Optimizing Graph Processing on GPU'S: A Review

[1]Abhinav Singh Andotra, [2]Sonia Sharma
[1,2]Dept. of Computer Science, G.N.D.U, Amritsar, Punjab, India

Abstract
Distributed vertex-centric model has been recently proposed for large-scale graph processing. Due to the simple but efficient programming abstraction, similar graph computing frameworks based on GPUs are gaining more and more attention. However, prior works of GPU-based graph processing suffer from load imbalance and irregular memory access because of the inherent characteristics of graph applications. In this paper, we propose a generalized graph computing framework for GPUs to simplify existing models but with higher performance. In particular, two novel algorithmic optimizations, lightweight approximate sorting and data layout transformation, are proposed to tackle the performance issues of current systems. With extensive experimental evaluation under a wide range of real world and synthetic workloads, we show that our system can achieve 1.6x to 4.5x speedups over the state-of-the-art.

Keywords
GPGPU, Graph Computing, Pregel, Bulk Synchronous Model, Load Imbalance

I. Introduction

With the fast and increasing development of the Internet, processing very large web graphs has become a hot and main research issue in both the academia and industry now days. Large and big scale graph processing frameworks are becoming increasingly important for solving problems in scientific computing, data mining, and other domains such as social networks. Large graph processing is now a serious component of many data analytics. Graphs have usually been used to represent the relationship between different entities and have been the representation of choice in varioius domains, such as web page ranking, social networks, tracking drug interactions with cells, genetic interactions, and communicable disease spreading.Thus, a general graph programming framework that provides supports for high performance processing of a wide range of graph algorithms is often most wanted. As a result, in the last several years, we have observed and noticed a increasing interest in distributed graph processing, such as Pregel, GraphLab, PowerGraph, GPS and Mizan, which are purposely-built distributed graph computing systems with easy-to-use programming interfaces and practical performance under large scale workloads. Many graph processing approaches use the Bulk Synchronous Parallel model.In this execution model, graphs are processed in synchronous iterations, called supersteps. At every superstep, each vertex will execute a user function that can send/receive messages,change its value or change values of its edges. There is no defined order in which the vertices are handled within each superstep, but at the end of each superstep all vertex computations are promised to be completed.
Graph processing has been corresponding or equivalent to run on big or large compute clusters using well known cluster compute paradigms such as Hadoop and MapReduce that have been re-targeted to run graph applications. More recently, graph processing specific computing frameworks, such as Pregel and Giraph have also been planned.Pregel, for instance, relies on vertex-centric computing.An application developer describes a vertex: compute ( ) function which specifies the computation that will be performed at each vertex.The computation can be as simple as finding the minimum value of all the adjacent vertex values.The famous and well known one is Pregel,which was first time projected in 2010 as a programming model to address the challenges in parallel computing of large graphs.The high-level programming model of Pregel plays a major role in summarzing architectural details of parallel computing from programmers.Particularly, the vertex-centric programming model proposed by Pregel greatly reduces the efforts of performing computation on large-scale data-intensive graphs, and provides high expressibility for a large range of graph algorithms.GPU has become a more generalized or universal computing device.Specific purpose calculating on GPU (GPGPU) has found its way into many fields as various as biology, linear algebra, image processing, and so on, given the fantastic and great computational power provided by GPUs such as massive parallel threads and high memory bandwidth as contrasted to CPUs.In parallel to this trend, GPUs are increasingly influenced to speed up the graph algorithms with either GPU specific optimizations or Pregel like programming interfaces to cover the hardware details and working.CuSha is a graph processing framework that makes possible users to write vertex-centric algorithms on GPU, suggesting a new graph representation called G-Shards to minimize non-combined memory accesses.However, this representation is not space efficient, which may make worse the situation when more GPU memory is required to process large graphs.Medusa is a general purpose GPU-based graph processing framework that offers high-level APIs for easy programming and scales to multiple GPUs.The possible performance of Medusa may be limited by its internal data organization and processing logic that result in both irregular memory access and imbalanced workload distribution among GPU threads.Proposes techniques such as deferring outliers and dynamic workload distribution to alleviate intra-warp deviation and gets or attains a balanced load among different-different warps.However, the improvement in performance is limited because of its more or less heavyweight implementation.

