Medusa: A Parallel Graph Processing System on Graphics Processors

Jianlong Zhong
Nanyang Technological University
jzhong2@ntu.edu.sg

Bingsheng He
Nanyang Technological University
bshe@ntu.edu.sg

ABSTRACT

Medusa is a parallel graph processing system on graphics processors (GPUs). The core design of Medusa is to enable developers to leverage the massive parallelism and other hardware features of GPUs by writing sequential C/C++ code for a small set of APIs. This simplifies the implementation of parallel graph processing on the GPU. The runtime system of Medusa automatically executes the user-defined APIs in parallel on the GPU, with a series of optimizations based on the architecture features of GPUs and characteristics of graph applications. In this paper, we present an overview of the Medusa system and a case study of adopting Medusa to a research project on social network simulations. With Medusa, users without GPU programming experience can quickly implement their graph operations on the GPU, which accelerates the discovery and findings of domain-specific applications.

1. INTRODUCTION

GPGPU (General-Purpose Computation on Graphics Processing Units) is gaining increasing popularity in performance acceleration for many applications, including graph processing [8, 10]. A Modern GPU can have over an order of magnitude higher memory bandwidth and computation power than a multi-core CPU. With intensive and application-specific optimizations, GPU-based graph algorithms have shown significant performance improvement over CPU-based implementations. For example, the GPU accelerated breadth first search (BFS) algorithm is up tp 14 times faster than multi-core implementation [10]. However, despite the recent improvements on GPGPU programmability, writing a correct and efficient GPU program is challenging in general and even more difficult for the highly irregular graph applications.

To address the above-mentioned issues and simplify programming graph processing algorithms on the GPU, we propose the Medusa parallel graph processing programming framework. Like existing programming frameworks such as Hadoop [17] and Mars [9], Medusa provides a small set of APIs for developers to implement their applications by writing sequential (C/C++) code. Different from Hadoop and Mars, which adopts the MapReduce programming model [3], Medusa adopts our novel "Edge-Message-Vertex" (EMV) graph programming model for fine-grained processing on edges, messages and vertices. Medusa embraces an efficient message passing based runtime. It automatically executes user-defined APIs in parallel within one GPU and across multiple GPUs, and hides the complexity of GPU programming from developers. Thus, developers can write the same APIs, which automatically run on one or multiple GPUs. To maximally leverage the power of GPUs, Medusa embraces a series of optimizations based on the architecture features of GPUs and characteristics of graph applications.

As a case study, we adopt Medusa to implement GPU accelerated simulation of information propagation over social networks. Simulation of information propagation is computationally intensive and fortunately highly parallelizable, which makes it viable for GPU acceleration. By the case study, we demonstrate that Medusa both improves the coding productivity and brings significant performance speedups.

This paper gives an overview of Medusa based on work reported in [20, 18, 11, 12, 13]. We first present the design of the main components of Medusa's system architecture and evaluation results. We then present the case study developed based on Medusa. Finally, we conclude this paper and present the future work.

2. RELATED WORK

Parallel graph processing. It has been observed that many common graph algorithms can be formulated using a form of the bulk synchronous parallel (BSP) model [14] (we call it *GBSP*). Under the GBSP model, local computations on each vertex are performed in parallel iteratively. At the end of each iteration, vertices exchange data by message passing, followed by a global synchronization. More recently, the asynchronous model has been applied to improve the convergence speed of some graph applications [14]. Due to the synchronous nature of the GPU programming model, we only support the synchronous GBSP model on GPU at the moment.

GPGPU. We use NVIDIA CUDA's terminology. The GPU consists of multiple of streaming multi-processors (SM), inside which there is an array of scalar cores. A CUDA program, which is called a *k*ernel, usually consists of thousands of threads. The massive number of threads of a kernel runs on all the SMs of a GPU in parallel. Each 32 of the threads are grouped into a warp and execute synchronously. Divergence inside a warp introduces severe performance penalty since different paths are executed serially. The GPU requires *coalesced access* to its memory to ensure high utilization of its memory bandwidth. To achieve coalesced access, threads in the same warp must access the same memory segment each time. Another important feature for performance optimization on the GPU is the *shared memory*, which is a small piece of scratch pad memory on the SM. Shared memory has much lower latency compared with the GPU memory. Utilizing shared memory can greatly improve the data access performance of CUDA programs.

