Optimization of Asynchronous Graph Processing on GPU with Hybrid Coloring Model

Xuanhua Shi[1], Junling Liang[1], Sheng Di[2], Bingsheng He[3], Hai Jin[1], Lu Lu[1], Zhixiang Wang[1], Xuan Luo[1], Jianlong Zhong[3]

[1]Services Computing Technology and System Lab/Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China
[2]Argonne National Laboratory, USA
[3]School of Computer Engineering, Nanyang Technological University, Singapore
[1]{xhshi, junlingliang, hjin, llu, wangzhx123, luoxuan}@hust.edu.cn, [2]disheng222@gmail.com, [3]{bshe, jlzhong}@ntu.edu.sg

Abstract

Modern GPUs have been widely used to accelerate the graph processing for complicated computational problems regarding graph theory. Many parallel graph algorithms adopt the asynchronous computing model to accelerate the iterative convergence. Unfortunately, the consistent asynchronous computing requires locking or the atomic operations, leading to significant penalties/overheads when implemented on GPUs. To this end, coloring algorithm is adopted to separate the vertices with potential updating conflicts, guaranteeing the consistency/correctness of the parallel processing. We propose a light-weight asynchronous processing framework called Frog with a hybrid coloring model. We find that majority of vertices (about 80%) are colored with only a few colors, such that they can be read and updated in a very high degree of parallelism without violating the sequential consistency. Accordingly, our solution will separate the processing of the vertices based on the distribution of colors.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Design, Algorithms, Performance

Keywords Graph Processing, Asynchronous Computing, GPGPU

1. Introduction

Modern GPGPUs are often used to accelerate the graph processing algorithms, not only because GPU has many more cores for parallel computing, but also due to much higher memory bandwidth and lower latency. Whereas, there are still some issues in the existing solutions.

- Most existing GPU-accelerated graph frameworks (such as Totem [1], Medusa [2], CuSha [3]) are designed based on the synchronous processing model - Bulk Synchronous Parallel (BSP) model. Such a model, however, will introduce a huge cost in synchronization especially as the graph size grows significantly, because any message processing must be finished in the previous super-step before moving to the next one.
- In comparison to the synchronous model, there are some asynchronous models that have been proved more efficient in processing graphs, but they are not very suitable for parallel graph processing on GPU. In order to ensure the correct/consistent processing results in the parallel computations, many existing solutions (such as GraphLab [4] and GraphChi [5]) adopt fine-grained locking protocols or update most vertices sequentially for simplicity. Locking policy, however, is unsuitable for GPU-based parallel processing because of the huge cost of the locking operations on GPU.

In our work, we design a lock-free parallel graph processing method named Frog[1] with a graph coloring model. In this model, each pair of adjacent vertices with potential update conflicts will be colored differently. The vertices with the same colors are allowed to be processed simultaneously in parallel, also guaranteeing the sequential consistency of the parallel execution. We observe that a large majority of vertices (roughly 80%) are colored with only a small number of colors (about 20% or less). Based on such a finding, our solution will process the vertices based on their coloring distributions. In particular, we process the vertices with the same colors in a high degree of parallelism on GPU, and process the minority of the vertices that account for majority of colors in a separate super-step.

2. Design Overview

Many graph algorithms have been shown to converge faster when solved asynchronously. Synchronous computation incurs costly performance penalties since all vertices have been updated all the time and the runtime of each phase is determined by the slowest GPU thread. While having a faster convergence, we should ensure the serializability in our asynchronous approach: all parallel executions have an equivalent sequential execution to make sure the computation correctness.

A classic technique to achieve a serializable parallel execution of vertices in a graph is to construct a vertex coloring that assigns a color to each vertex such that no adjacent vertices share the same color. For a data graph with billions of vertices, hundreds of colors can be used to complete the graph coloring. We observe that a large majority of vertices are colored with only a small number of colors. Hence, we need a relaxed graph coloring algorithm to

[1] The code can be found at https://github.com/AndrewStallman/Frog.git

PPoPP '15, February 7–11, 2015, San Francisco, CA, USA.
ACM 978-1-4503-3205-7/15/02.
http://dx.doi.org/10.1145/

