I.J.Modern Education and Computer Science, **2011, 4, 55-62**
Published Online July 2011 in MECS (http://www.mecs-press.org/)

Design and Implementation of GPU-Based Prim's Algorithm

Wei Wang
Zhengzhou Information Science and Technology Institute, Zhengzhou, China
Email: wangwei8137@gmail.com

Yongzhong Huang and Shaozhong Guo
Zhengzhou Information Science and Technology Institute, Zhengzhou, China
Email: xy_gsz@163.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.**

***Index Terms*—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 $w : E \rightarrow \mathbf{R}$, find a connected sub graph $T = (V,E' \subset E)$ with $|E'| = |V| - 1$ that minimizes the objective function $\sum_{e \in E'} w(e)$. Fig. 1 illustrates the MST concept described above. MST plays a key role in a broad domain of applications, including network organization, touring problems and VLSI layout. Moreover, they are typically only a part of more complex graph algorithms.

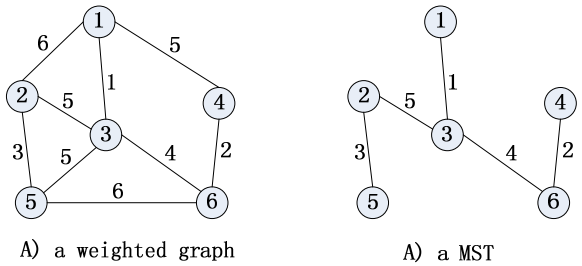

Figure 1. Minimum spanning tree concept

Prim's algorithm is one of the most commonly used MST algorithms, and it belongs to classic greedy algorithm. 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][2][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 (Compute Unified Device Architecture) [4] and OpenCL (Open Computing Language) [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

In our previous work, we have implemented the GPU version of Prim's algorithm using common approach on CUDA platform, but the performance of that version's algorithm is even worse than CPU sequential implementation in industrial Boost Graphic Library [7]. After analysis, we consider that the key problem is the parallelization difficulty of finding minimum value in the inner loop of Prim's algorithm. Recent studies on irregular algorithm reveal that 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. This result can be helpful in paralleling the step of finding minimum value.

In this paper, we design and implement the parallel Prim's algorithm using data parallel primitives under

Manuscript received January 1, 2008; revised June 1, 2008; accepted July 1, 2008.

CUDA architecture on GPU. Our work extends and improves the approach of Mark Harris et al. [8] 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. [9] 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 (i.e. Min-Reduction step) and updating the candidate edge set(i.e. Comparing and updating MST step) .

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. [8] presented a reduction primitive implementation on GPU using CUDA. Blelloch[10] formulated MST algorithm using the scan primitive on an EREW PRAM model. Vineet et al. [11] used three GPU primitives to solve MST problem using Borůvka's approach. Aydin Buluc [12] researched the application of GPU data parallel primitives in several graph theory problems such as APSP (All Pairs Shortest Path).

Besides, some GPU data parallel primitive libraries under CUDA architecture appear recently, including CUDPP [13], Thrust [14]. The utilization of these libraries will make the development of GPU application more convenient, but the disadvantage is that the primitives in these libraries are not abundant yet and sometimes may cause redundant memory access.

III. Graph Representation On CUDA Platform

A. Traditional graph representation

Traditional data structure for graph representation includes adjacency matrix and adjacency list [15]. Fig. 2 shows these two structures. For the adjacency matrix representation of a graph $G = (V, E)$, we assume that the vertices are numbered 1, 2, ... , $|V|$ in some arbitrary manner. Then the adjacency matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$, and if $(i,j) \in E$, then $a_{ij} = 1$, otherwise $a_{ij} = 0$.The adjacency list representation of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V . For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$.

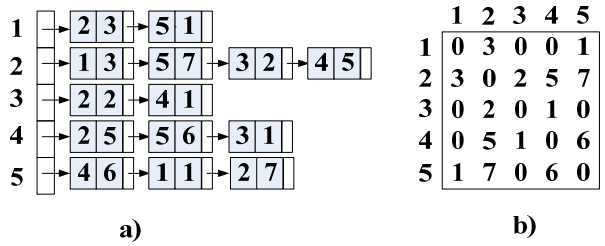

Figure 2. Traditional graph representation

After analyzing the feature of these two graph data structure, we draw the conclusion as follows. 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, especially when the graph size is very large. Adjacency list is suitable for sparse graph and its space requirement is only $O(|V|+|E|)$.But the pointer data in traditional adjacency list is too many, CUDA platform is not adept in processing pointer data, so traditional adjacency list is also not suitable for graph representation on GPU device.

