GPU-Based Fast Minimum Spanning Tree Using Data Parallel Primitives

Wei Wang[1] Shaozhong Guo[1] Fan Yang[2] Jianxun Chen[1]
[1]Zhengzhou Information Science and Technology Institute, Zhengzhou, 450002, China
[2]The Hongqiao Road No.2285, Shanghai, 200336, China
wangwei8137@gmail.com

***Abstract*-Minimum spanning tree is a classical problem in graph theory that plays a key role in a broad domain of applications. This paper proposes a minimum spanning tree algorithm using Prim's approach on Nvidia GPU under CUDA architecture. By using new developed GPU-based Min-Reduction data parallel primitive in the key step of the algorithm, higher efficiency is achieved. Experimental results show that we obtain about 2 times speedup on Nvidia GTX260 GPU over the CPU implementation and 3 times speedup over non-primitives GPU implementation.**

Keywords-GPU; minimum spanning tree; Prim's algorithm; data parallel primitives; CUDA

I. INTRODUCTION

A. MST Problem and Prim's Algorithm

Minimum spanning tree (MST) is a fundamental concept in classic graph theory. It is defined as follows: Given an undirected graph $G = (V, E)$ with a weight mapping $w : E \rightarrow R$, find a connected sub graph $T = (V, E' \quad E)$ with $|E'| = |V| - 1$ that minimizes the objective function $\sum_{e \in E'} w(e)$. MST plays a key role in a broad domain of applications, including network organization, touring problems and VLSI layout.

Prim's algorithm is one of the most commonly used MST algorithms. The serial computational complexity of Prim's algorithm implemented with traditional data structure is $O(|V|^2)$. Research of Prim's algorithm concentrates on its serial version. There have been several parallel formulations of Prim's algorithm [1-3]. The common disadvantage of these algorithms is that the speedup is limited compared with parallel Borůvka's MST algorithm especially when the graph size is very large.

B. GPU-Based Accelerating Applications

Graphics Processor Unit (GPU) is used for many general purpose applications recently. Modern GPU provides high computational power at a low cost. Many new development platforms such as CUDA [4] and OpenCL [5] make GPU become a affordable and accessible computing coprocessor. Due to the features of GPU architecture, those applications that have good speedup effect are mostly belong to regular problem. GPU model is best suited to process independent data instances. Processing less regular data on GPU architectures is a challenge to programmers [6]. These irregular problems include graph theory and computing geometry. MST problem is an irregular problem. To our best knowledge, there has been no GPU version of parallel Prim's algorithm up to the present.

C. Our Work and Contribution

The use of efficient primitives to map the irregular aspects of problem to the data-parallel architecture of these massively multithreaded architectures is central to obtain high performance. In this paper, we design and implement the parallel Prim's algorithm using data parallel primitives under CUDA architecture on GPU. Our work extends and improves the approach of Mark Harris et al. [7] and designs new GPU Min-Reduction primitive suited to Prim's algorithm. After fully optimization, when the input size is 16384 vertices, we achieve about 2 times speedup on GTX260 GPU over the BGL CPU serial implementation and about 3 times speedup over non-primitives GPU implementation. Besides, the reason of limited speedup in our implementation is also be analyzed.

II. RELATED WORK

A. Parallel Prim's Algorithms

Kumar et al.[8] pointed out that the outer while loop in serial Prim's algorithm is hard to parallelize due to its inherent character, but the two inner loop steps can be parallelized and they are: finding minimum weighted edge in the candidate edge set and updating the candidate edge set.

There are several parallel implementations of Prim's algorithm. Rohit Setia et al. [1] presented a new parallel Prim's algorithm targeting SMP with shared address space, and obtained 2.64 times speedup. Gonina et al. [2] used a novel extension of adding multiple vertices per iteration to achieve significant performance improvement under MPI environment. Bader et al. [3] proposed a parallel MST algorithm which uses a hybrid approach of Borůvka's and Prim's algorithm.

B. GPU Data Parallel Primitives

Data parallel primitives are common fundamental parallel operations when developing parallel algorithms. In the GPU parallelization process of serial algorithm aimed at irregular problem, the use of data parallel primitives can achieve crucial performance improvement. Mark Harris et al. [7] presented a reduction primitive implementation on GPU using CUDA. Blelloch [9] formulated MST algorithm using the scan primitive on an EREW PRAM model. Vineet et al. [10] used three GPU primitives to solve MST problem using Borůvka's approach. Aydin Buluc [11] researched the application of GPU

Supported by the National Natural Science Foundation of China (863 Program, Grant No.2009AA012201) and the Key Programs for Science and Technology Development of Shanghai Science Committee (Grant No.08dz501600).

