Load-Balanced Breadth-First Search on GPUs

Zhe Zhu, Jianjun Li, and Guohui Li

School of Computer Science & Technology,
Huazhong University of Science & Technology, China
luokezhu@gmail.com, Jianjunli,Guohuili@hust.edu.cn

Abstract. Breadth-first search (BFS) is widely used in web link and social network analysis as well as other fields. The Graphics Processing Unit (GPU) has been demonstrated to have great potential in accelerating graph algorithms through parallel processing. However, BFS is difficult to parallelize efficiently due to the irregular workload distribution, leading to load imbalance between threads. Previous work has proposed several strategies to alleviate the load imbalance but none of them solves this issue in general.

This paper presents a new GPU BFS algorithm that focuses on full load balance. Each BFS iteration is decoupled into two phases: work redistribution and neighbor gathering. Work redistribution phase reorganizes the irregular workloads in order for the neighbor gathering phase to visit the vertices in a load-balanced way. The evaluation results show that the proposed approach achieves speedups of up to 39x and 1.42x over CPU sequential implementation and state-of-the-art GPU implementation respectively.

Keywords: Breadth-first search, GPU, load balance, graph algorithms, parallel algorithms

1 Introduction

Graph algorithms are becoming increasingly important, with applications ranging from web link analysis to computer-aided design to machine learning. Breadth-first search (BFS) is an important low-level operation that serves as a fundamental building block for more complicated graph algorithms. Thus efficient parallelization of BFS has gained much attention.

Unfortunately, exploiting the nested parallelism in BFS is challenging. Assigning the workloads to each thread evenly is non-trivial because the work distribution patterns are determined by the structure of the input graph.

Modern GPUs have become popular general computing devices due to their high memory and computational throughput, low costs and power efficiency. However, accelerating BFS on GPUs requires much more attention. The wide SIMD architecture of GPUs is particularly sensitive to load imbalance [3]. Inadequate handling of this issue can lead to a significant performance hit.

Prior work has proposed several parallelization approaches [7,8,10,11]. They mainly rely on overlapped execution of massive amount of threads, local reorga-

F. Li et al. (Eds.): WAIM 2014, LNCS 8485, pp. 435–447, 2014.
© Springer International Publishing Switzerland 2014

nization of workloads and work stealing to limit load imbalance to some extent. However, none of them eliminates this issue in general.

In this paper, we present a load-balanced GPU BFS algorithm, which decouples each BFS iteration into two phases: work redistribution and neighbor gathering. Work redistribution phase serves as a preprocessing operation, employing a parallel expansion to reorganize the nested and irregular workloads of a BFS iteration. Neighbor gathering phase then subsequently assigns the workloads to threads uniformly and visits each neighbor in a load-balanced way.

Specifically, we make the following contributions:

- We propose a load-balanced GPU BFS algorithm. To the best of our knowledge, ours is the first BFS implementation on GPUs that achieves fully load-balanced neighbor gathering.
- We analyze the coupling possibilities between different phases of the algorithm for optimal performance. Coupling separate procedures into one kernel reduces I/O overhead but may amplify load imbalance. We show that a hybrid coupling strategy has the best performance.
- Our approach delivers great performance on a wide diversity of real-world graphs, achieving speedups of up to 39x and 1.42x over CPU sequential implementation and state-of-the-art GPU implementation, respectively.

2 Background and Motivation

In this section, we first introduce some unique properties of GPU architecture. Then we review existing BFS algorithms on GPUs and motivate our approach.

2.1 Modern GPU Architecture

In order to deliver high computational throughput, modern GPUs adopt a wide SIMD architecture[3], meaning threads within a *warp* execute the same instructions synchronously. Control flow divergence among these threads will result in serialization of different execution paths. Warps are grouped into *cooperative thread arrays* (or CTAs). Threads within a CTA can communicate through a local *shared memory*, and GPU hardware treats the CTA as the unit of scheduling. A program running on the GPU is called a *kernel*.

