GasCL: A Vertex-Centric Graph Model for GPUs

Shuai Che
{Shuai.Che}@amd.com
Advanced Micro Devices

***Abstract*—There are increasing research efforts of using GPUs for graph processing. Most prior work on accelerating GPGPU graph algorithms has been focused on algorithm and device-specific optimizations. There is little research on studying high-level programming models and associate run-time systems for graph processing on GPUs, which will be useful to solve diverse real-world problems flexibly.**

This paper presents a preliminary implementation of a graph framework, GasCL, supporting the well-known "think-like-a-vertex" programming model. The system is built on top of OpenCL and portable across diverse accelerators. We describe our design and use two applications as case studies. The initial performance result shows an average of 2.5× speedup on a GPU compared with a CPU.

I. Introduction

Graph processing is a key workload in big-data analytics. A National Research Council report [19] has identified it as one of the seven computational giants in the area of massive data analysis. Graph computations are increasingly used in social-network analysis, web analysis, business analytics, bioinformatics and infrastructure planning and so on. Recently, there are growing interests in accelerating graph applications on GPUs. Many graph and irregular graph applications achieved performance speedups on GPUs [2], [5]. However, they are mostly *ad-hoc* implementations for particular algorithms.

For real-world graph processing, in addition to function-level libraries (e.g., BFS, SSSP) for basic graph algorithms, developers need a high-level programming framework to develop graph applications flexibly for a variety of real-world use cases on GPUs. This also will help improve programmability and portability which are big challenges faced by GPGPU developers. In addition, the framework needs efficient and scalable mechanisms to schedule tasks on parallel computation resources within a machine node or across multiple nodes (e.g., such as MapReduce). Furthermore, graph applications present irregular random accesses, requiring SIMD-friendly data structures and layouts. In fact, many graph algorithms can be abstracted in a graph-centric manner, where parallel computations are performed independently on individual graph structures (e.g., vertices) and inter-vertex communications are achieved through message passing. Such frameworks are based on the concept of a "think-like-a-vertex" model.

This paper presents a first-step implementation of an efficient graph processing framework, GasCL (Gather-Apply-Scatter with OpenCL), targeting GPUs. Our approach, similar to those in GraphLab [13], Giraph [22] and Pregel [14], is flexible to support a wide variety of graph algorithms and extensible to diverse architectures or multi-node clusters. In GasCL, graph applications are described in vertex-centric computation kernels, which are launched over multiple *supersteps* with a series of gather-apply-scatter phases. When necessary, data exchanges are performed by passing messages among vertices. Our API is currently built on top of the OpenCL C++ static-kernel language extension [20]. We prototype a runtime system mapping GasCL graph applications in parallel on GPU SIMD engines.

978-1-4799-6233-4/14/$31.00

This work makes the following contributions:

- We present a preliminary graph framework for developing graph applications on GPUs. GasCL provides an API to program graph applications in a Gather-Apply-Scatter (GAS) style.
- We describe our system design and execution model for efficient parallel work distribution and message passing.
- We study representative graph algorithms, PageRank and Single-Source Shortest Path, and show how to program them with GasCL. We also demonstrate preliminary performance results on an AMD discrete GPU.

II. Background of Graph Processing

Parallel graph computations can be expressed in multiple abstractions. In one research direction, graphs are stored in sparse matrices, and applications are constructed with sparse-matrix and vector building blocks. For instance, Combinatorial BLAS [1] is graph library offering a powerful set of linear algebra primitives for graph analytics. Graph BLAS [16] is a recent effort defining standard linear-algebra building blocks for graph processing. This area of research on graph algorithms in linear algebra is documented in the work by Kepner and Gilbert [12].

In another research direction, large-scale graph frameworks are developed to solve web analysis, business analytics, social network problems and so on. These systems are designed, with a different concept. Their models are somewhat similar to MapReduce [6] but the API and runtime is designed and optimized for graph processing. Examples of development in this domain include Google's Pregel [14], Giraph [22], GraphLab [13] and Grappa [9]. In these frameworks, programmers are asked to write applications in a gather-apply-scatter programming model. The system schedules parallel work of individual each vertices or edges to multiple processing elements or machines. GasCL adopts a similar model but targets the GPU platform.