978-1-4244-7941-2/10/$26.00 ©2010 IEEE

data parallel primitives in several graph theory problems such as APSP (All Pairs Shortest Path).

III. DESIGN AND IMPLEMENTATION

A. Graph Representation

Traditional data structure for graph representation includes adjacency matrix and adjacency list. Adjacency matrix is suitable for dense graph representation, but the $O(|V|^2)$ space requirement is too large to be suitable for the limited GPU device memory. Adjacency list is suitable for sparse graph and its space requirement is only $O(|V|+|E|)$. Under CUDA architecture, device memory is treated as general arrays and can be accessed efficiently. In this paper, we use compress adjacency list that is represented in the complete arrays, it can be read and written efficiently by GPU.

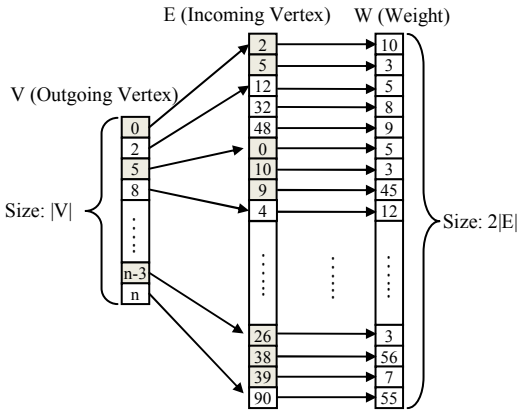

Figure 1. Compress adjacency list

Compress adjacency list (Fig. 1) consists of three arrays: vertex array V, edge array E, weight array W. The length of V is $|V|$ and its element points to the start index in E and W. The graph in MST algorithms is a undirected graph, so the lengths of E and W are both $|E|*2$.

B. GPU Min-Reduction Primitive

Reduction data parallel primitive is a kind of parallel operation which processes a group of data elements and gets a single value, such as sum, min and max. Based on the GPU Sum-Reduction primitive presented by Mark Harris et al. [7], we develop a new GPU Min-Reduction primitive which is suitable for Prim's algorithm. It contains two kinds of search: global memory search and shared memory search. Table I states the important values and arrays and their purpose.

TABLE I. IMPORTANT ARRAYS AND VALUES

Name	Purpose
V, E, W	Store vertices, edges, weights.
R1, R2, R3	MST edge list (outgoing vertex, incoming vertex, weight).
T1, T2	Store temporary reduction results (value, index).
R3'	Current searching section in R3.
C	Store R2 vertex of the edge newly added to MST.
Kernel1, Kernel2	Two CUDA kernels in GPU Min-Reduction.

1) Improvement and extension:

There are three main extensions. a) The number of data elements is not confined to power of 2, it can be random number; b) It can obtain final result after at most two kernel invoke steps; c) We add a index array which contains the corresponding index of the minimum weight in R3.

2) Global memory search:

Fig. 2 illustrates the global memory search of GPU Min-Reduction. This search is used in Kernel1. The boundary of threads' global memory access is modified to the length of R3', so it adapts to array with any length. We obtain the flexibility at a cost of some performance loss in nature. Due to the definition of MAX_BLOCKS, we can foresee the number of temporary reduction results. The reduction will finish after at most two kernel invoke steps and it facilitates the host invoke.

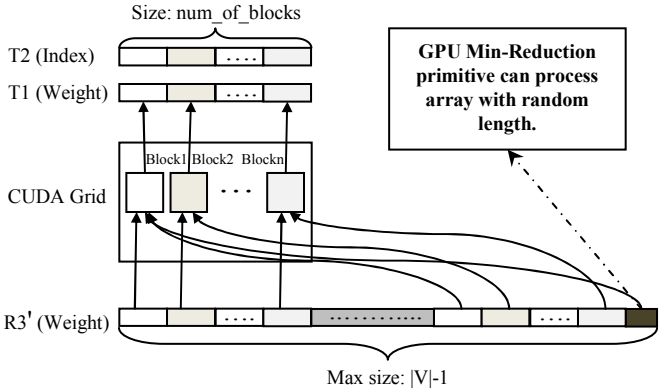

Figure 2. Global memory search

3) Shared memory search:

Fig. 3 illustrates the shared memory search of GPU Min-Reduction. This search is used in the second half of Kenrel1 and Kernel2. Because the index of minimum weighted edge is needed in the candidate edge list adjust, we add index arrays in all steps. There are two skills here. One skill is adjacent threads operating adjacent elements, so bank conflicts in shared memory access is avoided. The other skill is canceling _syncthreads() function in operations of the last 32 threads, it saves possible performance loss in thread synchronism.

