ELectronics EXpress

LETTER

IEICE Electronics Express, Vol.13, No.6, 1–10

Fast bidirectional shortest path on GPU

Lalinthip Tangjittaweechai[1a)], **Mongkol Ekpanyapong**[1b)], **Thaisiri Watewai**[2], **Krit Athikulwongse**[3], **Sung Kyu Lim**[4], **and Adriano Tavares**[5]

[1] *Microelectronics and Embedded Systems, Asian Institute of Technology, Thailand*

[2] *Chulalongkorn Business School, Chulalongkorn University, Thailand*

[3] *National Electronics and Computer Technology Center, Thailand*

[4] *School of Electrical and Computer Engineering, Georgia Institute of Technology, Georgia, USA*

[5] *Centre Algoritmi, Department of Industrial Electronics, University of Minho, Portugal*

a) *st113130@ait.asia*

b) *mongkol@ait.asia*

Abstract: The bidirectional shortest path problem has important applications in VLSI floor planning and other areas. We introduce a new algorithm to solve bidirectional shortest path problems using parallel architectures provided by general purpose computing on graphics processing units. The algorithm performs parallel searches from the source and sink using Dijkstra's classic approach modified with pruning and early termination. We achieve substantial speedup over a parallel method that performs a single parallel search on the GPGPU from the source to all other nodes but early terminates when the shortest path to the specified target node is found. Experimental results demonstrate a speedup of nearly 2× over the parallel method that performs a parallel search from the source with early termination on the GPGPU.

Keywords: shortest path algorithm, single pair, GPGPU, parallel, bidirectional search

Classification: Integrated circuits

References

[1] P. Hart, N. Nilsson and B. Raphael: IEEE Trans. Syst. Sci. Cybernetics **4** (1968) 100. DOI:10.1109/TSSC.1968.300136

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein: *Introduction to Algorithms* (MIT Press and McGraw–Hill) 2nd ed. 595.

[3] J. Pearl: *Heuristics: Intelligent Search Strategies for Computer Problem Solving* (Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1984).

[4] G. Nannicini, D. Delling, L. Liberti and D. Schultes: WEA'08 (2008) 334. DOI:10.1007/978-3-540-68552-4_25

[5] M. N. Rice and V. J. Tsotras: SoCS (2012).

[6] L. Sint and D. D. Champeaux: JACM **24** (1977) 177. DOI:10.1145/322003.322004

© IEICE 2016

DOI: 10.1587/elex.13.20160036
Received January 15, 2016
Accepted February 5, 2016
Publicized March 3, 2016
Copyedited March 25, 2016

[7] D. D. Champeaux: JACM **30** (1983) 22. DOI:10.1145/322358.322360
[8] H. Kaindl and G. Kainz: J. Artif. Intell. Res. **7** [1] (1997) 283.
[9] P. Harish and P. J. Narayanan: HiPC (2007) 197. DOI:10.1007/978-3-540-77220-0_21
[10] P. Harish, V. Vineet and P. J. Narayanan: Technical report of International Institute of Information Technology Hyderabad, INDIA, Tech. Rep. III/TR/2009/74 (2009).
[11] G. Vaira and O. Kurasova: DB&IS (2011) 422.
[12] Dimacs: http://www.dis.uniroma1.it/challenge9/download.shtml.
[13] SSCA2: http://www.cse.psu.edu/~kxm85/software/GTgraph/.

1 Introduction

Shortest and longest path algorithms are crucial to applications spanning several areas in computing, particularly in VLSI floorplanning, where the longest path is, for instance, used to calculate the total area required for a circuit. Furthermore, techniques for timing-aware global placement and detailed placement take the longest path length as a parameter in order to identify a feasible clock period. VLSI routing algorithms perform extensive searches for shortest route paths without violating routing constraints. With millions of nets to be routed, or millions of gates to be placed, shortest path algorithms must be run on a massive amount of data, requiring substantial runtime resources in terms of time and memory. For example, during placement optimization, the shortest/longest path algorithm may be executed thousands or millions of times while the solution space is being explored before a good solution is obtained. Hence, a substantial amount of the time required in many VLSI optimizations is devoted to this shortest/longest path calculation. The most common operation is to find the shortest path from a specific source node to a specific target node.

