An Effective GPU Implementation of Breadth-First Search

Lijuan Luo Martin Wong Wen-mei Hwu
Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign
{lluo3, mdfwong, w-hwu}@illinois.edu

ABSTRACT

Breadth-first search (BFS) has wide applications in electronic design automation (EDA) as well as in other fields. Researchers have tried to accelerate BFS on the GPU, but the two published works are both asymptotically slower than the fastest CPU implementation. In this paper, we present a new GPU implementation of BFS that uses a hierarchical queue management technique and a three-layer kernel arrangement strategy. It guarantees the same computational complexity as the fastest sequential version and can achieve up to 10 times speedup.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Algorithms, Performance

Keywords

CUDA, GPU computing, BFS

1. INTRODUCTION

The graphics processing unit (GPU) has become a popular cost-effective parallel platform in recent years. Although parallel execution on a GPU can easily achieve speedup of tens or hundreds of times over straightforward CPU implementations, to accelerate intelligently designed and well optimized CPU algorithms is a very difficult job. Therefore, when researchers report exciting speedup on GPUs, they should be very careful about the baseline CPU algorithms. Fig. 1 illustrates the performance difference between the 1000x accelerated n^2 algorithm and the $n\log n$ algorithm. After n grows to a certain point, the greatly accelerated n^2 algorithm becomes slower than the un-accelerated $n\log n$ algorithm. Hence the speedup is only meaningful when it is over the best CPU implementation. A similar observation was also made in [1]. Breadth-first search (BFS) is a graph algorithm that has extensive applications in computer-aided design as well as in other fields. However, it is well known that accelerating graph algorithms on GPUs is very challenging and we are aware of only two published works [2, 3] on accelerating BFS. Harish and Narayanan pioneered the acceleration of BFS on the GPU [2]. This work has great impact in high performance computing and graph algorithm areas. It has been cited not only by research papers but also in NVIDIA CUDA Zone and university course materials. However, for certain types of graphs, such as sparse graphs with large BFS levels, this work is still slower than the fastest CPU program. The other work [3] showed 12 to 13 times speedup over a matrix-based BFS implementation, but it should be noted that such BFS implementation is slower than the traditional BFS algorithm of [8].

*This work was partially supported by the National Science Foundation under grant CCF-0701821. Experiments were also made possible by NSF CNS grant 05-51665 and the CUDA Center of Excellence, sponsored by NVIDIA Corporation.

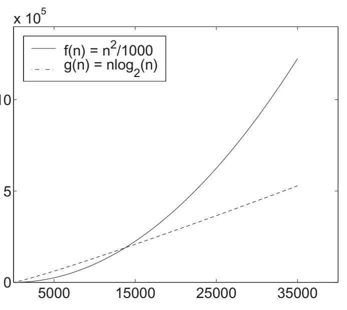

Figure 1: Compare performance difference between 1000x accelerated n^2 algorithm and $n\log n$ algorithm

This paper will present a new GPU implementation of BFS. It uses a hierarchical technique to efficiently implement a queue structure on the GPU. A hierarchical kernel arrangement is also proposed to reduce synchronization overhead. Experimental results show that up to 10 times speedup is achieved over the classical fast CPU implementation [8].

2. PREVIOUS APPROACHES

Given a graph $G = (V, E)$ and a distinguished *source* vertex s, breadth-first search (BFS) systematically explores the

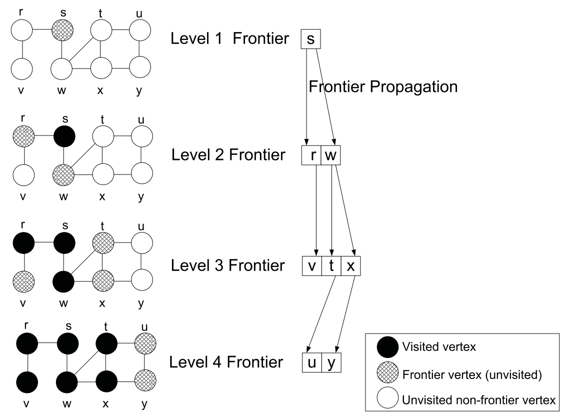

Figure 2: The operation of BFS

edges of G to discover every vertex that is reachable from s. We use $V(E)$ to represent both the vertex (edge) set and the number of vertices (edges). BFS produces a breadth-first tree with root s that contains all reachable vertices. The vertices in each *level* of the tree compose a *frontier*. *Frontier propagation* checks every neighbor of a frontier vertex to see whether it is visited already; if not, the neighbor is added into the new frontier. Fig. 2 shows an example operation of BFS. The traditional BFS algorithm outlined in [8] uses a queue structure to store the frontier. Its computational complexity is $O(V + E)$. For sparse graphs with $E = O(V)$, the complexity of BFS is $O(V)$.

