ATLANTIS PRESS

Advances in Intelligent Systems Research, volume 134

2nd International Conference on Control, Automation, and Artificial Intelligence (CAAI 2017)

A Synchronization Mechanism between CUDA Blocks for GPU

Bingru Wang[1], Changyou Zhang[2], Feng Wang[3] and Jun Feng[1,*]

[1]Shijiazhuang Tiedao University, Shijiazhuang, China

[2]Institute of Software, Chinese Academy of Science, Beijing, China

[3]Shenyang Institute of Automation, Chinese Academy of Science, Shenyang, China

***Abstract*—GPU(Graphic Processing Unit) provides a promising solution with massive threads and its advantage is high performance computing. The emergence of CUDA(Compute Unified Device Architecture) opens the door of using GPU's powerful computing power. However, because of the limitation of CUDA itself, direct communication is not supported between SMs(streaming multiprocessors) on GPU. It is time-consuming by atomic operation or barrier synchronization. A synchronization mechanism has been proposed in this paper, that is, on the premise of result available, the times of kernel launched should be reduced. Each kernel launched, it should be computed enough on GPU, the results back to the CPU. Based on SSSP, the validity of this method is illustrated by delta-stepping. For facebook dataset, compared with atomic operation, the speedup ratio is about 1.8. For New York map dataset, compared with atomic operation and barrier synchronization, the speedup ratio is about 9.3 and 1.7 separately.**

Keywords-GPU; synchronization mechanism; SSSP; parallel computing; delta-stepping; CUDA

I. INTRODUCTION

In today's world, graph is widely used in many field such as social network, web network, biological data analysis, etc. As social develop, the size and complexity of graph is increasing. Parallel computing under traditional CPU has not satisfied people's requirements for computing speed. Like CPU, GPU is designed for complex mathematical and geometric calculations. Early phase, the emergence of GPGPU(general-purpose GPU), it began to be used by researchers, but there are many shortcomings on GPGPU, such as low hardware programmability, difficult application development, etc. The emergency of CUDA opens the door of GPU's powerful computing ability for universal computing.

When it comes to parallel computing, synchronization must be mentioned. For GPU, CUDA supports communication between intra-SMs, which can be realized by function __syncthreads(). However, CUDA does not directly support communication between inter-SMs, so the computing ability is limited. Traditionally, barrier synchronization occurs by terminating the current GPU-offloaded computation and re-launching a new GPU-offloaded computation. In CUDA 2.2, NVIDIA introduced the function __threadfence(), which could improve performance and guarantee the correctness between inter-SMs on the GPU. However, this function introduced so much overhead when using it in direct GPU synchronization. It could be more worse than (indirect) barrier synchronization[1].

In order to solve this problem, reduce the synchronization overhead and improve the whole computing efficiency, this paper proposes a synchronization mechanism, signal mechanism on GPU. Two sets of experiments prove the effectiveness of this method.

II. GPU ARCHITECTURE AND CUDA

GPU, a graphics processing unit, early for graphic image processing. Figure I shows the CPU and GPU hardware architecture comparison. CPU is oriented to general calculation, a large number of transistors are used for cache and control circuits[2]. GPU is oriented to parallel computing based on computationally intensive and large amounts of data, a large number of transistors are used for the calculation unit. In addition, programming models such as NVIDIA's Compute Unified Device Architecture, and the recently available OpenCL allow applications to be more easily mapped onto the GPU.