3. SYSTEM OVERVIEW

In this section, we give an overview on how users implement their graph processing algorithms based on Medusa, and outline the key modules of Medusa.

3.1 Programming with Medusa

We propose the EMV model as the programming model of Medusa. EMV is similar to the GBSP model and specifically tailored for parallel graph processing on the GPU. To facilitate efficient and fine-grained graph processing on the GPU, EMV decomposes each iterative graph operation in GBSP into three sub-iterations, i.e., processing on edges, messages and vertices. Medusa hides the GPU programming details from users by offering two kinds of APIs, EMV APIs and system-provided APIs, as shown in Tables 1 and 2, respectively. Through those APIs, Medusa enables programmability and efficiency for parallel graph processing on the GPU.

Each EMV API is either for executing user-defined computation on vertices ($VERTEX$), edges ($ELIST$, $EDGE$) or messages ($MESSAGE$, $MLIST$). The vertex and edge APIs can also send messages to neighboring vertices. The idea of providing these APIs is mainly for efficiency. It decouples the single vertex API of previous GBSP-based systems into separate APIs which target individual vertices, edges or messages. Each GPU thread executes one instance of the user-defined EMV API. The fine-grained data parallelism exposed by the EMV model can better exploit the massive parallelism of the GPU. In addition, a *Combiner* API is provided to aggregate results of *EDGE* and *MESSAGE* using an associative operator. This enhances the execution performance since segmented-scan has very efficient and load balanced implementation on the GPU [15].

```
Device code APIs:                                  Iteration definition:
/* EDGE API */                                     void SSSP() {
struct SendDistance{                                 /* Initiate message buffer to UINT_MAX */
__device__ void operator() (Edge e){                 InitMessageBuffer(UINT_MAX);
  int head_id = e.get_head_id();                     /* Invoke the EDGE API */
  Vertex head(head_id);                              EMV<EDGE>::Run(SendDistance);
  if(head.get_updated())                             /* Invoke the message combiner */
  {                                                  Combiner();
    unsigned int msg = v.get_distance() +            /* Invoke the VERTEX API */
                       e.get_length();               EMV<VERTEX>::Run(UpdateDistance);
    e.sendMsg(msg);                                }
  }                                                Configurations and API execution:
}                                                  int main(int argc, char **argv) {
/* VERTEX API */                                     /* Load the input graph. */
struct UpdateDistance{                               Graph my_graph;
__device__ void operator() (Vertex v){               conf.combinerOpType = MEDUSA_MIN;
  unsigned int min_msg = v.combined_msg();           conf.combinerDataType = MEDUSA_UINT;
  if(min_msg < v.get_distance())                     conf.gpuCount = 1;
  {                                                  conf.continueIteration = false;
    v.set_distance(min_msg);                         /*Setup device data structure.*/
    v.set_updated(true);                             Init_Device_DS(my_graph);
    Medusa_Continue();                               Medusa::Run(SSSP);
  }                                                  /* Retrieve results to my_graph. */
  else                                               Dump_Result(my_graph);
    v.set_updated(false);                            ......
}                                                    return 0;
                                                   }
```

Figure 1: User-defined functions in SSSP implemented with Medusa.

A small set of system provided APIs is designed to hide the GPU-specific programming details. Particularly, Medusa provides $EMV < type >:: Run()$ to invoke the device code APIs, which automatically sets up the thread block configurations and calls the corresponding user-defined functions. Medusa allows developers to define an *iteration* which executes a sequence of $EMV < type >:: Run()$ calls in one host function (invoked by $Medusa :: Run()$). The iteration is performed iteratively until predefined conditions are satisfied. Medusa offers a set of configuration parameters and utility functions for iteration control.