This hierarchical model introduces several types of workload imbalance. The SIMD execution within a warp will cause thread load imbalance and underutilization if control flow diverges. Within a CTA, the warp with the highest workload will cause other completed warps to sit idle and prevent the completion of the CTA, which in turn will prevent other CTAs in the wait queue from being scheduled. Likewise, few CTAs taking too much time to complete can extend the completion time of the kernel. Figure 1 illustrates these three types of workload imbalance.

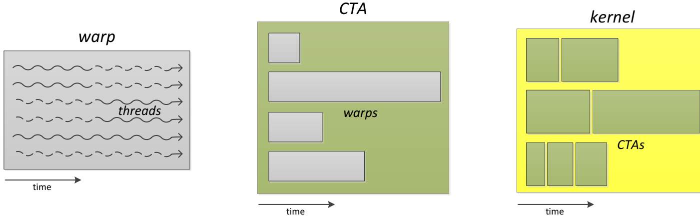

Fig. 1. (Left to right) thread imbalance, warp imbalance and CTA imbalance

Algorithm 1. Linear-work parallel BFS

Input: v_0 , input queue inQ and output queue $outQ$
Output: Array $dist[0...n-1]$ holding the distance from s to each vertex

```
1  initialize all elements in dist[0...n − 1] to ∞ and empty inQ
2  dist[v0] ← 0, iteration ← 0
3  inQ.Enqueue(v0)
4  while inQ not empty do
5      empty outQ
6      foreach v ∈ inQ in parallel do
7          foreach neighbor of v in parallel do
8              if StatusLookup(neighbor) = valid ∩ dist[neighbor] = ∞ then
9                  dist[neighbor] ← iteration + 1
10                 outQ.Enqueue(neighbor)
11     iteration + +
12     switch inQ and outQ
```

2.2 Existing BFS Algorithms on GPUs

Given a source vertex v_0, the BFS process traverses the vertices in breath-first order and label each vertex with its distance from v_0. Other variants of BFS may record other attributes such as the parent of each vertex.

Earlier GPU BFS research mainly focuses on work-inefficient parallelization [7,8] which has quadratic work complexity ($O(n^2 + m)$ or $O(mn)$, n and m represent the vertex and edge numbers, respectively). Luo et al. [10] present the first linear work BFS ($O(m + n)$) and achieve much better performance. In this paper, we will focus on work-efficient algorithms.

The skeleton of the linear-work BFS algorithm on the GPU is similar to the standard serial BFS on the CPU [9], which is listed as Algorithm 1. On each iteration, vertices are taken out of the input queue, and their neighbors are visited and inserted into the output queue for next iteration. However, there are two main differences between CPU and GPU BFS algorithms, which are also the main challenges of GPU BFS:

Parallel neighbor gathering. The neighbor gathering process read in all the neighbors of the input vertices. Both the vertices in the input queue and

all the neighbors of a vertex are independent of each other so there is sufficient parallelism to exploit. However this nested and irregular loop structure makes the parallelization difficult. A poor mapping strategy between threads and vertices will suffer from severe workload imbalance.

Status lookup. When inspecting the neighbors, they need to be checked to see if they have already been visited. This often results in many costly random accesses to the *dist* array. An effective optimization is to add a status lookup process and use a bitmap array to check the status, leading to reduced global memory overhead and improved cache hit rate.

We will focus on the neighbor gathering process, as it is where load imbalance happens and can easily become the bottleneck of the whole BFS algorithm.

The simplest strategy is to map each thread to a vertex in the input queue, having each thread inspect the neighbors of the assigned vertex serially. Harish et al. [7] and Luo et al. [10] use this strategy. It only exploits the parallelism of the outer loop, and can lead to severe thread imbalance within a warp for graphs having non-uniform degree distributions. Moreover, the arbitrary memory accesses from each thread result in terrible coalescing too.