C. Minimum Spanning Tree Algorithm

Prim's algorithm belongs to greedy algorithm. It starts by selecting an arbitrary vertex as the root of the tree. It then grows the tree by adding a vertex that is closest to the current tree and adding the minimum weighted edge from any vertex already in the tree to the new vertex. The algorithm terminates once all vertices have been added to the tree [11]. The output is a list of edges present in MST.

1) Growing MST:

Initializing MST edge list: In order to exploit efficiency of GPU Min-Reduction primitive, we divide traditional MST

edge list into three arrays (R1, R2, R3) whose lengths are all $|V|$-1. The same position holds three properties of one edge respectively: outgoing vertex, incoming vertex and weight. They hold all edges that start from vertex 0. If there is no edge between vertex 0 and one vertex, write MAX_WEIGHT to the corresponding position in R3.

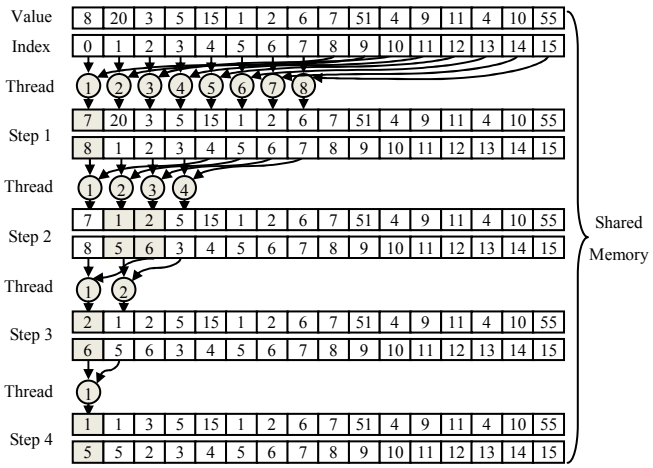

Figure 3. Shared memory search

Finding minimum weighted edge: This step use GPU Min-Reduction to find minimum weighted edge in R3' and its corresponding index (Fig. 4). We introduce two temporary arrays (T1, T2) to hold the weight and index. There are two kernels in this step. T1 and T2 hold the results of Kernel1. Kernel2 is invoked only When |R3'|>MAX_BLOCK* MAX_THREADS_PER_BLOCK. Unlike Kernel1, Kernel2's operating targets are T1 and T2 instead of R3', and its results are T1[0] and T2[0]. We use static shared memory instead of dynamic shared memory in Mark Harris et al.'s [7] implementation due to the limitation of number of CUDA blocks. The size of used shared memory in every block is |MAX_THREADS_PER_BLOCK|*2.

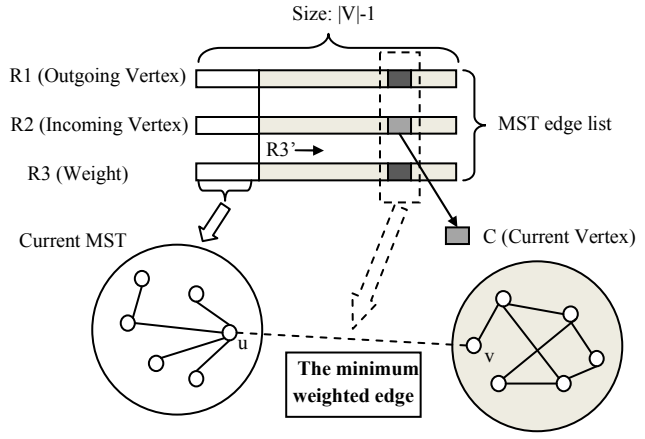

Figure 4. Finding minimum weighted edge

Adding new MST edge: T1[0] and T2[0] are read in this step, then we add minimum weighted edge and its vertex to the MST by moving the edge to the first position of MST candidate edge list. If Kernel2 is invoked, it takes place after Kernel2, otherwise after Kernel1. The other operation is saving the current vertex to C. All operations of this step are processed by one thread to avoid access conflict in global memory.