To demonstrate the usage of Medusa, we show an example of the SSSP (Single Source Shortest Path) implementation with Medusa, as shown in Figure 1. The function *SSSP()* consists of three user-defined EMV API function calls, which is the three main steps of the algorithm. First, we use an *EDGE* type API (*SendDistance*) to send tentative new distance values to neighbors of updated vertices. Second, we use a message *Combiner* to calculate the minimums of received distances of each vertex. Third, we

Table 1: EMV APIs

API Type	*Parameters*	*Variant*	*Description*
ELIST	Vertex v, Edge-list el	Collective	Apply to edge-list el of each vertex v
EDGE	Edge e	Individual	Apply to each edge e
MLIST	Vertex v, Message-list ml	Collective	Apply to message-list ml of each vertex v
MESSAGE	Message m	Individual	Apply to each message m
VERTEX	Vertex v	Individual	Apply to each vertex v
Combiner	Associative operation o	Collective	Apply an associative operation to all edge-lists or message-lists

Table 2: System provided APIs and parameters in Medusa

API/Parameter	*Description*
AddEdge (void* e), *AddVertex*(void* v)	Add an edge or a vertex into the graph
InitMessageBuffer(void* m)	Initiate the message buffer
maxIteration	The maximum iterations that Medusa executes ($2^{31} - 1$ by default)
halt	A flag indicating whether Medusa stops the iteration
Medusa :: *Run*(Func f)	Execute f iteratively according to the iteration control
EMV<*type*>:: *Run*(Func f')	Execute EMV API f' with *type* on the GPU

invoke a *VERTEX* type API (*UpdateDistance*) to update the distances of vertices which receive new distances lower than their current distances. In the main function, we load the input graph and configure the execution parameters such as the *Combiner* data type and operation type, the number of GPUs to use and the default iteration termination behavior. *Medusa::Run(SSSP)* invokes the *SSSP* function.

3.2 System Internals

We proposed various optimization techniques for Medusa internals to ensure the high performance. For example, we optimize the graph data layout on the GPU memory, as well as layout of user-defined data structures for the efficiency of GPU memory access. Specifically, Medusa mainly consists of the following key modules.

Graph Storage. The storage module of Medusa allows developers to load the graph data through adding vertices and edges with the system provided APIs *AddEdge* and *AddVertex*. During loading of the graph data, data are stored in our novel graph layout optimized for GPU access [20]. Compared with the classic adjacency list layout, the optimized layout facilitates coalesced access during execution of graph algorithms to ensure high memory bandwidth utilization. After the graph is loaded into main memory, it is automatically transfered to the GPU memory.

Medusa code generation tool chain. Users build Medusa-based programs with standard C/C++. The Medusa code generation tool chain translates the user code into CUDA code. Firstly, Medusa translates user-defined EMV APIs into kernel codes, and generates corresponding kernel invocation codes. Secondly, Medusa translates user-defined data structures, such as data structures for vertex and edge, into GPU data structures and corresponding data transfer codes. Thirdly, Medusa inserts segmented-scan code for the *Combiner* APIs.

Medusa runtime. This module provides runtime support for user applications and manages GPU resource and kernel executions. In particular, this module has three main functionalities. First, it supports the message passing interface. We develop a novel graph aware message passing mechanism for efficiency [20]. Second, Medusa runtime enables transparent execution of user applications on the multi-GPU environment. We further propose techniques such as overlapping kernel execution with data transfer and multi-hop replication to increase the scalability of multi-GPU execution. Third, Medusa runtime supports concurrent execution of multiple Medusa tasks from different users. In the multi-user environment, Medusa coordinates the GPU memory allocation and kernel execution commands to prevent resource allocation deadlocks and exploit opportunities for performance improvement. Meudsa runtime also exploits the complementary resource requirements among the kernels to improve the throughput of concurrent kernel executions. More details about our concurrent kernel scheduling mechanism can be found in our paper [19].

4. RESULTS

We evaluate Medusa with both synthetic graphs generated by GTGraph [7] and publicly available real world graphs [16, 1]. The details of the graph data, including the numbers of vertices and edges, and standard deviations of the vertex degree, are shown in Table 3. Our workload includes a set of common graph computation and visualization primitives. The graph computation primitives include PageRank, BFS, maximal bipartite matching (MBM), ans SSSP. Our experimental platform is a work station with four NVIDIA Tesla C2050 GPUs and two six-core Intel Xeon E5645 CPUs at 2.4 GHz. We developed a set of visualization primitives

Table 3: Details of graphs used in the experiments

Graph	Vertices (10^6)	Edges (10^6)	σ
RMAT	1.0	16.0	32.9
Random (Rand)	1.0	16.0	4.0
BIP	4.0	16.0	5.1
WikiTalk (Wiki)	2.4	5.0	99.9
RoadNet-CA (Road)	2.0	5.5	1.0
kkt_power (KKT)	2.1	13.0	7.5
coPapersCiteseer (Cite)	0.4	32.1	101.3
hugebubbles-00020 (Huge)	21.2	63.6	0.03

Table 4: Coding complexity of Medusa implementation and hand-tuned implementations.