A better strategy is to map a whole warp or CTA to a vertex in the input queue, which is adopted by Hong et al. [8] and Merrill et al. [11]. In this way, the whole warp or CTA gather the adjacency list of the vertex in parallel. This approach provides good thread balance for vertices having large numbers of neighbors. However for vertices with the adjacency list sizes smaller than the warp/CTA width, some threads in the warp/CTA will go unused, imposing underutilization of the warp/CTA. Furthermore, there may exist warp imbalance or CTA imbalance if the adjacency list sizes vary significantly.

Another scan-based strategy introduced by Merrill et al. [11] maps a CTA to a certain number of vertices in the input queue. The CTA first constructs a shared array of neighbor locations corresponding to the concatenation of the assigned adjacency lists. Then the CTA reads in the locations from the shared array and gather the neighbors iteratively. Compared to the CTA mapping approach, this strategy solves the CTA underutilization problem at the cost of additional concatenating operations, which is efficient for vertices having small sizes of adjacency lists. Since each thread constructs its part of the shared array serially and its workload is proportional to the size of the assigned adjacency list, large adjacency lists can impose thread imbalance and inefficiency.

Each of the above mapping strategies is suitable for certain types of graphs. Merrill et al. [11] therefore adopt a hybrid approach. For vertices having more neighbors than the CTA width, CTA mapping is applied. For vertices having the number of neighbors smaller than the CTA width but larger than the warp width, warp mapping is applied. Finally, scan-based mapping is performed on the remaining vertices. This hybrid approach limits thread imbalance and warp imbalance, which is the current state of the art on GPU BFS.

Other works explore general graph algorithms on GPUs [12,16]. They focuses on flexibility and clarity but lacks specific optimization. Their BFS implementations are inefficient.

2.3 Motivation of This Work

All the existing parallelization strategies suffer from load imbalance issues. They cannot achieve consistent performance over various graphs. The hybrid CTA+ warp+scan approach has been shown to perform efficiently. However, this solution is not good enough for the following reasons:

(1) It does not solve the load imbalance problem in general, but only limits thread imbalance and warp imbalance to some extent. CTA imbalance is not addressed. Instead, it relies on work stealing to alleviate CTA imbalance.
(2) The neighbor gathering and status lookup process must be put in separate kernels for optimal performance because fusing these two processes would amplify the CTA imbalance. This leads to additional global data movement.
(3) It is complicated and unintuitive. Work partitioning and neighbor gathering logic are mixed up, resulting in an algorithm difficult to understand.

To address these problems, we present a load-balanced BFS algorithm. It is decoupled into two phases: work redistribution and neighbor gathering. Moreover, in the absence of CTA imbalance we get to fuse neighbor gathering and status lookup into one kernel and further improve performance.

3 Parallel Expansion

The nested and highly irregular parallelism shown in BFS, together with the static thread creation mechanism of GPUs, make a balanced work partitioning very difficult. The latest NVIDIA GPU architecture GK110 supports *dynamic parallelism* [3] in order to ease this problem, which enables the GPU kernel to launch other kernels itself. However, this does not solve this issue in general because the number of newly allocated threads does not match the problem size very well. Vertices with few neighbors would be provisioned entire CTAs, leading to underutilization. To address this problem, we preprocess the input to reorganize the workloads, eliminating the nested parallelism. In this section, we introduce the *expand* operation which is the basis of the workload reorganization, and the parallelization of *expand*.

3.1 The *expand* Operation

To get rid of the nested workload structure, we pack the neighbor gathering work produced by each input vertex together into a single sequence, with each element of the sequence representing the gathering address. In this way, threads can be uniformly mapped to this sequence and do the neighbor gathering in a load-balanced fashion.