The researchers from IIIT presented the first work of implementing BFS on the GPU [2]. (Hereafter, we will refer to this work as IIIT-BFS.) It points out that to maintain the frontier queue can cause a huge overhead on the GPU. Hence they eliminate the frontier structure. Instead, for each level, they exhaustively check every vertex to see whether it belongs to the current frontier. Let L be the total number of levels; then the total time spent to check frontier vertices is $O(VL)$. The time to explore every edge of the graph is $O(E)$. Therefore, the time complexity of IIIT-BFS is $O(VL + E)$, higher than $O(V + E)$ of the CPU algorithm. In the worst case scenario, we have $L = O(V)$. It means a complexity of $O(V^2 + V) = O(V^2)$ for sparse graphs, much slower than the $O(V)$ implementation. Based on our observation in Section 1, the accelerated version of a slow algorithm is still worse than a fast sequential algorithm. This will be further verified by the experimental results shown in Section 4. Of course, when the graph is dense with $E = O(V^2)$, or when L is very small, IIIT-BFS can be faster than the sequential program.

The second GPU work [3] accelerated a matrix-based BFS algorithm for sparse graphs. In this BFS procedure, each frontier propagation can be transformed into a matrix-vector multiplication; hence there are totally L multiplications, where L is the number of levels. The running time of each matrix-vector multiplication is $O(E)$. In addition, it takes $O(V)$ time to initialize the vector. Therefore the total running time is $O(V + EL)$, higher than $O(V + E)$. Again in the worst case, we have $L = O(V)$ and the running time is $O(V + EL) = O(V^2)$. Hence the same argument about IIIT-BFS applies here. For large graphs, the accelerated matrix-based BFS will still be slower than the traditional sequential BFS.

Lauterbach et al. solved a different BFS problem in [4]. Under the framework of BFS, they do intensive computation on each node. This work uses a compaction technique to maintain the queue structure on the GPU. However, the compaction will add intolerable overhead to our BFS problem.

There are also parallel BFS algorithms proposed for other parallel architectures such as [5]-[7]. However, since different architectures have different efficiencies for synchronization, atomic operations, memory accesses, etc., the parallelism ideas applicable to those architectures may not be used on the GPU. As far as we know, the ideas proposed in this paper were never used on other architectures.

3. OUR GPU SOLUTION

The BFS process propagates through the graph by levels. In each level, there are two types of parallelism: one is to propagate from all the frontier vertices in parallel, and the other is to search every neighbor of a frontier vertex in parallel. Since a lot of EDA problems such as circuit simulation, static timing analysis and routing are formulated on sparse graphs, we do not expect many neighbors for each frontier vertex. Therefore, our implementation of BFS only explores the first type of parallelism, i.e. each thread is dedicated to one frontier vertex of the current level.

In the following, we will first introduce the NVIDIA GTX280 GPU architecture and the CUDA programming model. Then we introduce an efficient queue structure to store the new frontier generated in each level. Finally we present hierarchical kernel arrangement to reduce the synchronization overhead. Note that our solution has the same computational complexity as the traditional CPU implementation.

3.1 Overview of CUDA on the Nvidia GTX280

The NVIDIA GeForce GTX280 graphics processor is a collection of 30 multiprocessors, with 8 streaming processors each. The 30 multiprocessors share one off-chip global memory. Note that global memory is not cached, so it is very important to achieve *memory coalescing*. Memory coalescing happens when consecutive threads access consecutive memory locations. In this case, several memory transactions can be *coalesced* into one transaction. Within each multiprocessor there is a shared memory, which is common to all 8 streaming processors inside the multiprocessor. The shared memory is on-chip memory. To access a shared memory only takes 2 clock cycles compared with approximately 300 clock cycles for global memory.