	Baseline	Warp-centric	Medusa (N/Q)
GPU code lines (BFS)	56	76	9/7
GPU code lines (SSSP)	59	N.A.	13/11
GPU memory management	Yes	Yes	No
Kernel configuration	Yes	Yes	No
Parallel programming	Thread	Thread+Warp	No

such as graph layout and drilling up/down. We implement the force direct layout algorithm [4]. The drilling up/down operation is implemented using BFS. The visualization experiments are conducted on a workstation with one NVIDIA Quadro 5000 GPU and one Intel Xeon W3565 processor with 4 GB memory.

Comparison with Hand-Tuned Implementations. We compare the traversed edges per second (TEPS) of Medusa-based BFS with the basic implementation from Harish et al [8] and the warp-centric method from Hong et al [10]. Compared to the basic implementation, Medusa performs much better on all graphs except KKT. The average speedup of Medusa over the basic implementation is 3.4. Medusa has comparable performance with the warp-centric method, while the latter has much more complicated implementation. We also compare Medusa-based SSSP with Harish et al's implementation. Medusa achieves comparable performance with Harish et al's implementation except on Road and Huge. This is because Road and Huge have large diameters, which lead to large numbers of iterations of Medusa. Table 4 shows the coding complexity of the three implementations. Medusa significantly reduces the developing effort by hiding the GPU programming details and reducing the lines of code.

Comparison with CPU-based Implementations. We compare Medusa with MTGL [2] based multi-core implementations running on 12 cores. Similar to Medusa, MTGL offers a set of data structures and APIs for building graph algorithms. MTGL is optimized to leverage shared memory multithreaded machines. For all the computation primitives, Medusa is significantly faster than MTGL on most graphs and delivers a performance speedup of 1.0-19.6 with an average of 5.5.

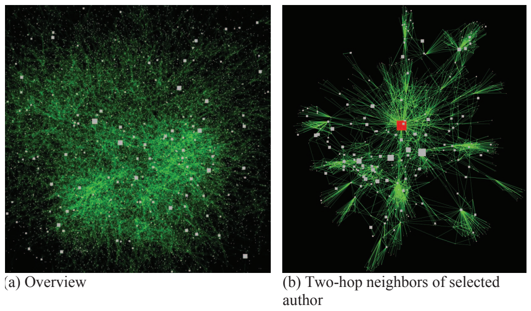

Figure 2: Visualization results on DBLP co-authorship graph.

We also evaluate the performance of our graph visualization primitives. The input is the co-author graph extracted from DBLP (http://dblp.uni-trier.de/xml/). Figure 2(a) shows the results of our layout primitive on the DBLP graph. On the Quadro 5000 GPU, Medusa takes only 120 seconds to compute the layout, while the 4-thread CPU implementation on the Intel Quad-core CPU takes over 1000 seconds. Figure 2(b) shows the two-hop neighbors of a selected author Jiawei Han obtained by a drill-down operation. Medusa greatly improves the responsiveness of graph visualization tasks.

Scalability. The memory sizes of our input graphs ranges from as small as less than 100 MB (Wiki) to as large as 1 GB (Huge). Our experiments show that Medusa fully utilizes the GPU for all the graph sizes. We also evaluate the scalability of Medusa on the multi-GPU environment with BFS and PageRank. The speedup of BFS and PageRank on four GPUs is 1.8 and 2.6, respectively. BFS is lightweight on computation compared with PageRank. Hence, the communication overhead is larger for BFS, which leads to fewer speedups.

5. USER EXPERIENCE AND LESSONS

In this section, we present a case study on applying Medusa to accelerate information propagation simulations over social networks. Information propagation simulations provide a flexible and valuable method to study the behaviors over complex social networks.