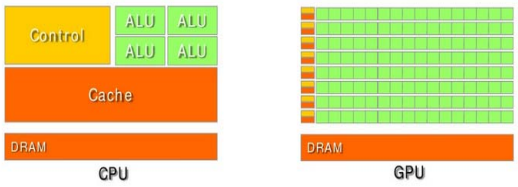

FIGURE I. THE CPU AND GPU HARDWARE ARCHITECTURE COMPARISON

NVIDIA Corporation released its first microprocessor-specific GPU in 1999, specifically for processing realistic images. Early stage, GPU is only used to speed up on compute-intensive graphics and image processing applications, and the locality of access memory is very good. So compared with CPU, more computing units used to realize the parallel processing are integrated on GPU, less units used to control and buffer. Therefor, SPMD (Single Program multiple Date) types of data-level fine-grained parallel problems are very suitable for execution on GPU[3]. Soon after, NVIDIA launched CUDA (Computer Unified Device Architecture) general parallel computing architecture and general parallel programming model[4], which greatly improves the efficiency of GPU program development. At present, CUDA has become the mainstream environment in the field of GPU. With the improving performance and programmability on GPU, it has evolved into general processor with high parallelism, strong computing ability and high DRAM bandwidth[5].

III. GPU SYNCHRONIZATION MECHANISM AND EXISTING PROBLEMS

A complete CUDA program should be consists of two parts, host program and equipment kernel function, as is shown in Figure II. Host program is performed by CPU, including data preparation before starting kernel, the initialization, data processing between kernels and serial code before program ending.

Usually, CPU + GPU heterogeneous model is adapted[6], that moves from the GPU to the CPU and back to GPU, that is barrier synchronization. CPU is responsible for not suitable for data parallel computing, such as complex logic processing and transaction processing. GPU is responsible for the mass data parallel computing of computationally intensive[8]. In this method, synchronization overhead is expensive.

GPU threads are organized by grid way, which contains a number of blocks. GPU usually maps well only to data-parallel or task-parallel whose execution requires relatively minimal communication between SMs on the GPU. This proclivity is mainly due to the lack of support for communication between SMs on the GPU. The threads in same block allow fine-grained parallel and communication. However different threads in different blocks are difficult to achieve communication, and the applications is very common, such as SSSP. A solution is put forward in this paper to solve synchronization issue between blocks.

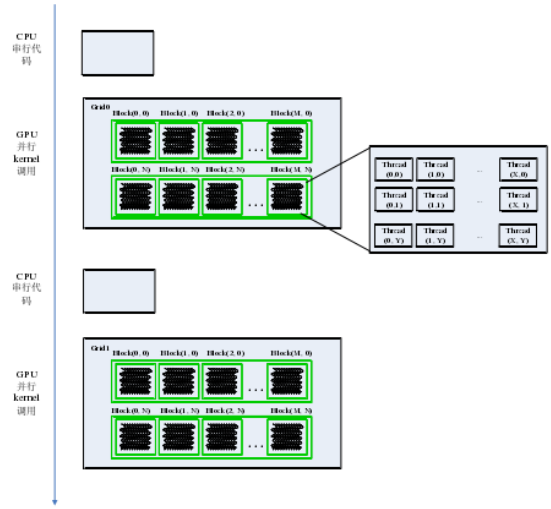

FIGURE II. THE COMPOSITION OF CUDA PROGRAM AND EXECUTION MODEL

IV. SSSP

Single-source shortest path[7]: for a given map, select a node as the source point, remembered as s. There may be more than one path form s to other nodes. The key of the problem is that find one path which is the shortest from s to the node. Namely the sum of weights through this path is shortest. Now, the popular serial algorithm to solve SSSP is Dijkstra[8], and parallel algorithm is △-stepping[9].

A. *Dijkstra Algorithm*

Dijkstra algorithm is one of the most popular serial methods to solve SSSP, based on lable-setting. All the nodes V are divided into three sets. Settled (S), the set of nodes that has been relaxed to the shortest path, tent[s]=0; Queued (Q), the set of nodes that has been relaxed, but does not know whether is the shortest path; Unreached (U), the set of nodes that has not been relaxed, tent[v]=∞. Dijkstra algorithm selects one node that the value is the smallest from Q and adds this node to S. The result depends on the previous update operation of Q. Dist[v] is the shortest path from s to v, tent[v] is always the weight of some path from s to v and hence an upper bound on dist[v].

