FlexBFS: A Parallelism-aware Implementation of Breadth-First Search on GPU

Gu Liu, Hong An, Wenting Han, Xiaoqiang Li, Tao sun, Wei Zhou, Xuechao Wei, Xulong Tang

School of Computer Science & Technology, University of Science and Technology of China

gliu@mail.ustc.edu.cn, {han,wthan}@ustc.edu.cn, {lixq520,suntaos,greatzv,xcwei,tangxl}@mail.ustc.edu.cn

Abstract

In this paper, we present FlexBFS, a parallelism-aware implementation for breadth-first search on GPU. Our implementation can adjust the computation resources according to the feedback of available parallelism dynamically. We also optimized our program in three ways: (1)a simplified two-level queue management,(2)a combined kernel strategy and (3)a high-degree vertices specialization approach. Our experimental results show that it can achieve 3∼20 times speedup against the fastest serial version, and can outperform the TBB based multi-threading CPU version and the previous most effective GPU version on all types of input graphs.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Distributed programming, Parallel programming; D.3.3 [*Language Constructs and Features*]: Frameworks, Patterns

General Terms Algorithms, Performance

Keywords Graph algorithms, Breadth-first Search, CUDA, GPGPU

1. Introduction

Graphics processing unit (GPU) has recently become a popular parallel platform for general computing but it still can't dominate this field because many irregular applications exist. These irregular programs are usually involved with pointer-based data structures like graphs and trees, and share some common features in runtime characteristics which are critical to GPU architecture. One example of these irregular applications is the Breadth-first search (BFS). Several published works[1–3] have tried to implement BFS effectively on GPU, but the performance optimization is still a big problem. One of the challenges is to make the thread configuration adapt to the parallelism patterns of input graphs. Traditional BFS implementations on GPU used fixed thread configuration, which requires knowledge of parallelism pattern of the input graph prior to the kernel launching. Unfortunately, the parallelism patterns are usually in great variety and hard to predict, which contributes to one of the reasons for the low efficiency of BFS implementation on GPU.

In this paper, we will present a parallelism-aware GPU implementation with corresponding optimization techniques.

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

Contributions:(1)We analyzed and profiled the available parallelism within BFS of different types of input graphs. The knowledge of various parallelism patterns can be used to yield a parallelism-feedback implementation for BFS.(2)We implement FlexBFS to effectively process working sets with different parallelism patterns for breadth-first search on GPU. (3)We introduced three optimization approaches. Our solution gained up to 20x speedup against the fastest serial version, and outperformed TBB based multi-threading CPU version and the previous most effective GPU version on all kinds of input graphs.

2. FlexBFS: Our GPU Implementation

2.1 Baseline

The available parallelism within BFS can be represented by the size of the frontier, in which the active nodes can be explored in parallel. By profiling the available parallelism of our benchmark graphs, we found they have a wide range of parallelism variation space, and the parallelism pattern within a single graph could be rather complex. In our baseline implementation, we used a global counter to record the size of the new frontier. At the end of the kernel, this counter is feedback to the host to assign adequate threads for the next kernel iteration. Since extra work generated by the redundant threads is cut off, GPU resource utilization is much higher than the fixed thread configuration.

Unfortunately, the baseline version of FlexBFS still surfers performance degradation which comes from the following three reasons. First, the non-coalesced memory accesses to the global frontier queue; second, the kernel launch overhead of separate kernel function call for each level; third, the imbalanced workload in those irregular graphs with skewed structure.

2.2 Optimizing Techniques

Two-level queue management In our baseline implementation, store operations into the next frontier queue will causes non-coalesced memory accesses. As a result, we present a two-level queue structure. A fast block-level queue is built in the shared memory within each multiprocessor, while the grid-level queue is still in the global memory. Each thread within a warp first stores the neighbor id into the block-level queue, then copy back into the grid-level queue in a regular way. It is noticeable that each thread access global memory at consecutive addresses, which meets the condition of coalescing memory operation.

Combined kernel strategy In the absence of support to global thread synchronization in CUDA, we have to launch one kernel instance for each level, which will bring huge kernel-launch overhead. Thus, we use an inter-block synchronization technique[4] to implement a combined kernel strategy, which lower down the launch overhead while the thread configuration still maintain flexible.We first evaluate the new frontier size. If it is within a certain

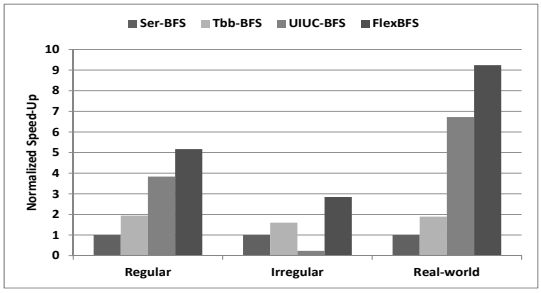

Figure 1. Comparison of average performances obtained by different BFS implementation.

range, there is no need to launch a new kernel instance. If the new frontier size is out of this threshold, a new thread configuration then will be set up for the next kernel invoking.