In order to generate this sequence, we first define a basic operation. As illustrated in Fig. 2, taking the degree of each vertex in the queue as input, this operation outputs an array whose length is equal to the total number of neighbors to be produced. Each element in the array represents the index of the vertex

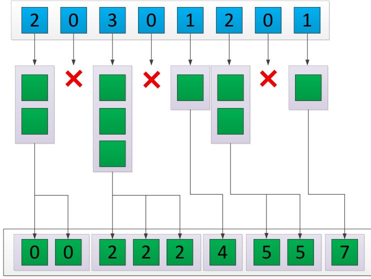

Fig. 2. The *expand* operation

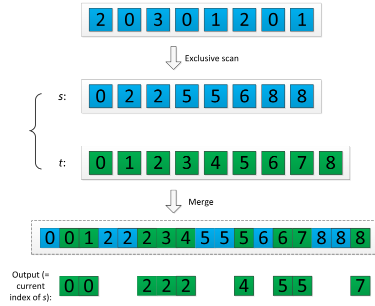

Fig. 3. Converting *expand* into *merge* operation

in the queue that will produce it so in the subsequent gathering phase we can find that vertex and its neighbors.

We will call this operation *expand*, which is a useful pattern in data-parallel algorithms. Using *expand*, the nested loop structure is reorganized and flattened, which is the key to achieving load balance. Obviously, serial implementation of *expand* has $O(m)$ time complexity, thus efficient parallelization of the *expand* operation is the basis of high performance of the whole BFS algorithm.

3.2 Parallelization of *expand*

The *expand* operation can actually be converted to a merging of two sorted arrays. As demonstrated in Fig. 3, we first run an exclusive scan [13] on the inputs, obtaining the result array s and the sum *total*. We then construct an array t of length *total* filled with $[0...total-1]$ and merge s and t. The difference compared to a normal *merge* is that we only output *total* elements, and the value of each output element equals to the current index of array s. In practice, the array t is not necessary because the indices and values are the same. Algorithm 2 shows the sequential implementation of the *expand* operation.

The parallelization of the *merge* operation has been studied for decades [15,6]. Basically, the input sequences are partitioned into non-overlapping segments, and the independent pairs of segments are merged in parallel. Odeh et al. [14] present a *merge path* algorithm that achieves a perfectly load-balanced partitioning. As depicted in Fig. 4, the two input sequences are listed perpendicularly. The merge process can be seen as the traversal of a path from the upper left corner to the bottom right corner, and each step represents a comparison operation. This path is partitioned by equispaced cross diagonals, and the intersection points are computed using binary searches (search along the diagonal for the dividing point between $s > t$ and $s \leq t$). In this way, each segment of the path contains exactly the same number of merge steps (except the last segment, which we can handle through padding), resulting in a load-balanced partitioning.

This partitioning scheme can be easily applied to GPUs. We first employ a coarse-grained CTA-wide partitioning, assigning each CTA with the same num-

Algorithm 2. Sequential *expand*

Input: Array in with each element representing the number of elements to be produced, the length in_count of array in

Output: Array out with each element representing the index of the input element that produced it

Function: $ExclusiveScan(input)$ returns the scan result array and the sum of the input elements

1. $(s, total) \leftarrow ExclusiveScan(in)$
2. $si \leftarrow 0, ti \leftarrow 0$
3. **while** $si < in_count \cap ti < total$ **do**
4.   **if** $ti < s[si]$ **then**
5. $out[ti] \leftarrow si - 1$
6. $ti + +$
7.   **else** $si + +$;
8. **if** $si = in_count$ **then**
9. $out[ti...total - 1] \leftarrow si - 1$

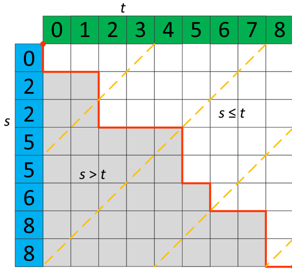

Fig. 4. Merge path partitioning

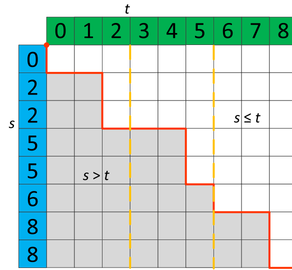

Fig. 5. Partitioning merge path vertically

ber $TILE_SIZE$ of elements to process, which is a tunable constant. This is done by placing each cross diagonal at a distance of $TILE_SIZE$ steps. After that, each CTA runs a similar fine-grained local partitioning to further assign $TILE_SIZE/CTA_SIZE$ input elements to each thread. When threads have obtained their independent segments of the input elements, they can run the sequential *expand* in parallel, leading to a high-performance parallel expansion.