To solve the single pair shortest path problem, two main algorithms are widely accepted. The first is the A^* search algorithm [1], and the second is Dijkstra's algorithm [2]. A^* search is a heuristic method [3]. Bidirectional A^* search algorithms were proposed by Nannicini et al. [4] and Rice et al. [5]. Most of bidirectional searches [6, 7, 8] use a heuristic to find the solution in a short amount of time. Dijkstra's algorithm is a path finding algorithm for computing a shortest path from a source to all other nodes in a graph. Dijkstra's algorithm can also be used to find the shortest path between a source and a target by stopping the algorithm when the target is reached. A parallel implementation of Dijkstra's algorithm for graphics processing units (GPUs) targeted for the single source, multiple targets shortest path problem was proposed by Harish, Vineet, and Narayanan [9, 10]. The idea is to take advantage of the hundreds or thousands of graphics cores on the GPU to parallelize the solution space exploration and consequently speed up the search for the optimal solution. Vaira and Kurasova [11] propose a parallel single pair shortest path algorithm that finds the exact shortest path by deploying a bidirectional version of Dijkstra's algorithm on a multi-core CPU. They report that their algorithm is almost three times faster than the sequential implementation of Dijkstra's algorithm. Given a graph, their algorithm

© IEICE 2016
DOI: 10.1587/elex.13.20160036
Received January 15, 2016
Accepted February 5, 2016
Publicized March 3, 2016
Copyedited March 25, 2016

deploys one thread to span from the source node and another thread to concurrently span from the target node. The algorithm terminates once those two threads meet at an intermediate node. This approach is valid because the shortest path property guarantees that the shortest path from the source node to the target node is on the shortest path from the source node to some intermediate node and on the shortest path from that intermediate node to the target node. However, this two-thread approach is limited as it cannot be easily extended to utilize the hundreds or thousands of threads available in a GPU. In addition, the straightforward termination condition is no longer valid for multi-threaded (more than two-thread) scenarios.

The parallel implementation of Dijkstra's algorithm proposed by Harish, Vineet, and Narayanan [9, 10] is targeted for the single source, multiple targets shortest path problem. In this paper, a parallel version of Dijkstra's algorithm for the single-pair shortest path non-negative weight problem is proposed and deployed on a graphics processing unit (GPU). The approach can be extended to the longest path problem as well. In our implementation, a bidirectional search is deployed by initiating search from both source and target nodes concurrently. This addresses the issue of idle threads in the GPU that normally occurs at the beginning of the computation, and it is also useful for graphs with linear structure. The contributions of this paper are as follows. First, we propose a bidirectional parallel search algorithm for GPGPUs. Second, we introduce a termination condition for multi-threaded parallel search. Finally, we describe a search pruning technique. Experimental results indicate a 2× speedup over the parallel method that performs a parallel search from the source with early termination on the GPGPU. The performance speedup comes from the parallelism exploited by traversing the graph from both directions concurrently, along with early termination and search pruning.

2 Methodology

2.1 Algorithm overview

Dijkstra's algorithm is iterative. In the first iteration, the source node is associated with a zero-weight cost. On each iteration, the shortest path calculation expands from the nodes already considered to their neighbors. In this paper, to better utilize the available threads in a GPU, we initiate concurrent searches from both the source node and the target node simultaneously. We define the *forward shortest path cost* as the cost computed from the source node to each of the other nodes as in the original Dijkstra algorithm. We also introduce *the backward shortest path cost*, which is the shortest path cost computed backward from the target node to each of the other nodes. On the first iteration, the forward shortest path cost of the source node and the backward shortest path cost of the target node are associated with zero-cost values. All forward and backward shortest path costs expanded from the source node and the target node are concurrently computed as shown in Fig. 1. We define S as the set of nodes expanded from the source side (shown in red in Fig. 1) and T as the set of nodes expanded from the target side (shown in blue in Fig. 1). Finally, set $P = S \cap T$ contains the intermediate nodes (shown in purple in Fig. 1) at which the expansions from S and T meet or converge. Each node in P has both a