II. Background

CPU-GPU heterogeneous computing has just attracted attention. GPUs provide huge or enormous parallel processing power cooperating with CPU.As the host for the GPU device, CPU organizes and calls up application kernel functions that execute on a GPU.Communication between the CPU and the GPU is performed via PCI-Express bus.GPUs consist of a number of streaming multiprocessors, each comprising of simple processing engines, called CUDA cores in the NVIDIA terminology.For instance, NVIDIA Tesla M2050 consists of 14 streaming multiprocessors (SMs), each comprising 32 stream processor (SP) cores and 64 KB shared memory shared among the SPs in a SM.
In M2050, up to 1536 hardware threads are supported and maintained by each SM and 21,504 threads are supported on the entire GPGPU.The kernel function called by host CPU is divided into many independent thread blocks or cooperative thread array (CTA).Inside of a thread block, a set of threads (32 in M2050), referred to as a warp, is scheduled on the SM to run at the same

time or alongside.A warp is a collection of threads that all run the same sequence of instructions but using different data operands. This execution model is referred to as single instruction multiple thread (SIMT) model and each thread within a warp has a dedicated set of execution resources which are referred to as SIMT lanes. GPUs provide a large, but slow off-chip global memory that can be accessed by all thread blocks and the CPU, while providing small but fast on-chip shared memories that are individually shared among threads in the same CTA.

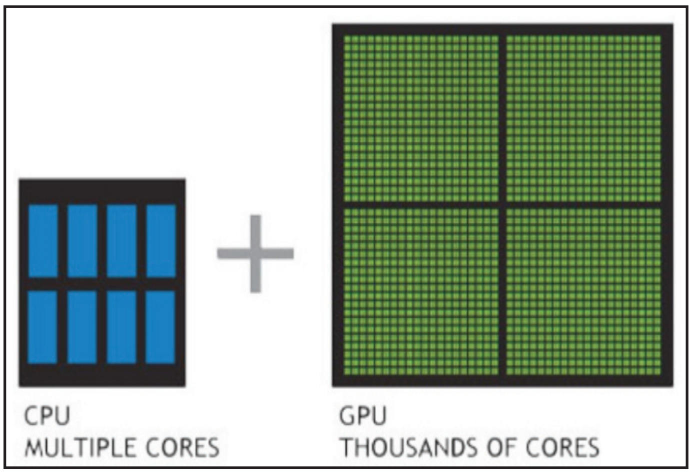


Fig. 1: Combination of CPU + GPU

III. System Design

A. Edge-Vertex Model

Firstly, we explain the reason why the vertex-centric model cannot fit in the GPU architecture directly.In the vertex- centric model, the developer needs to describes a function to perform computations on individual vertices (such as sending or receiving messages along edges).In order to fully utilize the computational resource of GPUs, we need to map the function to each GPU thread to exploit the fine-grained parallelism of GPUs.However, this benefit comes at the cost of acquiring programming complexity such as revealing the structure of the message buffer to allow open and clear management of messages.However, the expressibility of defining GPU-based computation on vertices and edges would be sufficient for most graph algorithms.

Further, summarizing the message buffer in the runtime system can not only reduce programming complexity but also provide the opportunity of implementing optimizations for different GPU hardware architectures.Therefore, we partition the computation in the vertex- centric model into two methods such as Edge Compute and Vertex Compute, which we call the Edge-Vertex model.It is based on the consideration that in our model edges are responsible for sending messages to the message buffer and vertices conduct the real computation with the messages received from the buffer.In other words, how the message buffer is constructed and managed is unseen or undetectable to the end user, striking a balance between the flexibility in optimizing certain graph algorithms and the complexity in writing graph applications.

B. Workflow and APIs

Fig. 2 below illustrates the workflow of our system.At first, the runtime system constructs the CSR representation for the input graph data, and common management tasks including memory allocation on the CPU and GPU and data transfer between the host memory and GPU are also automatically fulfilled by the runtime system.