4 Load-Balanced BFS

Having explained the parallel expansion in detail, we now use it as a work redistribution scheme to construct the full algorithms for a BFS iteration. We also explore the coupling possibilities of work redistribution and neighbor gathering.

To efficiently utilize the GPU memory model, we use the well-known compressed sparse row (CSR) format to store the graph in GPU main memory, which contains two arrays, namely column-indices C and row-offsets R.

4.1 Perfect Balance + Global Data Movement

The most straightforward approach is to separate the work redistribution phase and neighbor gathering phase into different kernels. In work reditribution phase, we read in the vertices in the input queue and construct the array $offsets$ holding the starting index of each vertexs adjacency list. Then we compute the adjacency list size $neighbor_num$ of each vertex, and run the parallel expansion using $neighbor_num$ as input. With the expansion output $vertex_index$ we can further compute the location of each neighbor to be gathered as:

$$\begin{aligned} gather_location = neighbor_index - scan_results[vertex_index] \\ + offsets[vertex_index]. \end{aligned} \tag{1}$$

These locations are then written to the output queue.

In neighbor gathering phase, the locations of all the neighbors are read back in. We then gather the neighbors at these locations and perform status lookup. Finally the valid vertices are output for distance update. As mentioned in Sect. 2.3, since the neighbor gathering is now fully load-balanced, the subsequent status lookup no longer needs to be put in a separate kernel.

This process is listed as algorithm 3, which requires at least five kernel launches (each code fragment marked by *in parallel* indicates a separate kernel). The work redistribution and neighbor gathering are both load-balanced: each CTA always processes $TILE_SIZE$ elements, making the algorithm insensitive to the difference in graph structure. But the net slowdown caused by writing and reading the work redistribution results will limit the obtained overall performance.

4.2 Imbalanced Redistribution + Balanced Gathering

A natural optimization is to fuse work redistribution and neighbor gathering. Unfortunately this will compromise the load balance property. Assume that the input queue has $input_total$ vertices and they generate $output_total$ neighbors. The redistribution phase will take $input_total + output_total$ elements as input but only output $output_total$ elements for the next phase, which makes the thread mapping policy inconsistent across the two phases if we fuse them together. In normal cases however, the performance gain through reduced I/O overhead can often make up for the impact of the load imbalance.

When coupling the redistribution and gathering, we may choose the thread mapping policy so that it benefits either process. We first focus on balanced gathering because it is the more time-consuming phase. Assuming each CTA processes $TILE_SIZE$ elements, the algorithm will assign $output_total/TILE_SIZE$ CTAs to perform the gathering. To achieve the coupling, the same number of CTAs should be assigned to the redistribution, and each CTA should output $TILE_SIZE$ elements at the end of the redistribution process which are then fed into the gathering process on the fly.

We use a different redistribution approach to fulfill these requirements. In the coarse-grained partitioning step, we partition the merge path vertically rather

Algorithm 3. Perfectly load-balanced BFS iteration

Input: inQ, $outQ$, inQ length in_total, column-indices array C and row-offsets array R

Output: Array $dist$

Function: $MergePathPartition(range, dist, k)$ partitions the merge path of the input $range$ diagonally at a distance of $dist$, and returns the kth independent range.