Large-scale network-based simulations involving information propagation often require a large amount of computing resources. It is therefore necessary to develop performance-oriented simulation techniques and to map those optimized simulation methods onto high performance computing (HPC) platforms such as GPUs. For complex networks, the network structures are highly irregular due to the complicated relationships among the verticess. Such irregularity of the data structure may exhibit very poor memory access locality. The irregularity of the network data structure poses great challenges on efficient GPU acceleration.

5.1 Models of Information Propagation

The simulation of information propagation is to investigate the interactive behaviors between *Active* vertices and *Inactive* vertices within a given network. Currently, the Independent Cascade Model (ICM) [5] and the Linear Threshold Model (LTM) [6] are widely used in studying the behaviors of information propagation over networks. In the ICM, we define an initial set of active vertexs A_0 at step 0. The information propagation unfolds in discrete time steps: at step t, the newly active vertex v_i has a single chance to activate its inactive neighbor u with an independent probability $p_{(v_i,u)} \in [0,1]$. If v_i succeeds in activating u, u will transit its status from inactive to active at step $t+1$ and remain active afterwards [5]. Such a process continues until no more new activations are made in a step. In the LTM, each vertex on the network is assigned with a random threshold $T_u \in [0,1]$. At step t, each inactive vertex is influenced by its active neighbors (a set A_t, where A_t is $\o$ if no active neighbor exists). The influence weight between the active vertex v_i and the inactive vertex u can be expressed as a probability $b_{(v_i,u)}$. Thus, vertex u's influence weight from its active neighbors can be calculated and represented by $\sum_{i=1}^{l} b(v_i,u)$, where l denotes the number of active neighbors and v_i is the ith active neighbor of u [6]. If the transition probability $\sum_{i=1}^{l} b(v_i,u)$ is larger than the predefined threshold value, u's status will transit from inactive to active at step $t+1$ and remain afterwords.

5.2 Implementations and Evaluations

Based on the ICM and LTM models (with adaptions to our application scenarios [11]), we first introduce two types of simulation algorithms named as C-Loop and T-Loop. 1) C-Loop: Starting from the active vertices in the network, each active vertex goes through its neighbors at each simulation step and tests whether it can propagate the information to the inactive neighbors with a specific probability. If the inactive vertices receive the information, they will change status to be active at the next step. 2) T-Loop: In contrast to the C-Loop, the T-Loop starts from the inactive vertices and traces the neighbors' status at each step. The inactive vertex can be activated at the next step by any

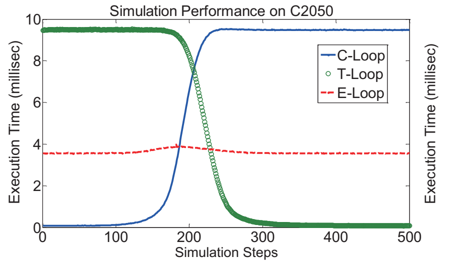

Figure 3: Execution time per simulation step.

active neighbor if the transmission probability is satisfied.

Both C-Loop and T-Loop can be easily implemented using the *ELIST* API provided by Medusa. Since the processing of neighbors in C-Loop and T-Loop imposes no edge order constraint, we also propose to use *EDGE* API to implement similar functionalities as C-Loop and T-Loop. This is inspired by the analysis from Medusa that *EDGE* API alleviates the load imbalance problem of *ELIST* API and exhibits better data access performance. We name the *EDGE* API based algorithm as E-Loop. Different from the vertex-oriented approach (C-Loop and T-Loop), the E-Loop approach starts from each edge element and checks the status of the connected pair of vertices. If the two connected vertices have different status such as Active-Inactive, the information can be propagated from active to inactive with the given probability.

Figure 3 shows the execution time per simulation step on C2050. The dataset is a random graph generated by GTGraph [7] with one million vertices and 8 million edges. Due to different memory access patterns, the above three algorithms can exhibit different costs in each step of the simulation. For example, the cost of a C-Loop step is initially small due to the small number of active vertices. As the number of active vertices increases, the cost of a C-Loop step increases. The cost of T-Loop iterations is opposite to that of C-Loop. In contrast, the cost of a E-Loop stays stable in different iterations. Compared to the CPU serial simulation performance, the C2050 GPU based simulation shows 12.5x, 13.1x, 15.6x speedup with CLoop, T-Loop, and E-Loop, respectively [11]. The different characteristics of the algorithms allow us to adaptively choose the best algorithm based on the per-step simulation information. More details on this adaptation can be found in our paper [11]. We also experiment with synthetic and real world graphs with varying sizes and obtained consistent speedups [11, 13].