Prior work studied and accelerated basic graph algorithms [2], [3], [5], [8], [10], [17], [24] on GPUs. Most of these works conduct algorithmic and device-specific optimizations to achieve high performance. Our goal is to develop a system to program and run diverse graph algorithms on GPUs, not restricted to any specific algorithm implementation. Other prior GPU research studies "think-like-a-vertex" models [7], [25], however they use different implementations (e.g., scan, internal expand) or are based on CUDA.

III. GasCL

Our GasCL framework is based on the Gather-Apply-Scatter (GAS) model. It consists of an API, graph runtime system and utility library. GasCL currently exploits the parallel resources of a single GPU node. The underlying GasCL implementations take into account of GPU-specific architecture features (e.g., SIMD execution and memory coalescing) and use a variety of optimizations for high performance. GasCL supports both "directed" and "undirected" graphs. Vertices, edges and messages are all abstracted in well-defined objects for developers to manipulate.

A. Gather-Apply-Scatter

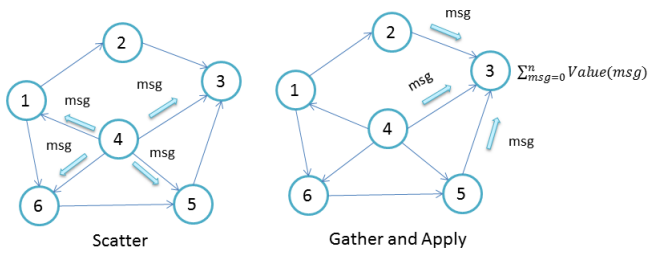

Fig. 1. Vertex-to-vertex communications are achieved by sending and receiving messages

The Gather-Apply-Scatter model is based on the observation that many graph algorithms present a common pattern that processing of vertices and edges can be localized in neighbouring vertices. A parallel program is written from the perspective of an individual graph element (e.g., vertex), and independent units of work are distributed in parallel to processing elements. The program takes multiple supersteps to converge with possible data exchanges among graph elements. To use such a model, programmers will need to adapt a graph application to the GAS programming style. The benefit is that the underlying runtime system is able to naturally exploit the parallelism described for vertices or edges in the kernels.

Figure 1 shows an example that certain computations of individual vertices can be done in parallel and inter-vertex communications are achieved through sending and receiving messages. For instance, in the first phase, vertex 4 sends along its edges four messages to its neighbors (1, 3, 5 and 6). All the other vertices are doing the same operation in parallel (not shown in the figure). The messages are stored in system-managed buffers for the next stage to consume. In the second phase, vertex 3 (similarly for all other vertices) collects all the messages it receives from its incoming edges, performs a sum operation, and updates its vertex value.

GasCL support three high-level abstractions and can be summarized into three major phases of compute kernels applied on all the vertices:

- **Gather**: The *gather* phase reads data and messages from the neighborhood of a vertex.
- **Apply**: The *apply* phase performs some computation and updates the value of a vertex.
- **Scatter**: The *scatter* phase sends data and messages to the neighborhood of a vertex.

The computation of individual vertices is conducted in *apply*. But in certain cases, the *gather* and *apply* phases can be combined; *apply* operates on the received messages from the neighborhood. Our first-step implementation adopts a vertex-centric model. Other proposed approaches are with a edge-centric view [21]. Extension to other models is left for future work.

Two important operations in the GasCL API are:

- *SendMsg()*: this function is used by a vertex to send a message to a destination vertex.
- *CombineMsg_Op()*: this function is used by a vertex to combine received messages. The operator (i.e., *Op*) can be $+$, $\times$, max, min, etc.

For *SendMsg()*, the destination vertex does not have to be in the frontier which is one hop immediately adjacent to the source, and can be an arbitrary vertex in the graph. For *CombineMsg_Op()*, it is useful when the operation of combining is associative and commutative; that is, the messages can be reduced in any order. This is similar to Combiner in MapReduce and Hadoop. In addition, GasCL provides users the interface to access and process the messages of a vertex, to perform any operations.

B. Execution Model and Runtime

This section briefly discusses the GasCL execution model and runtime. Figure 2(a) shows the entire GasCL software stack. GasCL provides an API (described in Section III-A) for programmers to write application-specific kernels. The API provides programmers capabilities to manipulate various basic graph elements, such as vertices, edges, edge lists, messages, etc. for a graph. They are encapsulated in well-defined classes or structs on both the host and the device with the actual implementations handled by the system. Graphs are constructed by the GasCL system in different formats. The choice of a particular format for an application is configurable by the user through the GasCL provided interface.