1 **foreach** $v \in inQ$ **in parallel do**
2 $\quad (start, end) \leftarrow (R[v], R[v+1])$
3 $\quad offsets[v_index] \leftarrow start$
4 $\quad neighbor_num \leftarrow end - start$
5 $\quad (inQ[v_index], out_total) \leftarrow ExclusiveScan(neighbor_num)$
6 $cta_num \leftarrow (in_total + out_total)/TILE_SIZE$
7 **for** $thread_id \in [0, cta_num - 1]$ **in parallel do**
8 $\quad coarse_range[thread_id] \leftarrow$
$\quad\quad MergePathPartition(whole_input, TILE_SIZE, thread_id)$
9 **for** $thread_id \in [0, cta_num * CTA_SIZE - 1]$ **in parallel do**
10 $\quad fine_range \leftarrow MergePathPartition(coarse_range[cta_id],$
$\quad\quad TILE_SIZE/CTA_SIZE, thread_id\%CTA_SIZE)$
11 $\quad$ shared array $indices \leftarrow Expand(fine_range)$
12 $\quad$ **foreach** $v_index \in indices$ **do** `/* strided */`
13 $\quad\quad gather_loc \leftarrow n_index - inQ[v_index] + offsets[v_index]$
14 $\quad\quad outQ[n_index] \leftarrow gather_loc$
15 **foreach** $loc \in outQ$ **in parallel do**
16 $\quad neighbor \leftarrow C[loc]$
17 $\quad$ **if** $StatusLookup(neighbor) = valid$ **then** scatter $neighbor$ to inQ
18 **foreach** $neighbor \in inQ$ **in parallel do**
19 $\quad$ update $dist$ array

than diagonally and at a distance of $TILE_SIZE$ as illustrated in Fig. 5. In this way, it is guaranteed that each CTA produces $TILE_SIZE$ outputs. We then do a fine-grained partitioning and expansion within each CTA.

This work redistribution process is relatively inefficient and imbalanced, because a CTA does not know the number of inputs it will process a priori. Fortunately, since the gathering process is the more time-consuming phase and is perfectly load-balanced, this approach can achieve better overall performance from the reduced I/O overhead.

4.3 Balanced Redistribution + Imbalanced Gathering

Another coupling strategy is to focus on balanced work redistribution. In this way, $(input_total + output_total)/TILE_SIZE$ CTAs are assigned to run the redistribution process as in the first approach, the outputs are then fed into the gathering process immediately. Since each CTA will take $TILE_SIZE$ elements

as input but produce less than $TILE_SIZE$ outputs, subsequent gathering can suffer from CTA imbalance: each CTA will gather 0 to $TILE_SIZE$ neighbors.

When *output_total* is much larger than *input_total*, the gathering load imbalance is negligible, and this approach can be more efficient than previous strategy because of the balanced redistribution process. However, if they are close, the imbalance problem can make the parallelism drop down by half, compromising the overall performance significantly.

4.4 Hybrid

The hybrid strategy combines the advantages of the *imbalanced redistribution + balanced gathering* and *balanced redistribution + imbalanced gathering* approaches. We define $expand_factor$ as the ratio of *output_total* versus *input_total*. If $expand_factor$ is larger than a threshold f_0 for a given BFS iteration, we invoke the *balanced redistribution + imbalanced gathering* approach because the gathering imbalance will be small enough to be safely ignored, achieving the best performance. Otherwise we invoke the *imbalanced redistribution + balanced gathering* approach to guarantee the efficiency of the gathering process, which dominates the overall performance.

By selecting an appropriate f_0, this strategy can ensure that the neighbor gathering phase is always load-balanced while the efficiency of the work redistribution phase is maximized.

5 Experimental Results

In this section, we evaluate the performance of the proposed BFS algorithms. Our algorithms are implemented using CUDA 5.5 [3], and all experiments are run on a host machine with 4GB memory, an Intel 3.4 GHz Core i7 2600k CPU and an NVIDIA Geforce GTX 580 GPU. For each graph, the BFS performance is measured by the average traversal throughput (edges per second) across 100 randomly-sourced traversals.

5.1 Strategy Evaluation

We first compare different coupling strategies of work redistribution and neighbor gathering. We use Random and R-MAT graphs generated with GTgraph [5], and adjust the average degree d to see its impact on each strategy. Fig. 6 and Fig. 7 plot the traversal throughput for each graph and strategy. All the graphs have 2 million vertices while d ranges from 2 to 64, and we choose the threshold f_0 to be 10 for the hybrid strategy. As anticipated, the *balanced redistribution + imbalanced gathering* strategy excels at traversing graphs with large d because the $expand_factor$ tends to be large for the BFS iterations. On the contrary, the *imbalanced redistribution + balanced gathering* strategy performs better on graphs with small d. The *hybrid* approach outperforms or is as good as the others in all the tests.