© IEICE 2016
DOI: 10.1587/elex.13.20160036
Received January 15, 2016
Accepted February 5, 2016
Publicized March 3, 2016
Copyedited March 25, 2016

forward and a backward shortest path cost. Each update for a node in P computes the sum of the forward shortest path cost and the backward shortest path cost to that node. The element of P with the smallest summed cost gives the current shortest path cost from the source to the target in the graph. On each iteration, we expand the forward and backward shortest path costs from each of the nodes in P to their neighbors. We then recompute the minimum forward shortest path cost and minimum backward shortest path cost for all active nodes in S and T. By active nodes, we mean the nodes whose forward/backward shortest path costs are updated in a given iteration, regardless of which set they belong to. The algorithm terminates early if the current shortest path cost from the source to the target is less than or equal to the sum of the minimum forward shortest path cost among the active nodes in S and the minimum backward shortest path cost among the active nodes in T. The monotonic property of path costs in the non-negative weighted graph dictates that the algorithm will not be able to find a lower cost path from the source to the target than the current shortest path cost unless there is an active node with forward or backward shortest path cost lower than the current shortest path cost.

Search pruning can further reduce the required computation time. We can stop search expansion from an active node if the value of the forward or the backward shortest path costs at that node exceeds the current shortest path cost from the source to the target. One example is shown in Fig. 2. In this example, the expansion fronts meet in the second iteration at node b, and the current shortest path cost from the source node to the target node for this iteration is set to 18. The current shortest path cost is compared with the forward/backward shortest path costs for all the other active nodes. The monotonic property of the path cost can be used to prune impossible active paths. For instance, suppose node d is expanded from the source in the third iteration, and its forward shortest path cost is equal to 20. We can stop expanding the search from node d, since the cost of any shortest path expanded from node d must be 20 or higher, which is greater than the current shortest path cost (18). Note that although we can conclude that node d's current shortest path cannot be on the shortest path from the source to the target and can prune node d as an active node, later, we may further expand search from node d to its neighbors if a new path updates node d with a lower forward cost. Similarly, for the backward search, we can stop expanding from node e (backward path cost $45 \geq 18$) unless and until some later update to its shortest path cost.

2.2 Algorithm details

The main algorithm, specified in Algorithm 1, operates on the CPU. The algorithm accepts a positive weighted graph G (described by vertices V, edges E, and weight values W), a source node s, and a target node t as inputs. In the algorithm, lines 1–4 perform initialization. The variable *completed* is used in the termination condition. The variable *CurrCost* records the current smallest cost of the path from the source to the target in the graph. The variables *MinF* and *MinB* record the minimum forward cost from the source node over all active nodes and the minimum backward cost to the target node over all active nodes. These values are updated on each iteration. *LatestMinF* and *LatestMinB* are temporary variables for *MinF* and *MinB*,

© IEICE 2016
DOI: 10.1587/elex.13.20160036
Received January 15, 2016
Accepted February 5, 2016
Publicized March 3, 2016
Copyedited March 25, 2016

