An Overview of Medusa: Simplified Graph Processing on GPUs

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

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

Abstract

Graphs are the de facto data structures for many applications, and efficient graph processing is a must for the application performance. GPUs have an order of magnitude higher computational power and memory bandwidth compared to CPUs and have been adopted to accelerate several common graph algorithms. However, it is difficult to write correct and efficient GPU programs and even more difficult for graph processing due to the irregularities of graph structures. To address those difficulties, we propose a programming framework named Medusa to simplify graph processing on GPUs. Medusa offers a small set of APIs, based on which developers can define their application logics by writing sequential code without awareness of GPU architectures. The Medusa runtime system automatically executes the developer defined APIs in parallel on the GPU, with a series of graph-centric optimizations. This poster gives an overview of Medusa, and presents some preliminary results.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming-Parallel programming; D.2.13 [*Software Engineering*]: Reusable Software-Reusable libraries; I.3.1 [*Computer Graphics*]: Hardware Architecture-Graphics processors

General Terms Algorithms, Performance

Keywords GPGPU, GPU Programming, Graph Processing, Runtime Framework

1. Introduction

Graphs are de facto data structures in various applications such as social networks modeling and web link analysis. Graph processing algorithms have been the fundamental tools (e.g., visualization [7] and exploration [1]) in various fields. Developers usually apply a series of operations on the graph edges and vertices to obtain the final result. The operations can be BFS, PageRank, shortest path and even their customized variants. For example, developers may apply different application logics when traversing each edge/vertex in the BFS process. The efficiency of graph processing is a must for high performance of the entire system. On the other hand, writing a graph algorithm from scratch is inefficient and time consuming, which loses the opportunities of sharing the same operation patterns, optimization techniques and common software components among different algorithms. A programming framework with programmability on supporting common graph processing applications and with high efficiency can greatly improve productivity.

Recently, graphics processors (GPUs) have been used as an accelerator for various graph processing applications [3, 5]. The GPU has evolved into a powerful many-core processor for general-purpose computation. New-generation GPUs can have over an order of magnitude higher memory bandwidth and higher computation power (in terms of GFLOPS) than CPUs. While existing GPU-based graph processing implementations have significant performance improvement over their CPU-based counterparts, they are limited to specific graph operations. We usually need to implement and optimize GPU programs with little reuse or even from scratch for different graph processing tasks.

Writing a correct and efficient GPU program is challenging, especially for graph applications. First, the GPU is a many-core processor with massive thread parallelism. To fully exploit the GPU computation power, algorithms should be designed with sufficient fine grained parallelism. Second, the GPU has a memory hierarchy that is different from the CPU. Since graph applications usually involve irregular accesses to the graph data, careful designs on the data layout and memory accesses are the key factor to the efficiency of GPU acceleration. Finally, developers have to explicitly manage the GPU programming details. Device management programming such as PCI-e data transfer should be carefully examined for efficiency. All these factors make programming the GPU for graph processing a difficult task.

We propose a software framework named Medusa to simplify programming graph algorithms on the GPU. The merits of Medusa are easing the pain and increasing productivity of adopting GPUs in graph computation tasks. Medusa offers a sequential programming interface and encapsulates the GPU programming complexities into a runtime system. Like existing GPU programming frameworks such as Mars [4], Medusa provides a small set of APIs for developers to implement their applications. The APIs are defined at the granularity of edges and vertices for fine-grained parallelism. Medusa automatically executes developer-defined APIs in parallel on a single GPU or multiple GPUs, and hides the complexity of parallel programming and GPU programming details.

This poster gives an overview of Medusa design and its optimizations, and presents our preliminary results. We demonstrate that Medusa achieves our goal of simplifying graph processing on the GPU, with high programmability and efficiency.

2. Overview

Medusa runs on one or multiple GPUs within the same machine, as illustrated in Figure 1. We develop a graph-centric programming interfaces named "EMV (Edge-Message-Vertex)". It provides the APIs on graph edges and vertices, which only require sequential C programming. Based on the EMV model, developers implement their algorithms using the EMV APIs. On supporting data flows among graph vertices, we develop an efficient message

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

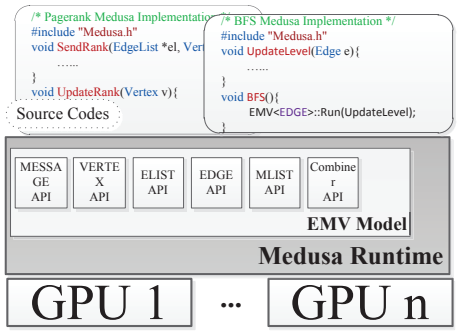

Figure 1. An overview of Medusa.