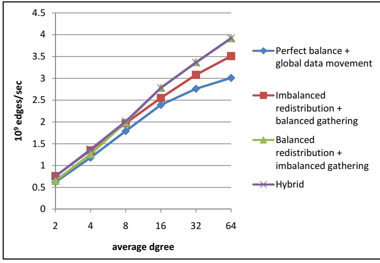

Fig. 6. Comparison of different strategies on random graphs

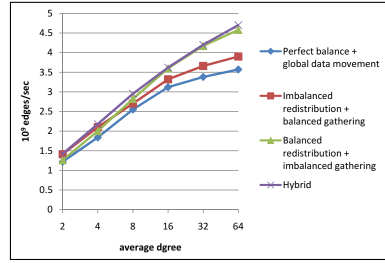

Fig. 7. Comparison of different strategies on R-MAT graphs

5.2 Comparison with Other Algorithms

To compare with the CPU implementation, we implement our own efficient sequential BFS according to the standard algorithm in [9], which has optimal single-threaded performance. For the GPU implementation, we compare our approach to that from Merrill et al. [11] which is the current state of the art and achieves the highest published performance for GPU BFS. Note that we compile and run Merrills source code on the same platform under the same configuration so the results are comparable.

Our benchmark suite incorporates twelve graphs listed in Table 1. In addition to *random* and *rmat*, the rest are from the Graph500 Competition [2], the 10th DIMACS Implementation Challenge [1] and the University of Florida Sparse Matrix Collection [4].

The results are presented in Table 1. As we can see, our hybrid approach performs very well compared to the CPU sequential BFS. For the majority of the graphs, our approach provides traversal speedups of an order of magnitude, and at the extreme, we achieve a 39x speedup for the *random* graph. We do not have an available parallel CPU BFS implementation, but we can simply assume a perfect 8x speedup for our 4-core/8-thread CPU, and our GPU approach still outperforms this theoretical performance upper bound for almost all the tests.

The state-of-the-art GPU implementation [11] employs a hybrid CTA+warp+scan strategy to do neighbor gathering. Through full load balance, our approach outperforms theirs for most of the tests, and we obtain up to 1.42x speedup. The last few tests also reveal the limitation of our approach. The advantage of load balance comes at the price of additional preprocessing of the input vertices. When the search depth is small, the number of vertices examined each iteration is large enough to fully utilize the high throughput of GPU hardware, making the preprocessing overhead negligible. However, when the search depth gets high, the overhead of the work redistribution process and additional kernel launches starts to become significant. As a result, the performance gain through load-balanced gathering is outweighed by the preprocessing penalty and we observe a slowdown for *germany.osm* and *hugebubbles-00020* datasets. In practice, we can choose to apply direct gathering for graphs with large diameters.

Table 1. Traversal rates ($10^9 edges/sec$) for different algorithms running on different graphs

Name	**Description**	**Vertices** (10^6)	**Edges** (10^6)	**Avg. Search Depth**	**Our hybrid algorithm** ($f_0 = 10$)	**Sequential BFS (speedup)**	**Merrill's algorithm (speedup)**
random	Uniform random	2.0	128.0	6	3.9	0.10 (39x)	3.0 (1.31x)
rmat	R-MAT (A=0.45, B=0.15, C=0.15)	2.0	128.0	6	4.7	0.16 (29x)	3.4 (1.40x)
kron_g500-logn20	Graph500 R-MAT (A=0.57, B=0.19, C=0.19)	1.0	100.7	7	4.5	0.20 (22x)	3.3 (1.36x)
hollywood-2009	Hollywood movie actor network	1.1	113.9	10	4.1	0.18 (23x)	3.1 (1.32x)
flickr	2005 crawl of flickr.com	0.8	9.8	12	3.6	0.18 (20x)	3.2 (1.13x)
eu-2005	small web crawl of .eu domain	0.9	32.3	14	4.3	0.47 (9.1x)	3.7 (1.16x)
wikipedia-20070206	Links between Wikipedia pages	3.6	45.0	20	2.6	0.069 (38x)	1.8 (1.42x)
FullChip	Circuit simulation	3.0	26.6	38	3.1	0.26 (12x)	2.4 (1.29x)
audikw1	Automotive finite element analysis	0.9	76.7	62	3.3	0.65 (5.1x)	3.2 (1.03x)
wb-edu	Links between *.edu web pages	9.8	57.2	143	2.9	0.16 (18x)	2.7 (1.07x)
germany.osm	Germany road network	11.5	24.7	5345	0.32	0.034 (9.4x)	0.38 (0.84x)
hugebubbles-00020	Adaptive numerical simulation mesh	21.2	63.6	6200	0.37	0.031 (12x)	0.48 (0.77x)