B. Compact Adjacency List

In recent related studies, there are two main graph data structures on CUDA platform (Fig. 3), which are both improvement forms of traditional adjacency list.

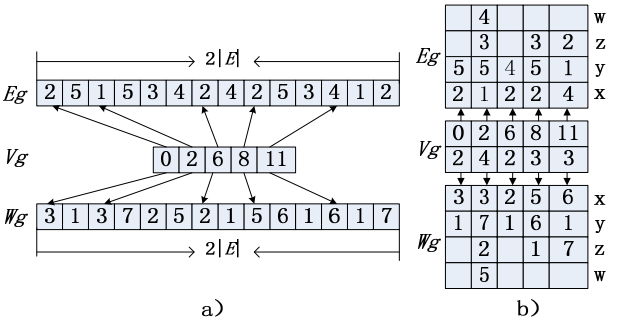

Figure 3. Compact Adjacency List

P. Harish et al.[16] propose a kind of graph representation named compact adjacency list. As Fig. 3a) shows, compact adjacency list consists of three arrays: vertex array V_g, edge array E_g, weight array W_g. The length of V_g is $|V|$ and its element points to the start index in E_g and W_g. The lengths of E_g and W_g are both $|E_g|*2$ when all edges of the graph is directed. Under CUDA architecture, device memory is treated as general arrays and can be accessed efficiently, so compact adjacency list is a efficient graph representation on GPU device. Many later studies [17] use that data structure as foundational graph representation. A. Leist et al. [18] propose another kind of graph representation which makes some

improvement on compact adjacency list. As Fig. 3b) shows, this representation uses two multidimensional arrays and adds the amount of vertices in every list compared to Fig. 3a).

C. Our Graph Representation

In this paper, we propose a new graph representation based on compact adjacency list. The key improvement of our graph representation is the usage of CUDA built-in vector data type, i.e. int2, which can be accessed efficiently under CUDA architecture.

As Fig. 4 shows, it consists of two two-dimensioned arrays, i.e. A[][],B[][]. A[][] includes information of all vertices of the graph. For every element in A[][], A[n].x is the number of vertices connected to vertex A[n] and A[n].y points to the start address in B[] connected to vertex A[n]. B[][] includes information of all edges of the graph. For every element in B[][], B[n].x represents a incoming vertex of one edge and B[n].y represents the corresponding weight of that edge. In order to fully exploit the efficiency of texture memory of CUDA memory model, we implement that graph representation in texture memory. Our experimental results show that our graph data structure obtains better performance compared to compact adjacency list.

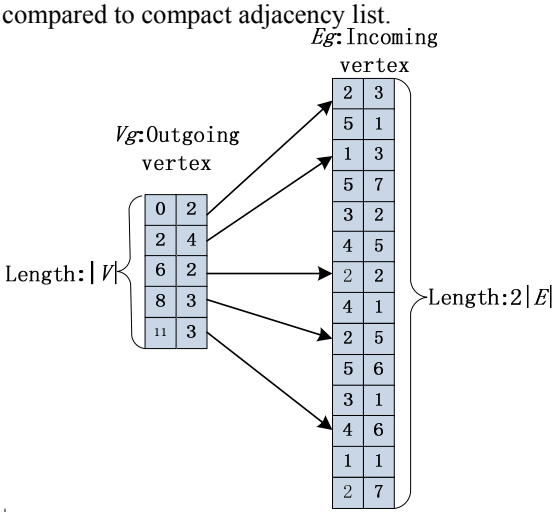

Figure 4. Our graph representation

IV. GPU MIN-REDUCTION PRIMITIVE

A. Reduction Data Parallel 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. Reduction primitive is a common process in parallel program design. Many situations of parallel computing involves comparing or summarizing all results of different threads, such as gather computing data or draw a specific value. Reduction primitive is the best choice in these situations. There are many studies about Reduction primitive under traditional parallel computing architectures. Recently, some studies about Reduction primitive under GPU architecture begin to appear. David Roger et al. [19] design and implement Reduction primitive on GPU using OpenGL shading language. Mark Harris et al. [8] proposed a GPU Sum-Reduction primitive under CUDA architecture and achieve good efficiency. For simplicity, GPU Sum-Reduction primitive will be abbreviated to GSR primitive in the rest of the paper.