The GasCL runtime system maps high-level GasCL programming constructs and API calls to the underlying graph elements and schedules parallel computations on the GPU. For instance, the task of a vertex can be mapped to an OpenCL work-item or a thread. Alternatively, multiple vertices can map to a single thread depending on the implementation. The graph utility library includes a variety of routines to

parse graphs in different formats with different preprocessing options. GasCL loads a graph input file and constructs its data structure representation in the host memory. The runtime also creates GPU-side buffers, and map/copy host-side graph objects to the device (in the case of HSA-enabled devices [11], pointers can directly be passed to GPU kernels). Each *gather*, *apply* and *scatter* step is associated with a GPU kernel call. Inside the kernel, vertex instances will be initiated, with each OpenCL workitem responsible for one vertex. The actual user-defined *gather*, *apply* and *scatter* functions are then applied on each vertex.

Figure 2(b) shows a typical execution flow of a GasCL application. First, the program initializes the OpenCL platform and context, sets up OpenCL command queues, compiles GPU kernel program, and conducts other platform-specific initializations. After GasCL builds the graph, the program performs application-specific initializations specified by the programmer. The application then enters the main computation loop with condition check. Each iteration of the loop is deemed to be a *superstep*. In our current implementation, all the work-groups for a phase complete and synchronize before proceeding to the next stage. However, this is currently restricted by synchronization primitives provided by OpenCL. It is possible that a future optimization may relax global synchronization in a superstep or across supersteps by tracking computation dependences among vertices. We leave this for future work. The condition check function, defined by the user, returns *true* to resume the loop and *false* to terminate the loop. For instance, in a graph traversal application, it can be programmed such that when all the vertices are visited, it returns *false*. GasCL maintains a mask vector to facilitate this conditional check (see Section V).

C. Graph Inputs and Data Structures

GasCL currently supports data layouts similar to *compressed sparse row (CSR)* and *coordinate list (COO)* to represent a graph in memory. We are working on extending the framework to support diverse data structures for different usages, since different algorithms and traversal patterns may prefer different data layouts. For graph parsing, GasCL currently implements the widely-used Matrix Market [15] (MM) format. The next step is to support DIMACS Challenge [4] and METIS [18] formats. GasCL contains utility routines to parse graph inputs stored in these formats. For the experiments in this study, we choose graphs from the University of Florida Sparse Matrix Collection [23].

IV. Graph Objects and Operations

In this section, we will discuss how we implement graph structures in the GasCL framework.

A. Graph Elements

GasCL provides abstractions and operations for basic graph elements (e.g. vertices and edges). For instance, for vertices, we provide member functions to send a message to a destination vertex (e.g., *sendMsg(int dstID, VARTYPE msg_val)*).

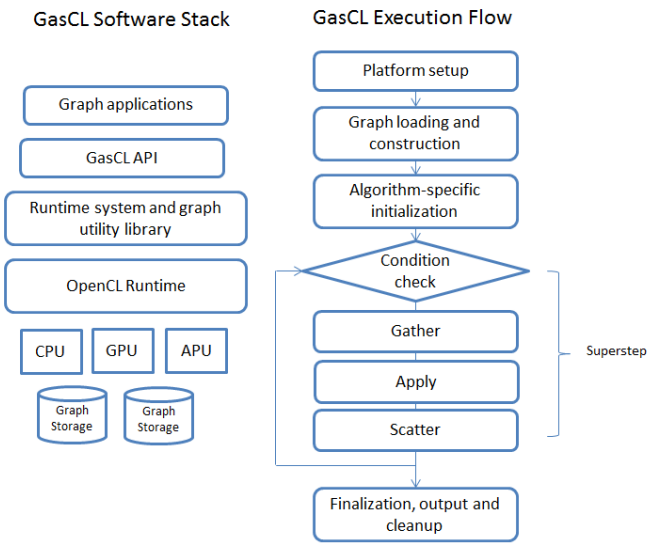

Fig. 2. (a) GasCL software stack and (b) GasCL execution flow

Also, combining messages with a + operator for a vertex can be done through the member function *combine_msg_sum()*. Also, each vertex is associated with some meta data (e.g., vertex id, edge offset/count, etc.) useful for computation. For some of them, we provide member functions to fetch and update these variables. For instance, *edge_count()* is used to obtain the number of neighbors for a vertex, and *get_value()* is used to get the node value of a vertex.