ELectronics EXpress

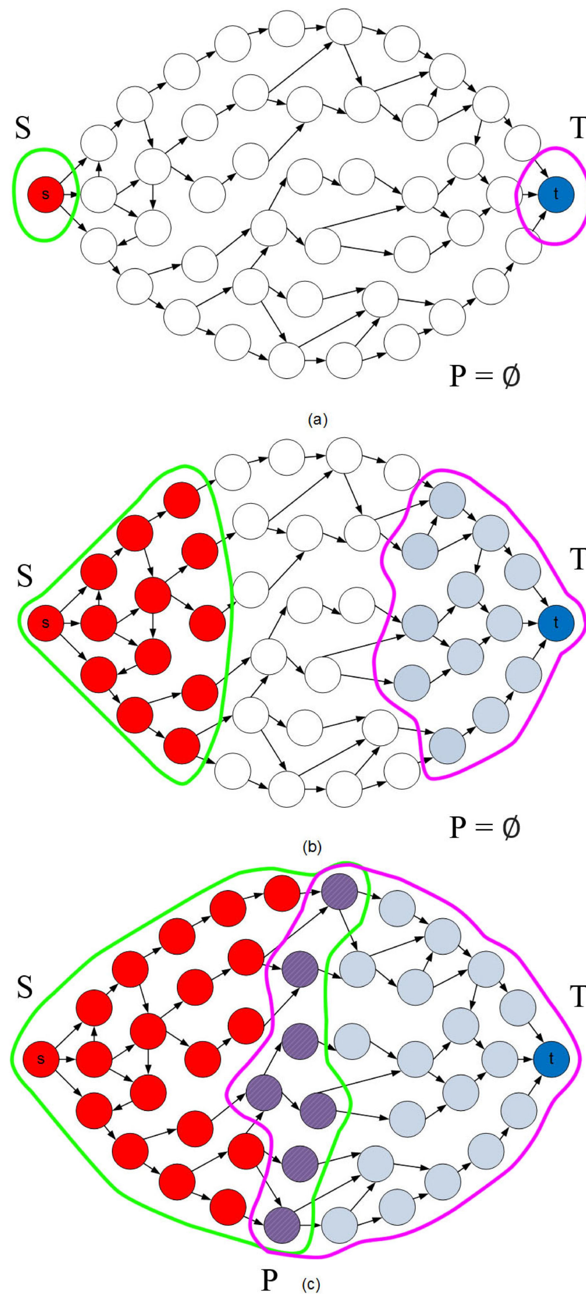

Fig. 1. Graph search expansion. (a) Initial state. (b) Expansion after a few iterations. (c) Overlapping expansions.

respectively. Finally, the variable *iter* counts the number of iterations. In lines 5–6, the algorithm invokes the *Mark_thread* and *Expand_nodes* GPU kernel functions. Lines 7–15 check the early termination condition. The algorithm terminates early when $MinF + MinB \geq CurrCost$, as shown in lines 8–10. Note that *MinF* and *MinB* are calculated for every node whose forward or backward cost is updated in a given iteration, not only the nodes in *P*.

© IEICE 2016
DOI: 10.1587/elex.13.20160036
Received January 15, 2016
Accepted February 5, 2016
Publicized March 3, 2016
Copyedited March 25, 2016

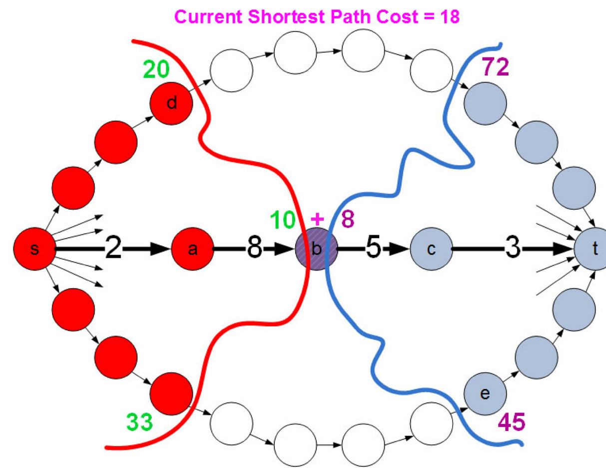

Fig. 2. Search pruning to reduce the required computation time. Node a is expanded from the source, while node c is expanded from the target in iteration 1. On the second iteration, node b is expanded from both the source and the target. Nodes d and e are expanded from the source and the target, respectively, in the third iteration. As nodes d and e currently have shortest forward/backward path costs higher than the current shortest path cost of 18, they may be pruned from further consideration