B. *Delta-stepping Algorithm*

In 2003, △-stepping algorithm[10] was put forward by Meyer and Senders[11], aimed at solving shortest path problem in parallel to speed up the calculation. The difference between Dijkstra and △-stepping algorithm is that the former only selects one node from Q set to S set every time, but the latter can select more than one node every time. △-stepping algorithm introduces the concept of bucket, forms an array bucket[], and the width of bucket[] is △. Bucket[i] stores {v∈V: v is queued and tent[v]∈[i*△,(i+1)*△] }. All the edges are divided into light edges(the weight of edges c(e)≤△) and heavy edges(the weight of edges c(e)>△). Light edges might lead to new nodes for the current bucket, but for the remaining heavy edges, it is sufficient to relax them once and for all when a bucket finally remains empty.

Because the advantage of GPU is parallel computing, the improved △-stepping algorithm no longer distinguish between light edges and heavy edges, all nodes are calculated as light edges that need multiple iterations.

The detail of improved △-stepping algorithm is an follows.

```
foreach v ∈ V do                          --initialize node data structure
    tent[v]= ∞                                          --unreached
relax(s,0); i=:0                          --Source node at distance 0
while ¬ isEmpty(bucket) do                --some queued nodes left
    S:=Ø                                  --No nodes deleted for this bucket yet
    while bucket[i] ≠ Ø do                                --new phase
    Req := {{w, tent[v]+c(v,w)}:v ∈ bucket[i]}
    S:=S⋃ bucket[i]; bucket[i]:= Ø
                                          --remember deleted nodes
    foreach (v,x) ∈ Req do relax(v,x)
                                          -- This may reinsert nodes
    i := i+1                                            -- next bucket

Procedure relax(v,x)                      --shortest path to v?
    if x<tent[v] then
                                          --Yes: decrease-key respectively insert
        bucket[⌊tent[v] / Δ⌋] := bucket[⌊tent[v] / Δ⌋] \ {v}
                                          --remove if present
        bucket[⌊x / Δ⌋] := bucket[⌊x / Δ⌋]⋃ {v}
                                          --insert into new bucket
        tent[v]:=x
```

V. SYNCHRONIZATION MECHANISM

In parallel computing, dist[] values need to deliver them to CPU, which might be modified by some thread. Dist[] is stored in global memory. When relaxed, one value of dist[] might been modified by several threads in different blocks simultaneously, writing error may occur[1]. In order to prevent this, atomic operation was brought forward by CUDA. The nature of atomic operation is to luck and unlock the shared resource, that is, some thread takes up shared resource exclusively. Atomic operation can be effective in preventing the occurrence of writing error, but the efficiency of this method id very inefficient in large-scale graph.

__syncthread() can achieve synchronization within the block, but in CUDA, that has no defined method to achieve synchronization between blocks. Pass intermediate results to CPU to achieve synchronization between blocks[12].

In order to promote efficiency, this paper proposes a synchronization mechanism. We define a set semaphores of Boolean type, modified[], whose length is the same with blocks[13].

It is stored in global memory, initialized to False. As shown in Table Ⅰ.

TABLE I. A SET SEMAPHORES

blocks	0	1	2	...	N
semaphores	Modified [0]	Modified [1]	Modified [2]	...	Modified [N]

Because modified[] is stored in global memory, it can be modified by all the blocks. Writing error might occur. In order to reduce this mistake, it can increase the layer number of semaphore, and every layer is independence. Figure 3 shows the flow chart of semaphores.

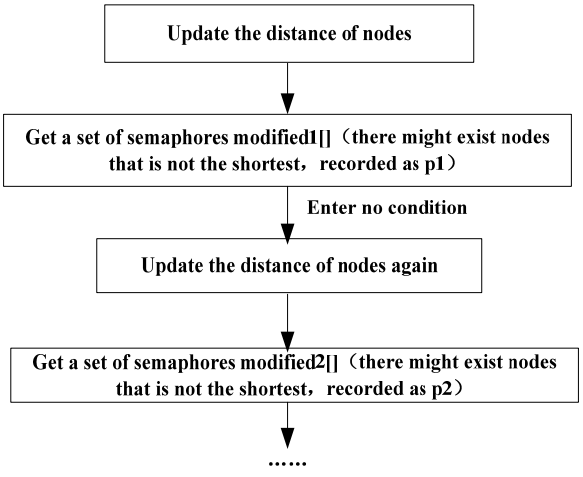

FIGURE III. THE PROCESS OF MULTI-LAYER SEMAPHORES

Figure Ⅲ shows the process of multi-layer semaphores, p1>p2>...≥0.