Data and state for a class or struct are usually stored in adjacent memory locations. However, this might not be ideal for SIMD computation when multiple threads access the same variable of different vertices. Ideally GPU threads (in a wavefront) should access contiguous data elements for the memory-coalescing purpose, thus memory accesses can be coalesced into fewer memory transactions. Our framework, where possible, organizes data in structure-of-arrays (SoA) instead of array-of-structs (AoS). For instance, the edge counts of different vertices can be stored in a single array. Also, we maintain vertex value array, edge-list array, message array, etc. which can be globally shared by different vertices. Accesses to these data structures are through member functions of different graph elements.

B. Message

In GasCL, a message can be a built-in or user-defined data type (e.g., float, integer, and others). For GasCL, we allocate a single linear array storing all the messages. Different segments of messages are grouped by vertices. Therefore, there could be two options: 1) the messages are stored together for the same source vertex or 2) the messages are stored together for the same destination vertex. In the first case, the implementation can associate a message with the outgoing edge from the source vertex (this solution can be easily integrated in a CSR-type data layout). However, one possible drawback is that an additional step is needed to collect all the messages from

different memory regions, and group and store them for the same destination vertices. In GasCL, the messages sent to the same destination vertex are stored consecutively, and once the message is sent from the source, it is directly written to the correct position for the destination vertex. This solution also simplifies the implementation of message combining. To achieve this, we use a similar technique discussed in the work [25]. We implement *reversed edge index* which gives the array offset to store the message. While loading the graph, GasCL constructs a reverse graph by swapping the head and tail of each edge. We assign a reverse ID for each edge in the original graph, whereby the reverse ID value of each edge equals the index of its reverse edge in the adjacency array. Reverse IDs can be generated by preprocessing graph input files by sorting the edge array based on the tail vertex ID, and keeping track of the array indices during sorting. The preprocessed graph can be stored for other uses.

V. Mask and Convergence

GasCL provides programmers a mask vector which is useful in different scenarios. Each position of the mask can be used to keep track of the status of a vertex (i.e., true or false). In GasCL, *set_mask(int id)*, *unset_mask(int id)* and *get_mask(int id)* are functions to set, clear and fetch the values from the mask vector respectively. When programming graph applications, these primitives are useful in determining and specifying the set of active vertices for graph traversal and computation.

As discussed in Section III-B, a typical pattern in graph applications is that the main loop launches multiple iterations of GPU kernels, each processing a subset of a graph. The application sometimes needs to determine when to converge because of the data-dependent feature. In GasCL, this is achieved through a user-define condition check. The return value (true/false) of this function is used by the host to decide whether to resume/terminate a loop. Programmers can manipulate mask structure through the GasCL interface to obtain the overall information of a graph.

VI. Applications

A. PageRank

PageRank (PRK) is an algorithm to calculate probability distributions representing the likelihood that a person randomly clicking on links arrives at any particular page. In PageRank, each vertex assigned to a task sends along its outgoing edges the current PageRank divided by the number of outgoing edges in each step of the main computation loop [14]. Each vertex then sums the values arriving at the corresponding vertex, and calculates a new PageRank value. The algorithm terminates when convergence is determined by an aggregator or after running a user-specified number of iterations (supersteps). Using GasCL, the entire algorithm can be expressed in two phases: the *scatter* phase where each vertex transfers some amount of PageRank value to its neighbors, and the *gather* and *apply* phase where each vertex collects the messages from all the incoming edges and updates the vertex with a new PageRank value. A new PageRank value is calculated with the combine_msg_sum() operation. The following pseudocode shows an example of PageRank with GasCL.

```
//Pseudo-code for PageRank (kernel side)
//Each vertex sends value to neighbors
class scatter
{
 public:
  void operator() (D_Vertex vertex)
  {
    int start = vertex.start_edge();
    int edge_cnt = vertex.edge_count();
    int end   = start + edge_cnt;
    //navigate through the neighbor list
    //send message to each my neighbor
    for(int i = start; i < end; i++){
        float msg_val = vertex.get_value()/edge_cnt;
        vertex.sendMsg(i, msg_val);
    }
  }
};

//Each vertex gathers received PageRank values
//and updates its own value
class gather
{
 public:
  void operator() (D_Vertex vertex)
  {
    //combine the messages with a sum operation
    float msg_sum = vertex.combine_msg_sum();
    msg_sum *= 0.85;
    vertex.set_value(0.15 + msg_sum);
  }
};
```