B. Overall Description of GMR Primitive

Aiming at the goal of paralleling the key step of finding minimum weighted edge in Prim's MST algorithm, we improve and extend the GSR primitive, and design the GPU Min-Reduction primitive under CUDA architecture. For simplicity, we will call GPU Min-Reduction primitive GMR for short in the rest of the paper.

1) Principles of GMR's design

The fundamental principles of GMR's design include the following content. To fully exploiting the efficiency of massively multithreaded architectures CUDA platform, we apply some optimizing techniques such as successive address, static shared memory and template parameter compilation in two GMR modules, i.e. global memory reduction and shared memory reduction, and quickly obtain the minimum value and corresponding index from an array of input values.

2) Structure of GMR's modules

Fig. 5 describes the structure of GMR's modules. GMR primitive is constituted of two CUDA Kernel invokes. The reason for using multiple Kernel invokes is that there is no efficient synchronizing method among thread blocks on CUDA platform. Because computing work in the first Kernel invoke are carried out by multiple thread blocks, when the number of thread blocks is large than one, one Kernel invoke cannot get the final result. GMR primitive uses global memory to synchronize among multiple thread blocks, so the number of Kernel invokes is larger than one.

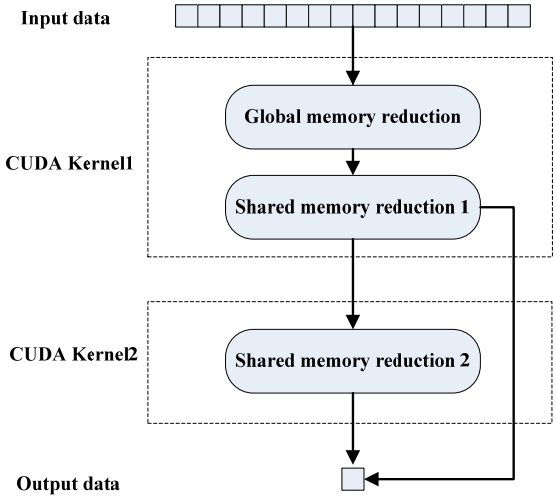

Figure 5. Structure of GMR's modules

3) Steps of GMR's process course

The basic steps of GMR's process course can be described as follows.

Firstly, all threads in thread grid make reduction operation on an array of input data in global memory, and obtain the minimum value and its corresponding index of every thread. This step changes the number of data from

random number to the number of all threads. Note that the results of the first step are stored in shared memory of every thread block.

Secondly, all threads in every thread block make reduction operation on the result data of the first step in shared memory, and obtain the minimum value and its corresponding index of every thread block. This step changes the number of data from the number of all threads from the number of all thread blocks.

If there is only one thread block, the fianl results have already obtained. But if the number of thread blocks is larger than one, the second shared memory reduction operation must be used, i.e. the third step. This step change the number of data from the number of all thread blocks to one, and the GMR primitive's process is over. The final results include two values, i.e. the minimum value and its corresponding index. In these steps, the first two steps constitute CUDA Kernel1, and the last step constitutes CUDA Kernel2.

Through these three steps of GMR's process, we can see that there are three main improvements and 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.Table I summarize the differences between GSR and GMR primitive.

TABLE I.
GMR'S IMPROVEMENT AND EXTENSIONS

Type	Function	Number of input data	Output
GSR primitive	Sum	Power of 2	Sum value
GMR primitive	Min	Random	Min value and index

C. Global Memory Reduction

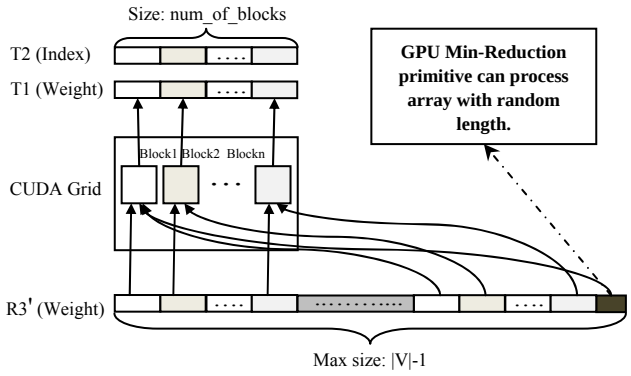

Figure 5. Global memory reduction

Fig. 5 illustrates the global memory reduction of GPU Min-Reduction. This reduction 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.