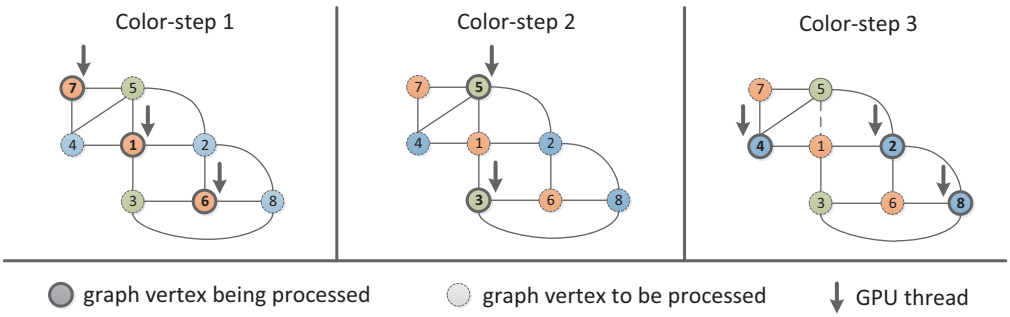

Figure 1: Asynchronous execution based on the hybrid coloring

Table 1: Execution time on different algorithms, datasets and frameworks (in milliseconds)

Algorithm	Dataset	Amazon	DBLP	WikiTalk	LiveJournal	Improvement
BFS	Frog	**4.968**	**3.380**	**3.532**	**7.482**	–
	Totem	52.155	53.478	25.923	272.335	15.82X - 36.4X
	CuSha	27.304	23.915	12.311	203.534	3.5X - 27.2X
CC	Frog	**11.762**	**10.748**	**12.986**	**213.720**	–
	Totem	50.156	52.371	26.275	294.390	1.4X - 4.8X
	CuSha	27.779	24.574	14.120	224.653	1.05X - 2.4X

color the graph with only a samll number of colors. Based on the relaxed coloring algorithm, we can develop our asynchronous graph processing system Frog which ensures the serializability.

We partition the graph based on the colors assigned to vertices, and process/update the vertices in parallel. Vertices with the same color are partitioned into the same chunk (also called *color-chunk* in the following text), which is to be processed in a separate *color-step*. The asynchronous execution model aims to process the color-chunks generated by our coloring algorithm one by one, and also guarantee the sequential consistency for each chunk.

Figure 1 shows us the asynchronous execution based on the hybrid coloring of a sample graph. In this example, we get three colors/partitions and process vertices V_1, V_6 and V_7 simultaneously in *color-step 1* (shown as the left sub-figure), and V_3 and V_5 can be processed simultaneously in *color step 2* (shown as the middle sub-figure). As shown in the right sub-figure, vertices V_2, V_4 and V_8 are assigned into the same partition. In this hybrid partition, V_2 and V_8 are adjacent vertices. As such, the color-step 3 is not the edge consistency model based step and we need to process this partition using different methods than previous partitions.

Table 2: Properties of real-world graphs used in our experiments

Datasets	Amazon	DBLP	WikiTalk	LiveJournal
Nodes	735,322	986,286	2,394,385	4,847,571
Edges	5,158,012	6,707,236	5,021,410	68,475,391

While processing the graphs, an update function is able to use the most recent values of vertices updated by the previous color-step. Between two color-steps, we need some operations to guarantee the data consistency. Accordingly, we combine several different colors together into the last partition, and process them separately: the vertices/edges are updated sequentially or updated using GPU atomic operations. Hence, we group n color-steps into two categories: *P-steps* and *S-step*. The first $n-1$ color-steps will be processed as the P steps, which means vertices and edges of color-step can be updated *in parallel* without concerning the data conflicts and consistency; for the nth hybrid partition, vertices and edges must be processed *sequentially* or using GPU atomic operations as the S step. The basic scheme is the simple kernel execution as per one color-step.

3. Performance Evaluation

We evaluate two algorithms in our experiments, Breadth First Search (BFS) and Connected Component (CC). We conduct our experiments on a Kepler-based GPU, NVIDIA Telsa K20m with 6GB main memory and 2688 CUDA cores. Table 1 shows the performance comparison between our asynchronous approach and two other systems, Totem [1] and CuSha [3]. Table 2 shows the properties of four real-world graphs that we used. Our system has an appreciable performance improvement as shown.

Acknowledgments

This work is supported by the NSFC under grants No.61133008 and No. 61370104, Chinese Universities Scientific Fund under grant No. 2014TS008, the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357, and Tencent. Bingsheng He and Jianlong Zhong are partly supported by a MoE AcRF Tier 2 grant (MOE2012-T2-2-067) in Singapore.

References

[1] A. Gharaibeh, L. Beltrao Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *PACT*, 2012.

[2] J. Zhong and B. He. Medusa: Simplied Graph Processing on GPUs. In *TPDS*, 2013.

[3] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. CuSha: vertex-centric graph processing on GPUs. In *HPDC*, 2014.

[4] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. In *VLDB*, 2012.

[5] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, 2012.