6 Conclusion

Load balance is a key factor in designing efficient algorithms for GPUs. We have demonstrated a GPU BFS algorithm that leverages the unique wide SIMD GPU architecture and achieves fully load-balanced neighbor gathering. We have showed that our approach achieves very high performance on a broad range of graphs, and outperforms current state-of-the-art implementation.

In order to exploit the nested and irregular parallelism on a BFS iteration, we have introduced a work redistribution process to flatten the nested workloads. It utilizes a parallel expansion to compute the gathering location of each neighbor so the neighbor gathering process can visit the neighbors in a load-balanced way. We have also explored the coupling possibilities of different phases, and proposed a hybrid approach that yields the best overall performance.

Acknowledgements. We thank all the anonymous reviewers for their valuable comments. This work is substantially supported by National Natural Science Foundation of China under Grants No.61300045, and China Postdoctoral Science Foundation under Grant No.2013M531696.

References

1. 10th dimacs implementation challenge, http://www.cc.gatech.edu/dimacs10/index.shtml
2. The graph 500 list, http://www.graph500.org/
3. Nvidia cuda, http://www.nvidia.com/cuda/
4. University of florida sparse matrix collection, http://www.cise.ufl.edu/research/sparse/matrices/
5. Bader, D.A., Madduri, K.: Gtgraph: A synthetic graph generator suite, Atlanta, GA (February 2006)
6. Deo, N., Sarkar, D.: Parallel algorithms for merging and sorting. Information Sciences 56(1), 151–161 (1991)
7. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007)
8. Hong, S., Kim, S.K., Oguntebi, T., Olukotun, K.: Accelerating cuda graph algorithms at maximum warp. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, pp. 267–276. ACM (2011)
9. Leiserson, C.E., Rivest, R.L., Stein, C., Cormen, T.H.: Introduction to algorithms. The MIT Press (2009)
10. Luo, L., Wong, M., Hwu, W.M.: An effective gpu implementation of breadth-first search. In: Proceedings of the 47th Design Automation Conference, pp. 52–55. ACM (2010)
11. Merrill, D., Garland, M., Grimshaw, A.: Scalable gpu graph traversal. In: ACM SIGPLAN Notices, vol. 17, pp. 117–128. ACM (2012)
12. Nasre, R., Burtscher, M., Pingali, K.: Data-driven versus topology-driven irregular computations on gpus. In: 2013 IEEE 27th International Symposium onParallel & Distributed Processing (IPDPS), pp. 463–474. IEEE (2013)
13. Nguyen, H.: Gpu gems 3. Addison-Wesley Professional (2007)
14. Odeh, S., Green, O., Mwassi, Z., Shmueli, O., Birk, Y.: Merge path-parallel merging made simple. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), pp. 1611–1618. IEEE (2012)
15. Shiloach, Y., Vishkin, U.: Finding the maximum, merging, and sorting in a parallel computation model. Journal of Algorithms 2(1), 88–102 (1981)
16. Zhong, J., He, B.: Medusa: Simplified graph processing on gpus. IEEE Transactions on Parallel and Distributed Systems 99, 1 (2013) (PrePrints)