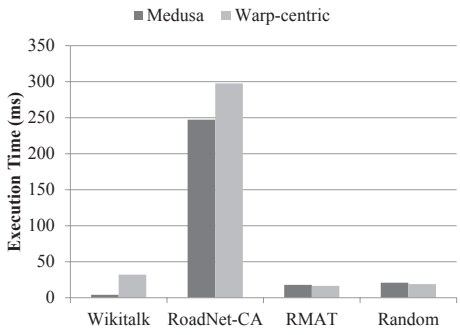

Figure 2. Performance comparison on BFS implementation between Medusa and the warp-centric approach [5].

passing mechanism that supports multiple GPUs. Medusa supports message passing between any pair of vertices. EMV supports two variants of APIs for individual and collective operations on graph edges/vertices and intermediate messages. The individual APIs take individual edge, message or vertex as input. The collective APIs works on groups of edges (grouped by head or tail vertex) or messages (grouped by destination vertex). Data exchange among vertices are realized by messages passing.

Medusa runtime is designed to meet the performance goal and to hide the underlying hardware details. We further develop a series of memory optimizations to improve the runtime efficiency. We briefly describe them here, and refer readers to our technical report [8] for more details.

- To reduce the uncoalesced memory access caused the irregular access patterns of graph algorithms, we study performance of different graph layouts on GPU and propose a novel layout to speed up iterative access of each individual edge in an edge group.
- Supporting message passing is a challenging task and we developed a graph-aware buffer scheme for message passing between adjacent vertices. Our message passing approach notably outperforms previous approaches such as hash table and pre-indexing [4].

Additionally, the Medusa runtime is extended to execute on multiple GPUs and this extension is transparent to developers. This further reduces the complexity of GPU programming. Medusa programs written for one GPU can run on multiple GPUs without modification.

We deliver Medusa with static libraries and source code templates. The current implementation is based on NVIDIA CUDA, and it is our future work for the implementations on OpenCL. The developer defined APIs are implemented as functors and passed to the system provided functions. This design overcomes the programming constrains of CUDA while offers both good encapsulation and simplicity. Tedious GPU related functions such as data transfer and multi-GPU coordination are implemented within the runtime. We expose the necessary configuration settings via simple C/C++ interfaces.

3. Preliminary Results

Our preliminary results examine the efficiency of Medusa in comparison with existing state-of-the-art GPU implementations. We conduct experiments on the Amazon EC2 Cluster GPU instance, which is equipped with two Nvidia Tesla M2050 GPUs. We choose both synthetic graphs (Random and RMAT) and real-world graphs (WikiTalk and RoadNet-CA) as our experimental data sets. We use the GTgraph graph generator [2] to generate the RMAT and Random graph. Wikitalk and Patent are downloaded from the Stanford Large Network Dataset Collection [6].

Figure 2 shows the performance comparison of the BFS algorithm between the warp centric method [5] and Medusa on a single GPU. Medusa achieves similar or even better performance than the warp-centric method. Moreover, Medusa greatly simplifies implementation of GPU programs for graph processing, with much fewer lines of source code written by developers.

4. Conclusion and Future Work

Massive parallelism architectures like GPUs have imposed great challenges in writing correct and efficient parallel graph processing programs. We propose a programming framework named Medusa, to address the efficiency and programmability issues of GPU-based graph processing. Medusa embraces an optimized runtime system to hide the programming complexity of implementing parallel graph computation tasks for the GPUs. Our preliminary results demonstrate the effectiveness of our optimizations and the efficiency of the framework. We are currently extending Medusa to utilizing both both GPUs and CPUs for heterogeneous execution as well as GPUs in a cluster. Beyond that, we are also interested in extending Medusa to handle dynamic graphs.

Acknowledgement

This work is supported by an NVIDIA Academic Partnership (2010-2011) and an AcRF Tier-1 grant in Singapore. The authors would like to thank Gao Cong for his constructive comments in the project.

References

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *SC*, pages 1–11, 2010.

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

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

[4] 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.

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

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

[7] J. Zhong and B. He. Gviewer: Gpu-accelerated graph visualization and mining. In *SocInfo*, pages 304–307, 2011.

[8] J. Zhong, B. He, and G. Cong. Medusa: A unified framework for graph computation and visualization on graphics processors. Technical Report NTU-PDCC, Feb 2011. URL `http://pdcc.ntu.edu.sg/`.