Simulation on Multiple GPUs. Using multiple GPUs for the simulation is a fast way to increase the memory and computation capacities of the system. We use the default graph partitioning method of Medusa to partition the graph. Medusa automatically builds the replicas for each partition and handles the update of the replicas. Thus, with Medusa, the network-based information propagation is enabled on multiple GPUs with minimized effort. The simulation performance is improved by 2.7 times on four GPUs compared with on one GPU. More details can be found in our paper [12].

5.3 Experience on Using Medusa

Medusa requires no knowledge of GPU programming and greatly simplifies our work on utilizing GPUs for information propagation simulation. First, the programming model of Medusa fits well with the real information propagation process. Information propagation over social networks usually happens among neighboring vertices. Similarly, Medusa allows users to formulate their algorithms with the granularity of individual vertices or edges. Second, the individual and collective APIs of Medusa allow us to develop different approaches (i.e., vertex-oriented and edge-oriented) for the simulation, leading to more opportunities for improving the overall performance of the simulation. Third, despite the fact that Medusa provides an abstract data model and hides the GPU related implementation details from users, experienced users can still easily access the underlying data structures and conduct further customization. Forth, a Medusa program can transparently run on multiple GPUs. This feature of Medusa allows the user to enjoy the benefits of multiple GPUs (more memory space and computation power) with little extra implementation effort.

6. CONCLUSIONS AND FUTURE WORK

The design and implementation of Medusa show that parallel graph computation can efficiently and elegantly be supported on the GPU with a small set of user-defined APIs. The fine-grained API design and graph-centric optimizations significantly improve the performance of graph computation on the GPU. As for future work, we are considering offering dynamic graph processing support in Medusa, and extending Medusa to large scale systems such as clusters and clouds.

The source code of Medusa is available at `http://code.google.com/p/medusa-gpu/`.

7. ACKNOWLEDGEMENT

The authors would like to thank anonymous reviewers for their valuable comments. This work is supported by a MoE AcRF Tier 2 grant (MOE2012-T2-2-067) in Singapore.

8. REFERENCES

[1] 10th DIMACS implementation challenge. http://www.cc.gatech.edu/dimacs10/index.shtml.

[2] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *IPDPS*, pages 1–14, 2007.

[3] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21:1129–1164, 1991.

[5] J. Goldenberg, B. Libai, and E. Muller. Talk of the network: A complex systems look at the underlying process of word-of-mouth. *Marketing letters*, 12(3):211–223, 2001.

[6] M. Granovetter. Threshold models of collective behavior. *American journal of sociology*, pages 1420–1443, 1978.

[7] GTGraph Generator. *http://www.cc.gatech.edu/ kamesh/GTgraph/.*

[8] P. Harish and P. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *HiPC*, pages 197–208. 2007.

[9] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *PACT*, pages 260–269, 2008.

[10] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *PPoPP*, pages 267–276, 2011.

[11] J. Jin, S. J. Turner, B.-S. Lee, J. Zhong, and B. He. HPC simulations of information propagation over social networks. *Procedia Computer Science*, 9:292–301, 2012.

[12] J. Jin, S. J. Turner, B.-S. Lee, J. Zhong, and B. He. Simulation of information propagation over complex networks: Performance studies on multi-GPU. In *DS-RT*, pages 179–188, 2013.

[13] J. Jin, S. J. Turner, B.-S. Lee, J. Zhong, and B. He. Simulation studies of viral advertisement diffusion on multi-GPU. In *Winter Simulation Conference (WSC)*, pages 1592–1603, 2013.

[14] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *UAI*, 2010.

[15] S. Sengupta, M. Harris, and M. Garland. Efficient parallel scan algorithms for GPUs. NVIDIA. Technical report, 2008.

[16] Stanford Large Network Dataset Collections. *http://snap.stanford.edu/data/index.html.*

[17] T. White. *Hadoop: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2009.

[18] J. Zhong and B. He. Parallel graph processing on graphics processors made easy. *PVLDB*, 6(12):1270–1273, 2013.

[19] J. Zhong and B. He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6):1522–1532, 2014.

[20] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.