VI. EXPERIMENTS

Hardware environments are an follows. GPU model is NVIDIA GeForce GTX480, CUDA version is CUDA 6.0. CPU version is AMD Phenom(tm)II X4 960T, 4 cores.

For large-scale graph, the purpose of this experiment is to demonstrate the effectiveness of this method.

In this experiment, there are two data sets, one is facebook dataset from Stanford Large Network Dataset Collection[14], the other is New York map dataset from the ninth DIMACS application challenge[15].

A. *Facebook Dataset*

This dataset has 4039 nodes, 88234 edges. The max degree is 1045, min degree is 1, average degree is 21.85. Degree distribution is shown in Figure Ⅳ.

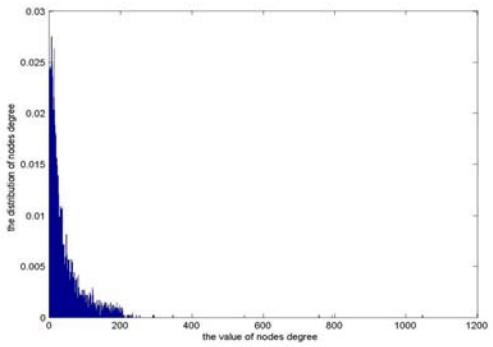

FIGURE IV. DEGREE DISTRIBUTION OF FACEBOOK DATASET

For facebook dataset, time consumption with different methods is shown in Figure Ⅴ. Y-axis is the time consumption (in ms).by atomic operation and one set of semaphores separately

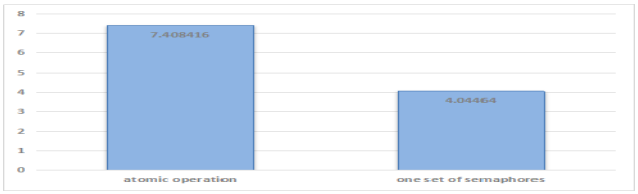

FIGURE V. THE TIME CONSUMPTION BY ATOMIC OPERATION AND ONE SET OF SEMAPHORES SEPARATELY

For facebook dataset, because the data volume is small, one set of semaphores is enough to ensure the results correctness.

B. *New York Map Dataset*

This dataset has 264346 nodes, 733846 edges. The max degree is 8, min degree is 1, average degree is about 3. Degree distribution is shown in Figure VI.

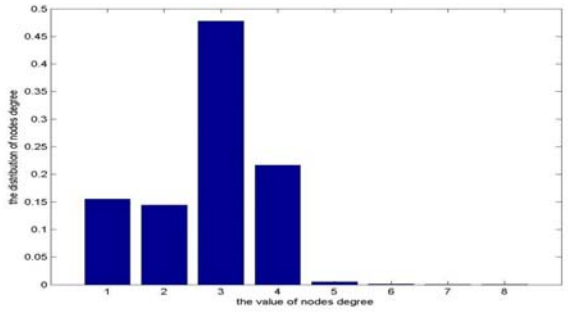

FIGURE VI. DEGREE DISTRIBUTION OF NEW YORK MAP DATASET

For New York map dataset, time consumption with different methods is shown in Figure Ⅶ. Y-axis is the time consumption (in ms) by atomic operation, synchronize by CPU, one set of semaphores, and two sets of semaphores separately.