Algorithm 1 Parallel_Shortest_Path $(g(V, E, W), s, t)$

```
1: Call Init_variables kernel
2: LatestMinF = INT_MAX, LatestMinB = INT_MAX
3: do
4:   completed = 1, MinF = INT_MAX, MinB = INT_MAX,
5:   Call Mark_thread kernel
6:   Call Expand_nodes kernel
7:   if MinF != INT_MAX and MinB != INT_MAX then
8:     if MinF + MinB >= CurrCost then
9:       Call Substitute_CurrCost kernel
10:      completed = 1
11:    else
12:      LatestMinF = MinF
13:      LatestMinB = MinB
14:    endif
15:  endif
16: while completed = 0
17: return CurrCost
```

On each iteration, the search expands from each active node to neighboring nodes. Before the expansion, the *Mark_thread* GPU kernel, as shown in Algorithm 2, computes a node status indicating which sets (S, T, P, or none) each node belongs to. The node status is based on the graph expansion depending on whether the search comes from the source side, target side, or both, as shown in lines 2–11. In this GPU kernel function, *Llabels* keeps the current forward shortest path cost expanded from the source to all other nodes in the graph (the array's size is equal to

© IEICE 2016
DOI: 10.1587/elex.13.20160036
Received January 15, 2016
Accepted February 5, 2016
Publicized March 3, 2016
Copyedited March 25, 2016

IEICE Electronics Express, Vol.13, No.6, 1–10

the number of nodes in the graph). Similarly, *Rlabels* keeps the current backward shortest path cost expanded from the target to all other nodes in the graph (the array's size is also equal to the number of nodes in the graph). A mark array *M* stores the node statuses. For the threads that are active from the source side, the algorithm checks whether there are any updates from the source, or equivalently, whether the forward shortest path cost is less than its initial value (i.e., Llabels[idx] != *INT_MAX*). If this is the case, the current forward shortest path cost (*Llabels*[*idx*]) must also be less than *CurrCost*, as previously explained regarding the search pruning technique. In lines 2, 4, and 7, we compare *Llabels*[*idx*] or *Rlabels*[*idx*] with *CurrCost* for search pruning purposes. If the new forward or backward shortest path cost of the node is higher than the current shortest path cost, further expansion from that node is not necessary. In addition, to restrict calculation to the active nodes, the comparisons of *Llabels*[*idx*] with *LatestMinF* and *Rlabels*[*idx*] with *LatestMinB* are added. Since we do not allow negative weights in the graph, the current forward/backward shortest path costs must be greater than the minimum forward/backward shortest path costs from the previous iteration, respectively. Paths that are expanded from both source and target sides update *CurrCost* if *Llabels*[*idx*] + *Rlabels*[*idx*] is a new minimum (see lines 12–16).

Algorithm 2 Mark_thread (*Llabels*, *Rlabels*, *M*, *CurrCost*, *LatestMinF*, *LatestMinB*) Note that this kernel is executed in parallel for every node.

```
1: int idx = getThreadID
2: if Llabels[idx] < CurrCost and Llabels[idx] >= LatestMinF then
3:   M[idx] = 'S'
4:   if Rlabels[idx] < CurrCost and Rlabels[idx] >= LatestMinB then
5:     M[idx] = 'P'
6:   endif
7: else if Rlabels[idx] < CurrCost and Rlabels[idx] >= LatestMinB then
8:   M[idx] = 'T'
9: else
10:  M[idx] = ' '
11: endif
12: if Llabels[idx] != INT_MAX and Rlabels[idx] != INT_MAX then
13:   if CurrCost > Llabels[idx] + Rlabels[idx] then
14:     atomicMin(&CurrCost, Llabels[idx] + Rlabels[idx])
15:   endif
16: endif
```