D. Shared Memory Reduction

Fig. 6 illustrates the shared memory reduction of GPU Min-Reduction. This reduction 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.

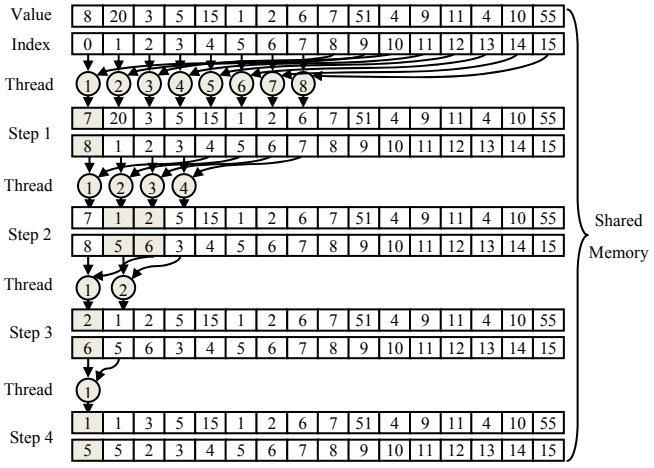

Figure 6. Shared memory reduction

V. DESIGN AND IMPLEMENTATION

A. Overall Design of GPU Prim's 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 [15]. The output is a list of edges present in MST.

We design and implement GPU Prim algorithm on CUDA platform. Fig. 7 shows the overall design flow diagram. We apply the GMR primitive in the key step of finding minimum weighted edge(i.e. Kernel1 and Kernel2 in Fig. 5), the step of comparing and updating MST edge list is paralleling by common CUDA techniques.

B. Important Arrays and Values

All data structures using in GPU Prim's algorithm include three groups. One is the graph data structure, it's the most important and the most frequently access data structure, we use the graph representation described in Fig. 4. The second data structure is used for storing algorithm results. We use three arrays as MST edge list when implementing GPU Prim's algorithm. The third group is other data structures, including temporary arrays *T1* and *T2*, global variable *C* etc. TABLE II. states the important arrays and values in GPU Prim's algorithm.

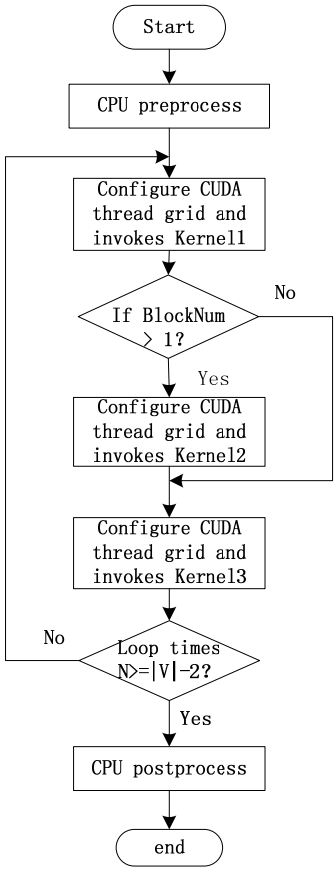

Figure 7. Overall design flow diagram

TABLE II.
IMPORTANT ARRAYS AND VARIABLES

Name	Purpose
V_g, E_g	Store vertices, edges, weights.
R1, *R2*, *R3*	MST edge list (outgoing vertex, incoming vertex, and 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.
Kernel3	Compare and update the MST edge list

C. Techniques Used in Implementation

We use some techniques in the process of implementing GPU Prim's algorithm. These techniques greatly improve the performance of program. We will describe the detail of these techniques step by step.

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 one chosen vertex (we choose vertex 0 in implementation). If there is no edge between vertex 0 and one vertex, write MAX_WEIGHT to the corresponding position in *R3*.

Finding minimum weighted edge: This step use G MR primitive introduced above to find minimum weighted edge in *R3'* and its corresponding index in *R2* (Fig. 8). 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*]. In our implementation, we use static shared memory instead of dynamic shared memory in Mark Harris et al.'s [8] 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.