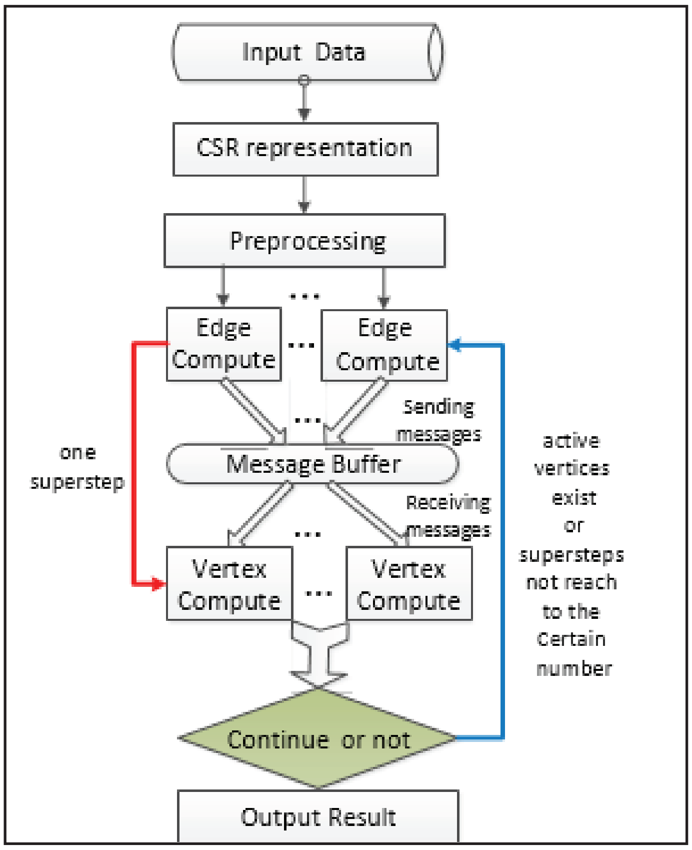


Fig. 2: The workflow for the system

1. Preprocessing

This stage is managed by the runtime and it is always transparent to the users.The internal message management module is an array-based buffer with associated operations of manipulating message delivery and reception.Before the real graph processing, we first need to establish the accurate size for the chunked message buffer with each chunk representing the storage space for messages belonging to certain vertices.In specific, for a chunk of the message buffer that is assigned to a specific vertex, we need to calculate the positions where an edge can send messages to and a vertex can receive messages from in advance.

2. Edge Compute Stage

In the Edge Compute stage, each GPU thread is responsible for one or more edges related or linked with a vertex and performs the operations of the following steps in order.Firstly, the value and state of the source vertex are read then, the weight of the edge is obtained and at the last, messages are delivered to destination vertices based on the results evaluated and calculated based on the vertex value and edge weight.

3. Vertex Compute Stage

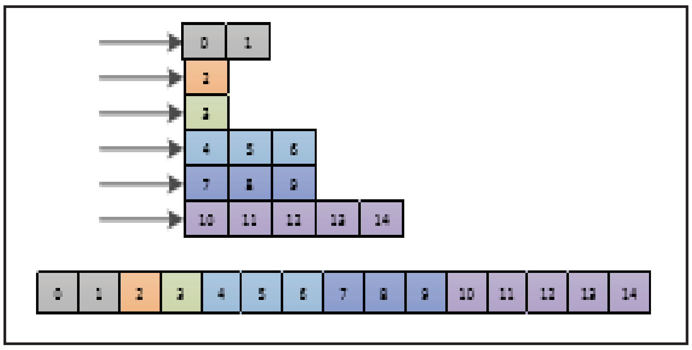
In the Vertex Compute stage, one or more vertices are given to a GPU thread that takes the following three steps similar to the Edge Compute stage.First, messages sent to the vertex are retrieved from the message buffer by the GPU thread.Second, the newly-obtained messages are used to compute the values for the vertex and its outgoing edges.Third, new state or value can be set to the vertex if needed.

4. Message Buffer

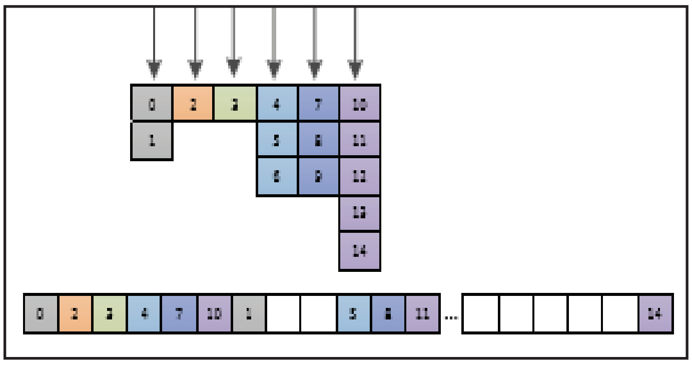
The message buffer is organized based on the total number of messages and the maximum number of messages each vertex can receive.Each edge is assigned a unique ID that is used as the index into the message buffer for both the stages.The ID values for the in-edges of a destination vertex are consecutive, which guarantees that messages sent to the same vertex are located contiguously in memory.Thus, the combiner in the Vertex Compute stage can process the messages using a simple loop.

5. A Running Example

In graph algorithms, finding the single source shortest paths (SSSP) in a graph is a well-known and easy-to-understand graph algorithm. For this graph algorithm, we need to mention a vertex as the single source vertex and find a shortest path between the source vertex and every other vertex in the graph.We use a running example to show how the three major functions Edge Compute,Vertex Compute, and Combiner are defined to implement the SSSP algorithm in our system.The SSSP procedure works as follows. In each iteration, we first get the source node of each edge, and send messages to sink nodes if the source node is updated in the previous iteration.Then, in the vertex compute stage, the minimum value is calculated by scanning a segment of the message buffer that belongs to the current vertex.If it is less than the value from the previous iteration, the old value is updated with the new one, and the vertex votes to continue to the next step.Otherwise, the vertex deactivates itself by voting to halt.The whole procedure terminates when all vertices are simultaneously inactive.