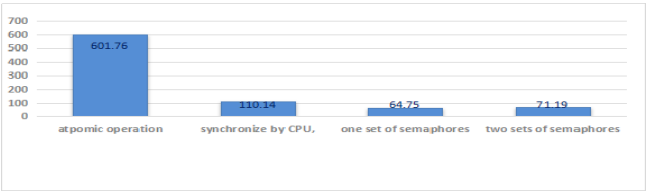

FIGURE VII. THE TIME CONSUMPTION BY ATOMIC OPERATION, SYNCHRONIZE BY CPU, ONE SET OF SEMAPHORES, AND TWO SETS OF SEMAPHORES SEPARATELY

For New York map dataset, because the data volume is relatively big, one set of semaphores is not enough to ensure the results correctness, several nodes still need further calculation. For this dataset, two sets of semaphores ensure the results correctness.

VII. CONCLUSION AND PROSPECT

According to the experimental results, compared with atomic operation and synchronize by CPU, the method proposed in this paper is effective. The bigger graph scale is, the more observe speeding up might be within a certain range. For facebook dataset, the difference of degree is bigger, compared with atomic operation, the speedup ratio of semaphores proposed in this paper is about 1.8. For New York map dataset, the distribution of degree is more stable. Compared with atomic operation and synchronize by CPU, the speedup ratio of semaphores proposed in this paper is about 9.3 and 1.7 separately.

So far, we have made a preliminary exploration for synchronization on GPU, and have made some achievements, but there are still many questions for further study, such as how many layer should be set to ensure the result right in different graph, which should be very meaningful and interesting.

ACKNOWLEDGEMENT.

This paper is supported by the Natural Science Foundation of China (61672508), Hebei Province Natural Science Foundation (F2013210109). The authors also would like to express appreciation to the anonymous reviewers for their helpful comments on improving the paper.

REFERENCES

[1] Feng W, Xiao S. To GPU synchronize or not GPU synchronize?[C]. Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on. IEEE, 2010: 3801-3804.

[2] Jason Sanders, Edward Kandrot, CUDA by example : an introduction to general-purpose GPU programming[M]. 2011.

[3] Bombieri N, Busato F, Fummi F. A fine-grained performance model for GPU architectures[C]//Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016. IEEE, 2016: 1267-1272.

[4] Ziliaskopoulos,Athanasios K.;Mandanas,Fotios D.;Mahmassani,Hani S..An extension of labeling techniques for finding shortest path trees[J]. European Journal of Operational Research. 2009, Vol.198(No.1) : 63-72.

[5] Meyer, Ulrich1 umeyer@mpi-sb.mpg.de.Average-case complexity of single-source shortest-paths algorithms: lower and upper bounds[J]. Journal of Algorithms.2003,Vol.48(NO.1) : 91-134.

[6] Esraa Shehabb; Alsayed Algergawyab; Amany Sarhanb. Accelerating relational database operations using both CPU and GPU co-processor[J].Computers and Electrical Engineering.2017 : 69-80.

[7] Changyou Zhang; Feng Wang; Kun Huang; Zhiyou Liu; Yifeng Chen. UniDegree: A GPU-Based Graph Representation for SSSP[J].Algorithms and Architectures for Parallel Processing.2015 : 704-715.

[8] Yefim Dinitz; Rotem Itzhak. Hybrid Bellman－Ford－Dijkstra algorithm[J].Journal of Discrete Algorithms.2017 : 35-44.

[9] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. Journal of Algorithms. 49 (2003) , 114-152.

[10] Feng Wang; Liehuang Zhu; Changyou Zhang. SSSP on GPU Without Atomic Operation[J].Human Centered Computing.2016 : 409-419.

[11] Esraa Shehabb; Alsayed Algergawyab; Amany Sarhanb. Accelerating relational database operations using both CPU and GPU co-processor[J].Computers and Electrical Engineering.2017 : 69-80.

[12] Yefim Dinitz; Rotem Itzhak. Hybrid Bellman－Ford－Dijkstra algorithm[J].Journal of Discrete Algorithms.2017 : 35-44.

[13] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. Journal of Algorithms. 49 (2003) , 114-152.

[14] http://snap.stanford.edu/data/egonets-Facebook.html.

[15] http://www.dis.uniroma1.it/challenge9/download.shtml.