2) Comparing and updating MST:

Global variable C is read and current vertex is obtained in this step. The weights between current vertex and other vertices can be found through referring to E, V and W. The index of current vertex is assumed to be n, and the result is Wn. If Wn < R3[n], we will adjust MST edge list by following operations: R3[n]=Wn, R1[n]=C. Data elements in MST edge list are independent to each other, so the operations described above can be parallelly handled through multiple threads.

Two skills are used in this step. One skill is using static shared memory. The other skill is searching the weight of C→n instead of n→C when searching for the weight between current vertex C and the other vertex n, and the cause is to avoid thread branch in a warp and achieve more consistency between threads when they refer to E and W. This skill is crucial to obtain efficiency under CUDA architecture.

3) Main outer loop:

The main outer loop invokes these two steps above for $|V|$-2 times. We re-define the size of CUDA block before every invoking kernel. In invoking Kernel1 and Kernel2, we adopt C++ template technique to completely unroll the reduction, and define the CUDA block size as one of these values: 256,128,64,32,16,8,4,2,1.

4) Complete algorithm outline:

Algorithm 1 CUDA_PRIM_MST

1: Read graph data from input file, initializing E, V, W
2: Construct R1, R2, R3 and global variable C in CPU host memory and initialize them.
3: Transfer E, V, W, R1, R2, R3, C to GPU device memory, construct temporary arrays T1, T2.
4: Define the size of CUDA grid and block based on the number of data elements in Min-Reduction.
5: Invokes Kernel1 and write the results to T1, T2. If reduction final results have been obtained, jump to step 7.
6: If |R3'|>MAX_BLOCK*MAX_THREADS_PER_BLOCK, invokes Kernel2, and write the results to T1[0], T2[0].
7: Read T1[0], T2[0] and add the corresponding edge to MST edge list. Write current vertex to C.
8: Re-define the size of CUDA grid and block based on the number of data elements in comparing MST step.
9: Read global variable C and get the current vertex, find the weight between current vertex and other vertices.
10: For every vertex, If new weight<old weight, adjust the corresponding values in R1 and R3.
11: Invokes step 4-10 for |V|-2 times until obtain the final result.
12: Transfer the result from GPU device memory to CPU host memory, and write the results into output file.

IV. PERFORMANCE ANALYSIS

A. Comparison Algorithms

We choose two comparison algorithms. One algorithm is CPU BGL serial Prim's algorithm [12]. The other is our new developed non-primitive CUDA Prim's algorithm, and the difference is that it adopts common GPU parallelization when finding minimum weighted edge.

B. Testing Platform

Intel Pentium4 3GHz CPU,2G host memory, NVIDIA GeForce GTX260 GPU, 896M device memory, Linux RedHat 5 OS.

C. Experimental Data

We choose the random generator from Georgia Tech graph generator suite [13]. The generated graphs have a short band of degree where all vertices lie, with a large number of vertices having similar degrees. The input graphs have 2^7-2^{14} vertices and 2^8-2^{15} edges. The weight of all edge is confined to 1-1K.