B. Single-source shortest path

Single-source shortest path (SSSP) is an important subroutine in various graph algorithms. Given a user-specified source vertex in the graph, the algorithm searches the path with lowest cost (i.e., the shortest path) between the source vertex and all the other vertices in the graph. SSSP keeps track of a distance array, saving the shortest distances of all the vertices evaluated so far. For each neighbor of a new visited vertex, if the calculated distance via passing through the vertex is smaller than the old distance, a new value of distance will be updated.

The following example shows the pseudo-code for single-source shortest path in GasCL. The algorithm expands vertex frontiers continuously to update shortest distances. The vertices will become active when reaching a new level of frontier or shorter distances for the already visited ones are found. Users can use mask API functions to implement the algorithm as shown in the code. The message sent from a vertex to a particular neighbor is its current distance of the vertex plus the weight of the outgoing edge to the neighbor. In addition, in contrast to PageRank, the combiner function applies a *min* operation instead of a + operation.

VII. Experiment Setup

The experiment results are measured on real hardware using an AMD Radeon HD 7950 (Tahiti) discrete GPU. The AMD Radeon HD 7950 features 28 GCN CUs with 1792 processing

elements running at 800 MHz with 3 GB of device memory. We compare the GPU results with those obtained from four CPU cores on an AMD A8-5500 accelerated processing unit (APU) with a 1.4-GHz clock rate and 2 MB L2 cache. We use AMD APP SDK 2.8 with OpenCL 2.1 support. In addition, this study is restricted to cases when the working sets of applications do not exceed the capacity of the GPU device memory.

```
//Pseudo-code for Single-Source Shortest Path (kernel side)
class scatter
{
 public:
  void operator() (D_Vertex     vertex,
                   D_Edge_Array edge_array,
                   D_Mask       mask)
  {
    int start     = vertex.start_edge();
    int edge_cnt  = vertex.edge_count();
    int end       = start + edge_cnt;
    int my_id     = vertex.get_vertex_id();

    //if myself is active,
    if(mask.get_mask(my_id)){
        mask.unset_mask(my_vid);
        //send the sum of my current distance
        //and the weight of the outgoing edge
        //to my neighbor
        for(int i = start; i < end; i++){
          float dst_val = vertex.get_value() \
                        + edge_array.edge_weight(i);
          vertex.sendMsg(i, dst_val);
          int neighbor_id = edge_array.edge_id(i);
          //set my neighbor active
          mask.set_mask(neighbor_id);
          }
        }
    }
};

class gather
{
 public:
    void operator() (D_Vertex vertex,
                     D_Mask mask)
    {
        int my_id = vertex.get_vertex_id();
        //if myself is active
        if(mask.get_mask(my_id)){
           //combine the messages with a min operation
           float min_sum = vertex.combine_msg_min();
           vertex.set_value(min_sum);
        }
    }
};
```

VIII. Results

In this section, we report some preliminary results for two applications we discussed (PageRank and Single-Source Shortest Path). We calculate the performance speedup by measuring the execution time on the GPU and compare it to the CPU. Figure 3 reports the performance speedups of two applications with sample graph inputs. The measurement includes PCI-E overhead excluding graph parsing and preprocessing. The arithmetic mean of speedups across all the program-input pairs is approximately 2.5×. Application performance is also input-dependent. For instance, for PageRank, the performance speedup ranges from 0.8× (*delaunay*) to 5.5× (*flicker*). For Single-Source Shortest Path, the performance speedup ranges from 0.3× (*G2-circuit*) to 4.7× (*mesh-deform*).

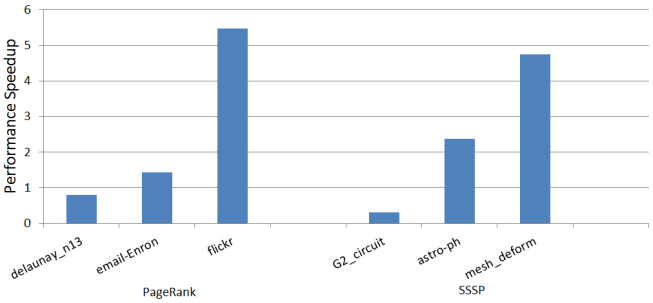

Fig. 3. Performance speedups of graph applications over different graph inputs (GPU vs. CPU). The calculation takes into account of the PCI-E overhead, but excludes parsing and preprocessing graphs.