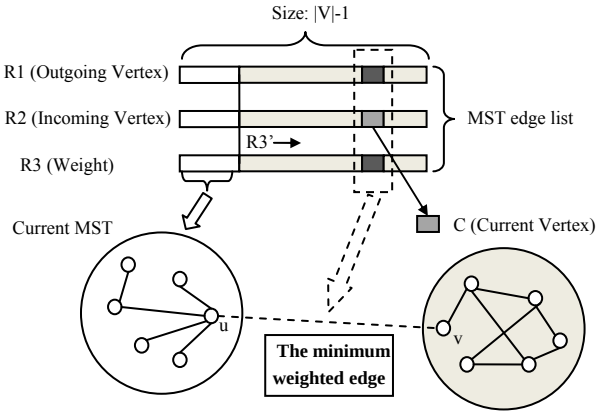

Figure 8. Finding minimum weighted edge

Adding new MST edge: After finding out the minimum weighted 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(i.e. *R1*, *R2*, *R3*). These operations take places in two possible positions. If Kernel2 is invoked, it takes place after Kernel2, otherwise after Kernel1. The other operation is saving the current vertex (i.e. *R2*[*T2*[*0*]]) , and we introduce global variable *C* for this purpose. All operations of this step are processed by one thread (we choose thread 0 in implementation) instead of multiple threads 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_g, V_g. The index of current vertex is assumed to be n, and the computing result is *Wn*. We compare Wn with the old weight *R3*[*n*]. 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 in thread grid. Similar to the reduction in global memory, every thread in GPU grid can process multiple data elements in our implementation and only one CUDA kernel invoke is needed.

Two skills are used in this step. One skill is using static shared memory, and the reason is similar to that of Kernel1 and Kernel2. The other skill is searching the weight of $C \rightarrow n$ instead of $n \rightarrow C$ 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_g. This skill is crucial to obtain efficiency under CUDA architecture.

3) Main outer loop

The main outer loop invokes these two steps for loop until the algorithm finishes. The number of loop times is a fixed number $|V|$-2. Kumar et al. [9] pointed out that the main outer loop is very difficult to run in parallel, and that is why the efficiency of parallel Prim's algorithm is worse than parallel Borůvka's MST algorithm. Unlike Mark Harris [8]'s approach, we re-define the size of CUDA block before every invoking kernel, and this can avoid the unbalanced loading in fixed size definition. In invoking Kernel1 and Kernel2, we adopt C++ template technique to completely unroll the reduction, and we define the CUDA block size in different conditions to one of these values: 256,128,64,32,16,8,4,2,1. In invoking kernel in Comparing and Updating the MST step, we define the CUDA block size to the number of data elements when the number of all data elements is less than MAX_THREADS_PER_BLOCK.

4) CPU preprocessing and postprocessing

GPU computing uses CPU-GPU cooperation pattern. Besides GPU's parallel computing work, some necessary serial work must to be processed by CPU, i.e. preprocessing and postprocessing. In preprocessing step, CPU startup the GPU device, construct graph data structures and MST edge list in CPU host memory, and read input file to initial these data structures. The other important work in this step is allocating memory space in GPU device memory with CUDA API and transferring graph data structures and MST edge list to device memory. In postprocessing step, CPU transfers the computing results from device memory to host memory, and write results to output file, free host and device memory space, finally exit from CUDA environment.

D. Complete Algorithm Outline

Algorithm 1 presents the complete algorithm as reported in the previous section.

Algorithm 1 CUDA_PRIM_MST

1: Read graph data from input file, initializing E_g, V_g.
2: Construct *R1*, *R2*, *R3* and global variable *C* in CPU host memory and initialize them.
3: Transfer E_g, V_g, *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 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.

VI. Performance Analysis

A. Comparison Algorithms

We choose two comparison algorithms. One algorithm is CPU Boost Graph Library serial Prim's algorithm [7]. 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, CUDA 2.3.

C. Experimental Data

We choose the random generator from Georgia Tech graph generator suite [20]. 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 27-214 vertices and 28-215 edges. The weight of all edge is confined to 1-1K.

D. The Results

Fig. 9 and Fig. 10 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.