After the update of the node status by the *Mark_thread* GPU kernel, active threads expand the forward/backward shortest path cost of the nodes neighboring active nodes according to the *Expand_nodes* GPU kernel, shown in Algorithm 3. Threads in *S* and *P* expand to outgoing neighbors as shown in lines 2–12. Each thread calculates the new costs for outgoing neighbors one by one by summing the current shortest path cost of that node (*Llabels*[*idx*]) with the weight of the edge

© IEICE 2016
DOI: 10.1587/elex.13.20160036
Received January 15, 2016
Accepted February 5, 2016
Publicized March 3, 2016
Copyedited March 25, 2016

IEICE Electronics Express, Vol.13, No.6, 1–10

connecting node *idx* with the neighbor. If the new cost is less than the current shortest path costs for any of the neighbors, the thread updates the current shortest path costs in *Llabels* for each of its neighbors using an atomic operation (see line 5). If there is any update in the graph, the *completed* variable is set to 0 (line 9), forcing the expansion to continue in the next iteration. If there is no update from any thread, the main algorithm can terminate the do/while loop (line 16 in Algorithm 1).

Algorithm 3 Expand_nodes ($g(V, E, W)$, *completed*, *MinF*, *MinB*, *Llabels*, *Rlabels*, *M*, *CurrCost*, *iter*)

```
1: int idx = getThreadID
2: if M[idx] = 'S' or M[idx] = 'P' then
3:   for all outgoing neighbors nidx of idx do
4:     if Llabels[nidx] > Llabels[idx] + W [idx, nidx] then
5:       atomicMin(&Llabels[nidx], Llabels[idx] + W[idx, nidx])
6:       if CurrCost < INT_MAX and iter % ITERATION = 0 then
7:         atomicMin(&MinF, Llabels[nidx]);
8:       endif
9:       completed = 0
10:    end if
11:  end for
12: endif
13: if M[idx] = 'T' or M[idx] = 'P' then
14:   for all incoming neighbors nidx of idx do
15:     if Rlabels[nidx] > Rlabels[idx] + W[nidx, idx] then
16:       atomicMin(&Rlabels[nidx], Rlabels[idx] + W[nidx, idx])
17:       if CurrCost < INT_MAX and iter % ITERATION = 0 then
18:         atomicMin(&MinB, Rlabels[nidx]);
19:       endif
20:       completed = 0
21:     end if
22:   end for
23: endif
```

Along the way, each thread updates *MinF* (the minimum shortest path cost updated among threads from the source side in that iteration, lines 6–8). *MinF* is used in the check for the early termination condition as shown in lines 7–10 in Algorithm 1. Maintaining the value of *MinF* and *MinB* on every iteration is compute intensive since it requires an atomic operation, and these variables are shared among all nodes. Therefore, our algorithm only updates *MinF* and *MinB* for nodes in P, where the expansions from source and target have met, i.e., when *CurrCost* is less than *INT_MAX* (line 6). Another speedup technique is to update the *MinF* and *MinB* variable at constant intervals (when *iter* % *ITERATION* = 0, where *ITERATION* is a constant value). The backward expansion (line 13–23) is similar to the forward expansion, except that the direction of the expansion comes from the target node instead of the source node.

© IEICE 2016
DOI: 10.1587/elex.13.20160036
Received January 15, 2016
Accepted February 5, 2016
Publicized March 3, 2016
Copyedited March 25, 2016