Most of the PCI-E overhead is due to copying the graph data structures from the CPU to the GPU. Its portion also varies across different program-input pairs. For instance, for PageRank, PCI-E overhead ranges from 5% for *email-Enron* to 40% for *delaunay* of the main computation part (excluding file I/O and preprocessing).

IX. Conclusion and Future Work

This paper presents a preliminary implementation of a vertex-centric graph model for the GPU. GasCL provides an API which allows programmers to develop diverse graph applications with the gather-apply-scatter model. It also dispatches parallel computation kernels efficiently on the GPU. Our system is currently built on top of OpenCL. Our first-step result shows an average of 2.5× performance speedup for PageRank and Single-Source Shortest Path.

Future research will evaluate the framework with more graph applications and inputs, diverse data layouts (e.g., ELLPACK, diagonal) and perform device-specific code optimizations. We will also compare our implementations to hand-tuned GPU codes for these graph algorithms on different hardware. Future work will extend GasCL to multiple GPU nodes and leverage unique advantages of both the CPU and the GPU for graph processing on an AMD Accerated Processing Unit (APU). It is also an interesting research to study productivity improvement such as relative speedup vs. relative effort.

Acknowledgements

We would like to thank the reviewers for their constructive comments and suggestions.

References

[1] A. Buluc and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *International Journal o High Performance Computing Applications*, 25(4), November 2011.

[2] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization*, Nov 2012.

[3] M. Burtscher and K. Pingali. An efficient cuda implementation of the tree-based Barnes Hut n-body algorithm. In *GPU Computing Gems Emerald Edition*, pages 75–92. Morgan Kaufmann, 2011.

[4] The DIMACS Implementation Challenge. Web resource. http://dimacs.rutgers.edu/Challenges/.

[5] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron. Pannotia: Understanding irregular GPGPU graph algorithms. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Sept 2013.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of ACM*, 51(1), January 2008.

[7] E. Elsen and V. Vaidyanathan. Vertexapi2 - a vertex-program api for large graph computations on the gpu. http://www.royal-caliber.com/vertexapi2.pdf.

[8] GALOIS. Resource. http://iss.ices.utexas.edu/?p=projects/galois.

[9] Grappa: Large-Scale Graph Processing with Commodity Processors. Web resources. http://www.cs.washington.edu/node/4217/.

[10] P. Harish and P. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of 2007 International Conference on High Performance Computing*, Dec 2007.

[11] Heterogeneous System Architecture (HSA). Web resource. http://hsafoundation.com/.

[12] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, January 2011.

[13] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010.

[14] G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, June 2010.

[15] Matrix Market Format. Web resouce. http://math.nist.gov/MatrixMarket/formats.html.

[16] T. Mattson, D. A. Bader, J. W. Berry, A. Bulu, J. Dongarra, C. Faloutsos, J. Feo, J. R. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. E. Leiserson, A. Lumsdaine, D. A. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo. Standards for graph algorithm primitives. In *Proceedings of IEEE High Performance Extreme Computing Conference*, pages 1–2, 2013.

[17] D. G. Merrill, M. Garland, and A. S. Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb 2012.

[18] METIS File Format. Web resource. http://people.sc.fsu.edu/~jburkardt/data/metis_graph/metis_graph.html.

[19] Committee on the Analysis of Massive Data; Committee on Applied, Theoretical Statistics; Board on Mathematical Sciences, Their Applications; Division on Engineering, and Physical Sciences; National Research Council. *Frontiers in Massive Data Analysis*. The National Academies Press, 2013.

[20] OpenCL Static C++ Kernel Language Extension. Web resource. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/CPP_kernel_language.pdf.

[21] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, Nov 2013.

[22] The Apache Giraph. Web resource. https://giraph.apache.org/.

[23] The University of Florida Sparse Matrix Collection. Web resource. http://www.cise.ufl.edu/research/sparse/matrices/.

[24] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics*, July 2009.

[25] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 99:1, 2013.