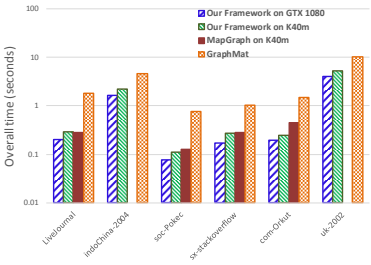
The experimental datasets contain both real-world and synthetic graphs, which are almost taken from the University of Florida sparse matrix collection [13] and Stanford Large Network Dataset Collection [14]. Since small-scale data processing is not deserve to be accelerated by GPU, we choose 11 large-scale graphs for evaluation. Table 1 shows the characteristics of these datasets, as well as the algorithms run on these graphs.



(a) PageRank



(b) BFS



(c) SSSP

Figure 3. Performance results (in log-scale) for PageRank, BFS and SSSP compared to MapGraph and GraphMat.

4.2 Performance Evaluation

As mentioned above, we implement graph algorithms on the proposed framework and select large-scale datasets for evaluation. We use GraphMat and MapGraph, representing advanced high performance frameworks based on CPU and GPU respectively, for performance comparisons. Figure 3 shows the time taken to run the graph algorithms after loading graphs into GPU or CPU memory (excluding time taken to allocate resources or read the graph from disk). Note that, we use the implementation with CSR-based matrix in GraphMat. And the time spent on GPU has been

counted in totally for MapGraph, in order for fair comparison. The *y*-axis on the figures represents runtime (in log-scale) except PageRank. As for PageRank, since each algorithm iteration takes similar time, we adopt time/iteration as y-axis. Lower bars indicate better performance. And GTX 1080 is not support to run MapGraph on our large-scale datasets. Any missing results on figures mean that the GPU ran out of memory.

As we can see from Figure 3(a), for PageRank, our framework running on Tesla K40m is within 97% of MapGraph performance, and 6.4X faster than GraphMat, on average. Concretely, the maximum speedup is 1.5X and 15.3X over MapGraph and GraphMat, respectively. While compared to GraphMat, our framework runs average 7.8X faster on GTX 1080, and yields the maximum speedups on com-youtube.

For Breadth First Search (BFS), when compared to MapGraph, our models is superior for all datasets except LiveJournal, and yields an average speedup of 1.2 and the maximum speedup of 1.7 (shown in Figure 3(b)). Meanwhile, our proposed framework runs 2.5-7.5X (4.8X on average) on Tesla K40m and 3.5-10.9X (7.1X on average) on GTX 1080 better than GraphMat.

Figure 3(c) shows the performance results in terms of Single Source Shortest Path (SSSP). Running on Tesla K40m, our framework runs about 20% faster than MapGraph, and the average speedup is 4.5X and the maximum speedup is 6.8X compared to GraphMat. While using GTX 1080, our framework achieves up to 10X (6.4X on average) faster than GraphMat.

We can easily conclude from those results that, our framework gain better performance on GTX 1080 rather than on Tesla K40m. And it should be noted that even using Tesla K40m which has larger on-chip memory, MapGraph still fails to run on indoChina-2004 or uk-2002, the two biggest datasets in our evaluation, because of an out of memory error. Since we do not need to allocate or manage considerable hardware resource, our framework are able to make full use of the available memory on different series of GPUs.

5. CONCLUSION

Graph analytics has been widely applied in many applications. Parallel graph computing frameworks have their respective strengths and weakness on CPU or GPU architectures [1,2,3,4,5,6,7,8,9,15]. In this papers, we propose a high performance graph programming framework on hybrid CPU-GPU platforms with good productivity. Our method is based on GraphMat, and leverages GPU to optimize it. The experimental results on large-scale graphs show that our implementation speeds up the computations significantly. Meanwhile, users or developers can improve our framework to complete more algorithms with little effort, which bridges the gap between performance and productivity. In future work, we plan to extend our framework to process graphs on multi-GPU compute clusters.

6. ACKNOWLEDGMENTS

The authors gratefully acknowledge supports from National Key Research and Development program under No.2016YFB1000400; National Nature Science Foundation of China under NSFC No. 61502509 and 61402504.

7. REFERENCES

[1] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: high performance graph analytics made productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214-1225. DOI: http://dx.doi.org/10.14778/2809974.2809983.

[2] Low Y, Gonzalez J E, Kyrola A, Graphlab: A new framework for parallel machine learning[J]. arXiv preprint arXiv:1408.2041, 2014.

[3] Malewicz G, Austern M H, Bik A J C, et al. Pregel: a system for large-scale graph processing[C]// ACM SIGMOD International Conference on Management of Data. ACM, 2010:135-146.

[4] Xin R S, Gonzalez J E, Franklin M J, et al. GraphX: a resilient distributed graph system on Spark[C]// International Workshop on Graph Data Management Experiences and Systems. ACM, 2013:2.

[5] Jianlong Zhong and Bingsheng He. 2014. Medusa: A Parallel Graph Processing System on Graphics Processors. SIGMOD Rec. 43, 2 (December 2014), 35-40. DOI=http://dx.doi.org/10.1145/2694413.2694421.

[6] Zhisong Fu, Michael Personick, and Bryan Thompson. 2014. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Proceedings of Workshop on GRAph Data management Experiences and Systems* (GRADES'14). ACM, New York, NY, USA, , Article 2 , 6 pages. DOI: https://doi.org/10.1145/2621934.2621936.

[7] Gonzalez J E, Low Y, Gu H, et al. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs[C]//OSDI. 2012, 12(1): 2.

[8] Combinatorial Blass v1.3. [Online]. Available: http://gauss.cs.ucsb.edu/aydin/CombBLAS/html/.

[9] Kang U, Tsourakakis C E, Faloutsos C. Pegasus: A peta-scale graph mining system implementation and observations[C]//Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on. IEEE, 2009: 229-238.

[10] Ashari, Arash, et al. "Fast sparse matrix-vector multiplication on GPUs for graph applications." Proceedings of the international conference for high performance computing, networking, storage and analysis. IEEE Press, 2014.

[11] Zhang, H., Chen, X., Xiao, N., Wang, L., Liu, F., Chen, W., & Chen, Z. (2016). Shielding STT-RAM Based Register Files on GPUs against Read Disturbance. ACM Journal on Emerging Technologies in Computing Systems (JETC), 13(2), 27.

[12] Liu Y, Schmidt B. LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs[C]//Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on. IEEE, 2015: 82-89.

[13] T. Davis. The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices.

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

[15] Chen, X., Li, P., Fang, J., Tang, T., Wang, Z., & Yang, C. (2017). Efficient and high‐quality sparse graph coloring on GPUs. Concurrency and Computation: Practice and Experience, 29(10).