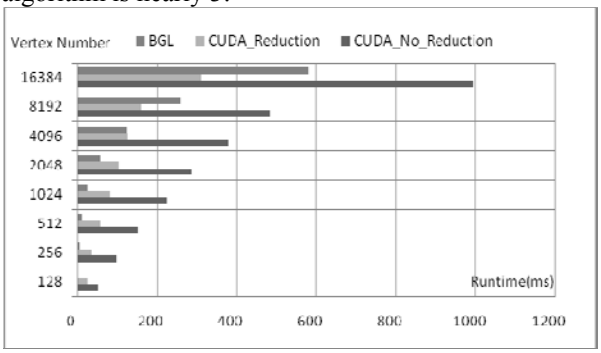

Figure 9. Runtimes of three algorithms

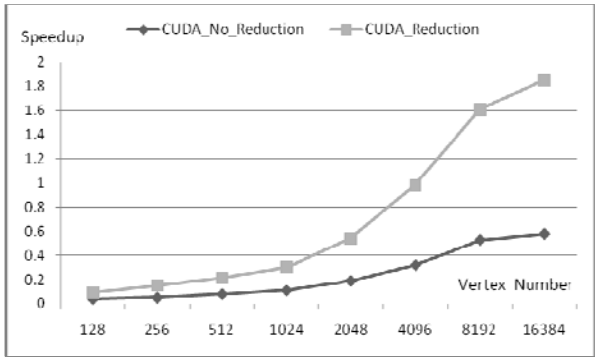

Figure 10. Speedup of two CUDA Prim'S algorithms

The result data above shows the advantage of using data primitives in performance improvements when developing GPU algorithms. Besides, we notice that different size of CUDA grid and block can result in different performance. In our implementation, we achieve the best performance when MAX_BLOCKS = 128 and MAX_THREADS_PER_BLOCK=128.

E. Analysis of Limited Speedup

Compared to parallel Borůvka's MST algorithm [16], 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.

VII. 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.

Acknowledgment

This work was 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).

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: parallel computing on the GPU and CPU," 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] J. Siek, L. Lee, and A. Lumsdaine, "The Boost graph library: user guide and reference manual," Addison-Wesley, 2002.

[8] M. Harris, "Optimizing parallel reduction in CUDA, " Nvidia, Tech. Rep., 2007.

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

[10] G. E. Blelloch, "Scans as primitive parallel operations," IEEE Transactions on Computers, v.38 n.11, p.1526-1538, November 1989.

[11] 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.

[12] A. Buluc, "Linear algebraic primitives for parallel computing on large graphs," Ph.D. thesis, University of California, Santa Barbara, 2010.

[13] M. Harris, J. Owens, S. Sengupta, Y. Zhang and A. Davidson. CUDPP: CUDA Data Parallel Primitives Library. http://www.gpgpu.org/developer/cudpp/, 2011.

[14] Thrust. Thrust homepage. http://code.google.com/p/thrust/. 2011.

[15] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, Introduction to Algorithms, MIT press, 2001.

[16] V. Vineet, P. Harish, and P. J. Narayanan, "Large graph algorithms for massively multithreaded architectures," Tech. Rep., 2009.

[17] J. M. Pedro, T. Robert and G. Antonio. "CUDA solutions for the SSSP problem," Proc of the 9th international conference on computational science, 2009, pp. 904–913.

[18] A. Leist, D. P. Playne and K. A. Hawick. "Exploiting graphical processing units for data-parallel scientific applications".Tech. Rep., December, 2009.

[19] D. Roger, U. Assarsson and N. Holzschuch. "Efficient stream reduction on the GPU". In Workshop on General Purpose Processing on Graphics Processing Units, 2007.

[20] D. A. Bader and K. Madduri, "GTgraph: a synthetic graph generator suite," Tech. Rep., 2006.

Wei Wang is born in TaiAn ShanDong China, born in 1983. Between the year 2002 and 2006, get bachelor's degree in computer science and technology in Zhengzhou Information Science and Technology Institute which is in Zhengzhou Henan China. He is a Master candidate in computer software and theory in Zhengzhou Information Science and Technology Institute, and is expected to graduate in June 2011. His research interests include distributed system, parallel computing, and general purpose computing on GPU.

Yongzhong Huang is born in 1968, ph. D. He is currently a doctor supervisor in distributed system in Zhengzhou Information Science and Technology Institute which is in Zhengzhou Henan China. His research interests include distributed system, parallel computing.

Shaozhong Guo is born in HeFei AnHui China, born in 1964. She is currently a master supervisor in computer software and theory in Zhengzhou Information Science and Technology Institute which is in Zhengzhou Henan China. His research interests include distributed system, parallel computing, and DBMS system.