High-degree vertices specialization Load imbalance is a severe problem in BFS algorithm when input graph is irregular. To avoid this problem we present the high-degree vertices specialization approach. We process those high-degree vertices separately using a special queue. We first check the degree of each active node; if the degree is larger than a threshold, this node is inserted into a special queue instead of exploring neighborhood. After other low-degree vertices are finished, nodes in special queue are processed using all available threads.

3. Experimental Setup

Our experiments were conducted on an NVIDIA Tesla C2050 GPU with an Intel quad core Xeon E5506 CPU. To cover all types of graph instances, we choose three kinds of graphs as our input working set. They are classified as regular, irregular and real-word graphs. The regular graph is generated based on grid graph. The irregular graphs are built on the bases of regular ones. We add 0.1% high-degree vertices into the original grid graph, which causes extremely load imbalance. We also used a set of real-world graphs. They are traffic route nets in different parts of USA from DIMACS challenge website.

4. Experimental Results

To evaluate the performance of our FlexBFS, we compare it with the serial version and TBB version running on CPU and UIUC-BFS [3] running on GPU. The optimal TBB performance is obtained using 4 threads on our CPU platform.

Figure 1 illustrates the performance of the four BFS implementations on 3 sets of input graphs. Generally speaking, our solution outperforms the other ones. The FLexBFS obtains an average speedup of 5 times faster than serial CPU version and 2.5 times against TBB version. Our method is also 1.5 times faster than UIUC-BFS, the previous most effective GPU implementation, because of the parallelism-aware scheme and other optimization methods. It is noticeable that the performance of UIUC-BFS is particulary poor when processing irregular graphs, even worse than CPU versions. L.Luo et. al. mentioned in their paper that they had to convert the irregular graphs into near-regular graphs by splitting the big-degree nodes before applying their BFS implementation. Without this pretreatment process, their version suffers seriously from imbalanced work. In Section 2.2, we introduced three optimization methods to improve performance. We combine these techniques with the baseline implementation into three configurations summarized in Table1. Figure 2 illustrates the effectiveness of our proposed optimization methods. In most cases, the baseline FlexBFS is well

Name	Description
Fixed	Basic GPU BFS implementation using a fixed thread configuration
Baseline	baseline FlexBFS implementation using flexible thread configuration
Config1	Baseline + Combined kernel strategy
Config2	Config1 + Two-level queue management
Config3	Config2 + High-degree vertices specialization

Table 1. Optimization configurations of FlexBFS

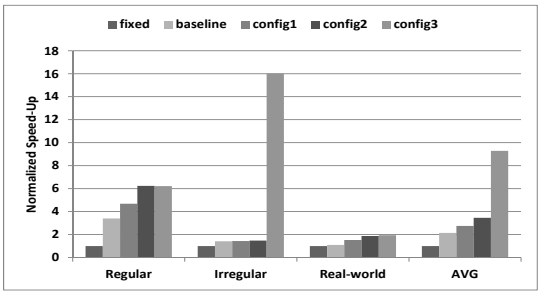

Figure 2. performance of BFS under a set of configurations using different optimization methods.

performed than the fixed configuration, which demonstrates that unexplored parallelism does exist in the fixed-thread BFS implementation. Config1 shows that the combined kernel strategy can shorten the overall execution time by up to 30%. Config2 improves the performance of regular graphs by reducing the random accesses overhead to a great extend. The rightmost bars of each column in figure 2 display the impact of the high-degree vertices specialization, from which the irregular graphs can benefit most.

5. Conclusions

We have presented a parallelism-aware implementation for breadth-first search algorithm on GPU. Our solution can deploy proper thread resources according to the dynamic profile of parallelism in BFS. We have analyzed the main reasons of performance degradation in BFS, and proposed several optimizing approaches for these problem. The experimental results show that our solution achieves up to 20x speedup over the fastest serial BFS program, and outperforms TBB version and previous GPU implementations on all three types of graphs.

Acknowledgments

This work is supported financially by the National Basic Research Program of China under contract 2011CB302501,the National Natural Science Foundation of China grants 60970023, the National Science & Technology Major Projects 2009ZX01036-001-002 and 2011ZX01028-001-002-3.

References

[1] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In HiPC'07.

[2] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. In PPoPP'11.

[3] L. Luo, M. Wong, and W. mei Hwu. An effective gpu implementation of breadth-first search. In 47th DAC 2010.

[4] S. Xiao and W. chun Feng. Inter-block gpu communication via fast barrier synchronization. In IPDPS 2010.