IEICE Electronics Express, Vol.13, No.6, 1–10

3 Experimental results

To evaluate our method, we used a GeForce GTX 480 with 480 CUDA cores on an Intel E7500 computer as our experimental testbed. First, we modified Harish's original parallel shortest path algorithm for GPGPU [9, 10] (named as "Harish") to include the early termination condition (named Harish with Early Termination or HET). The original Harish algorithm is targeted for the single source, multiple targets shortest path problem. The HET algorithm is the same as the original Harish algorithm except that HET algorithm terminates once the shortest path cost from the source node to the target node has been found. The modification transforms Harish into a GPGPU single-pair shortest path algorithm. To the best of our knowledge, there has been no report of any other GPGPU single pair shortest path algorithm based on Dijkstra's method in the literature. We named our implementation Parallel Source and Target Search (PSTS). We compute the runtime speedup of PSTS over HET. Note that we also report the original Harish runtime. For the results reported here, we set ITERATION = 100 for PSTS. The study was performed on two sets of benchmarks: the 9th Dimacs [12] and SSCA#2 [13] benchmarks. In all runs, benchmarks, node 1 was chosen as the source, and node N was chosen as the target, where N is the number of nodes in the graph. The execution times of Harish, HET, and PSTS on the 9th Dimacs benchmark with source = 1, target = N are shown in Table I. The number of nodes and edges in each benchmark graph are also reported as #Nodes and #Edges in the table. HET/PSTS shows the speedup of PSTS over HET. From the results, PSTS outperforms HET by 2.23×. The observed speedup is greater than 2.0 because of the search pruning in PSTS.

Table I. Runtime comparison on the 9th Dimacs benchmarks (source = 1, target = N).

Name	# Nodes	# Edges	Harish	HET	PSTS	HET/PSTS
NY	264346	733846	295	68	44	1.55
BAY	321270	800172	271	257	191	1.35
COL	435666	1057066	507	390	215	1.81
FLA	1070376	2712798	3227	614	805	0.76
NW	1207945	2840208	3604	1684	874	1.93
NE	1524453	3897636	2382	1214	379	3.20
CAL	1890815	4657742	5196	1181	274	4.31
LKS	2758119	6885658	13379	488	298	1.64
E	3598623	8778114	11818	6910	2912	2.37
W	6262104	15248146	19124	9931	3091	3.21
CTR	14081816	34292496	67074	19327	7993	2.42
					avg.	2.23

The second experiment uses the SSCA#2 graph model generated using the Georgia Tech graph generator tool [13]. As with Dimacs, node 1 was chosen as the source, and node N was chosen as the target. Table II shows the results with ITERATION = 80 for the PSTS case. In this benchmark, PSTS outperforms HET

© IEICE 2016
DOI: 10.1587/elex.13.20160036
Received January 15, 2016
Accepted February 5, 2016
Publicized March 3, 2016
Copyedited March 25, 2016

IEICE Electronics Express, Vol.13, No.6, 1–10

by 1.78× on average. We noticed that for some small graphs, PSTS's performance gain decreases. Factors such as the data structure and layout can affect execution time. These effects on execution time are compounded by the number of nodes to be updated in each step in the wave front set, which influences the current minimum cost calculation time.

Table II. Runtime comparison on SSCA#2 with, weight range 1–100.

# Nodes	Degree	Harish	HET	PSTS	HET/PSTS
65536	13	137	86	43	2.00
131072	17	369	73	73	1.00
262144	21	1516	227	113	2.01
524288	27	4561	4231	1538	2.75
1048576	34	15894	4220	3625	1.16
				Avg.	1.78

4 Conclusion

In this paper, we propose a parallel shortest path algorithm based on Dijkstra's method using a bidirectional search technique deployed on the GPGPU. Our algorithm finds the exact shortest path cost by traversing nodes from the source and the target in parallel, while applying early termination and search pruning strategies reduce search time. Experimental results obtained when setting source = 1 and target = N in the 9th Dimacs and SSCA#2 benchmarks show that our implementation can, on average, provide a speedup of 2.23× and 1.78×, respectively, over a parallel method that performs a single parallel search on the GPGPU from the source to all other nodes but early terminates when the shortest path to the specified target node is found.

Acknowledgments

This research was supported by Thailand Research Fund, the Office of Higher Education, and the Asian Institute of Technology under contract number MRG5380293.

© IEICE 2016
DOI: 10.1587/elex.13.20160036
Received January 15, 2016
Accepted February 5, 2016
Publicized March 3, 2016
Copyedited March 25, 2016