D. The Results

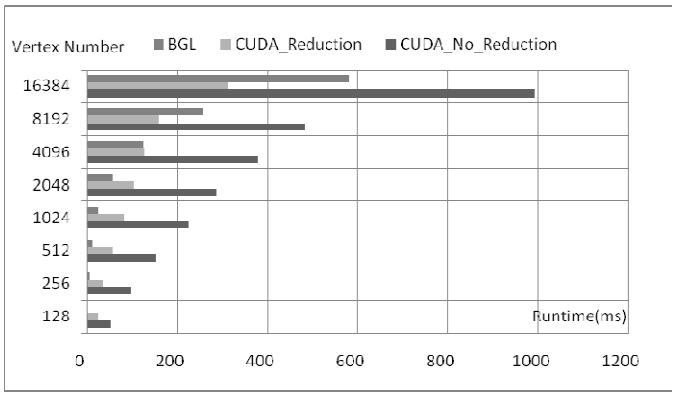

Figure 5. Runtimes of three algorithms

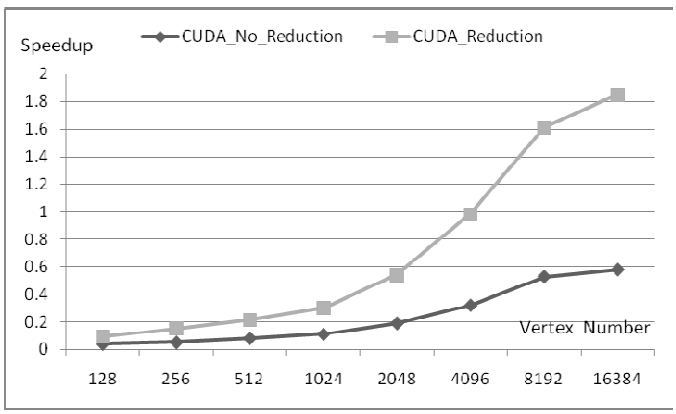

Figure 6. Speedup of two CUDA Prim's algorithms

Fig. 5 and Fig. 6 show the final results of runtimes and speedup. The performance of "CUDA_No_Reduction" algorithm is always worse than that of others. When the number of vertices is larger than 4096, the performance of "CUDA_Reduction" algorithm is better than the other two algorithms. The speedup of "CUDA_No_Reduction" algorithm is always less than 1. The maximum speedup of "CUDA_Reduction" is close to 2 and the speedup over "CUDA_No_Reduction" algorithm is nearly 3.

E. Analysis of Limited Speedup

Compared to parallel Borůvka's MST algorithm [14], the speedup of our implementation is not outstanding. We consider the main reason is that the main outer loop of Prim's algorithm is hard to parallelized, and this inherent character greatly limits the performance. The difficulty of parallelizing Prim's algorithm in MST problem is very similar to that of parallelizing Dijkstra's algorithm which is also a classic algorithm in SSSP (Single Source Shortest Path) problem.

V. CONCLUSION

In this paper, we implement Prim's algorithm using new developed Min-Reduction data parallel primitive under CUDA architecture on GPU to solve MST problem. The experimental results show that our algorithm effectively improves the performance compared to CPU BGL serial Prim's algorithm and GPU Prim's algorithm without primitives. The reason of limited speedup in our implementation is also be analyzed. We believe that using data parallel primitives in solving irregular problem including graph theory and computing geometry on GPU is helpful to achieve performance improvement.

REFERENCES

[1] R. Setia, A. Nedunchezhian, and S. Balachandran, "A new parallel algorithm for minimum spanning tree problem," Proc.International Conference on High Performance Computing (HiPC), pp. 1-5, 2009.

[2] E. Gonina and L. Kalé, "Parallel Prim's algorithm on dense graphs with a novel extension," Tech. Rep., 2007.

[3] D. A. Bader and G. Cong, "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs," Journal of Parallel and Distributed Computing, v.66 n.11, p.1366-1378, November 2006.

[4] NVIDIA Corporation. CUDA Programming Guide 2.3, http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf, 2009.

[5] A. Munshi, "OpenCL," , http://s08.idav.ucdavis.edu/munshi-opencl.pdf, 2008.

[6] R. Vuducy, A.Chandramowlishwarany, J. Choi, M. Guney, and A. Shringarpurez, "On the Limits of GPU Acceleration," http://www.usenix.org/event/hotpar10/tech/full_papers/Vuduc.pdf, 2010.

[7] M. Harris, "Optimizing parallel reduction in CUDA, " NVidia, Tech. Rep., 2007.

[8] G. Karypis, A. Grama, A. Gupta and V. Kumar. Introduction to Parallel Computing. Addison Weslesy, second edition, 2003.

[9] G. E. Blelloch, "Scans as Primitive Parallel Operations," IEEE Transactions on Computers, v.38 n.11, p.1526-1538, November 1989.

[10] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan, "Fast minimum spanning tree for large graphs on the GPU, " in HPG '09: Proceedings of the Conference on High Performance Graphics 2009, 2009, pp. 167–171.

[11] A. Buluc, "Linear Algebraic Primitives for Parallel Computing on Large Graphs," Ph.D. thesis, University of California, Santa Barbara, 2010.

[12] J. Siek, L. Lee, and A. Lumsdaine, "The Boost Graph Library: User Guide and Reference Manual," Addison-Wesley, 2002.

[13] D. A. Bader and K. Madduri, "GTgraph: A Synthetic Graph Generator Suite," Tech. Rep., 2006.

[14] V. Vineet, P. Harish, and P. J. Narayanan, "Large Graph Algorithms for Massively Multithreaded Architectures," Tech. Rep., 2009.