(a). Row Major



(b). Column major

Fig. 3: Two Kinds of Data Layouts

IV. Overall Performance

We implement three representative graph algorithms using the Medusa API as the baseline to compare the overall performance between our system and Medusa.The execution time contains two parts: the preprocessing time and the graph processing time. The three representative graph algorithms used for comparison are: PageRank, single source shortest path (SSSP) algorithm, and the HCC algorithm to find connected components in a graph.As shown in Fig. 3, the PageRank algorithm is run for 50 iterations, and we can observe that the our system reaches from 1.7x to 2.3x speedup.When apprassing the SSSP algorithm, we arbitrarily select the source vertex in the workloads.Figure (g) demonstrates that the speedup of our system ranges from 1.6x to 4.5x. In Figure (g), we can see 1.7x to 3.7x performance improvement for the HCC algorithm as compared to Medusa.The SSSP algorithm with the dataset 'com' achieves the best speedup (4.5x), mainly because of its simpler internal implementation logic as compared to the other two algorithms.Given that the 'com' graph is the most complex dataset we tested, this significant speedup for SSSP, from another perspective, reflects the effectiveness of our optimizations in handling large graphs.In summary, our system can attain 1.6x to 4.5x speedup for the three representative graph algorithms across all workloads in contrast to Medusa, and we plan to perform experimental evaluations on more graph algorithms and datasets in future work.

V. Conclusions

Graph processing is a key component of many data analytics.There have been several studies to optimize graph applications on GPU platform. However, there has not been a study that focuses on how graph applications interact with GPU microarchitectural features. In this paper, we propose a general graph computing framework for GPUs that achieves the goals of easy-to-use and good performance with a simplified programming model. Specifically, we develop a Edge-Vertex model in order to better utilize the fine-grained parallelism of GPUs; and we identify that load imbalance can be alleviated with the lightweight approximate sorting and that non-combined memory access can be diminshed by the data layout remapping.In addition, the integration of data remapping into the preprocessing of graph data can significantly improve the overall performance.As demonstrated by experimental evaluation, our system can achieve up to 4.5x performance speedup compared to the state-of-the-art across a wide range of workloads.As for future work, we are planning to extend our system to support multi-GPUs and distributed environments.

References

[1] "Apache giraph," [Online] Available: http://hama.apache.org.
[2] "Apache hama," [Online] Available: http://hama.apache.org.
[3] "Cuda profiler," [Online] Available: http://docs.nvidia.com/cuda/profiler-usersguide/.
[4] "Cudpp: Cuda data parallel primitives library," [Online] Available: http://cudpp.github.io/.
[5] "Nvidia cuda," [Online] Available: http://www.nvidia.com/object/cuda.
[6] D. A. Bader, K. Madduri,"Gtgraph: A synthetic graph generator suite," For the 9th DIMACS Implementation Challenge, 2006.
[7] F. K. K. V. R. G. L. N. Bhuyan,"Cusha:Vertex-centric graph processing on gpus," in Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC 2014). ACM, 2014, pp. 239–252.
[8] S. Brin and L. Page,"The anatomy of a large-scale hypertextual web search engine," Computer networks and ISDN systems, Vol. 30, No. 1, pp. 107–117, 1998.
[9] Y. Bu, B. Howe, M. Balazinska, M. D. Ernst,"Haloop: Efficient iterative data processing on large clusters," Proceedings of the VLDB Endowment, Vol. 3, No. 1-2, pp. 285–296, 2010.
[10] D. Cederman, P. Tsigas,"Gpu-quicksort: A practical quicksort algorithm for graphics processors," Journal of Experimental

Algorithmics (JEA), vol. 14, p. 4, 2009.

[11] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, E. Chen,"Kineograph: Taking the pulse of a fast-changing and connected world," In Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012, pp. 85–98.

[12] B. V. Cherkassky, A. V. Goldberg, T. Radzik,"Shortest paths algorithms: Theory and experimental evaluation," Mathematical programming, Vol. 73, No. 2, pp. 129–174, 1996.

[13] J. Dean, S. Ghemawat,"Mapreduce: Simplified data processing on large clusters," Communications of the ACM, Vol. 51, No. 1, pp.107–113, 2008.

[14] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good et al., "Pegasus: A framework for mapping complex scientific workflows ontodistributed systems," Scientific Programming, Vol. 13, No. 3, pp. 219–237, 2005.

[15] A. Gharaibeh, L. B. Costa, E. Santos-Neto, M. Ripeanu, "A yoke of oxen and a thousand chickens for heavy lifting graphprocessing," In Proceedings of the the 21st International Conference on Parallel Architectures and Compilation Techniques. ACM, pp. 345–354, 2012.