We use the NVIDIA CUDA computing model to do parallel programming on the GPU. A typical CUDA program consists of several phases that are executed either on the host CPU or on the GPU. The CPU code does the sequential part of the program. The phases that exhibit rich parallelism are usually implemented in the GPU code, called *kernels*. The action of calling a GPU function from the CPU code is termed *kernel launch*. When a kernel is launched, a two-level thread structure is generated. The top level is called *grid*, which consists of one or more *blocks*. Each block consists of the same number (at most 512) of threads. The whole block will be assigned to one multiprocessor.

3.2 Hierarchical Queue Management

It is difficult to maintain the new frontier in a queue because different threads are all writing to the end of the same queue and end up executing in sequence. To avoid the collision on the queue, we introduce a hierarchical queue structure. The idea is that once we have quickly created the lower-level queues, we will know the exact location of each element in the higher-level queue, and therefore copy the elements to the higher-level queue in parallel.

A natural way to build the frontier hierarchy is to follow the two-level thread hierarchy, i.e. build the grid-level frontier based on block-level frontiers. A block-level frontier can be stored in the fast shared memory. However, this strategy still cannot avoid the collision at the block level. Hence we add another level – warp level – into the hierarchy and completely eliminate collisions on the warp-level queues. Fig. 3 shows the overall hierarchy.

A *W-Frontier* is defined as a frontier only accessed by certain threads from a warp. A warp is the scheduling unit in GTX280, which is composed of 32 consecutive threads. Since there are only 8 streaming processors within each multiprocessor, a warp is further divided into four 8-thread groups. Fig. 4 illustrates how threads are scheduled along the timeline. The vertical lines represent the clock boundaries. $W_i T_\alpha$ represents Thread α from Warp i. We do not know the scheduling order among warps, but once a warp is scheduled, it always runs T_1-T_8 first, followed by T_9-T_{16}, T_{17}-T_{24}, and finally T_{25}-T_{32}. In GTX280, threads in the same row of Fig. 4 never execute simultaneously. If each row of threads write to one W-Frontier queue, we can guarantee no collision. This scheme is portable as long as the warp size assumption is parameterized. Note that writing to a queue actually includes two steps, i.e. write an element to the end of the queue and update the end of the queue. To guarantee the correctness in parallel execution, we use an atomic operation, so that we can get the current end location and update it in one instruction.

A *B-Frontier* is defined as the frontier common to a whole block. It is simply the union of 8 W-Frontiers. To copy the W-Frontier elements into the B-Frontier in parallel, we need indices of each element in the B-Frontier. With only 8 W-Frontiers, we use one thread to calculate the offsets for all W-Frontiers in the B-Frontier. Then the index of an element in the B-Frontier is simply its index within its W-Frontier plus the offset of its W-Frontier. Note that both W-Frontiers and B-Frontiers are stored in the fast shared memory.

Finally, a *G-Frontier* is defined as the frontier shared by all the threads of a grid. In other words, the G-Frontier stores the complete new frontier. A G-Frontier resides in global memory and consists of B-Frontiers from all the blocks. We use atomic operations to obtain the offsets of B-Frontiers so that parallel copying from B-Frontiers to the G-Frontier can be performed. Note that in the parallel copy, it is natural to have consecutive threads writing to consecutive locations in the G-Frontier; thus, memory coalescing is guaranteed.

3.3 Hierarchical Kernel Arrangement

The correct BFS implementation requires thread synchronization at the end of each level. Unfortunately CUDA does not provide any global barrier synchronization function across blocks. A general solution is to launch one kernel for each level and implement a global barrier between two launched kernels, which inflicts a huge kernel-launch overhead. This section presents hierarchical kernel arrangement, where only the highest layer uses this expensive synchronization method and the others use more efficient GPU

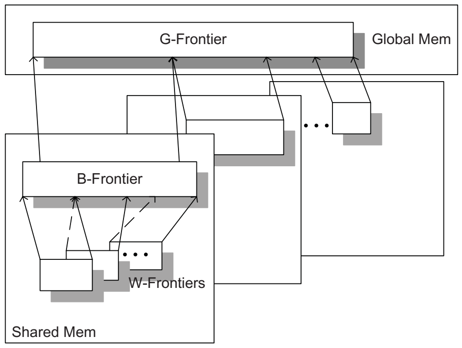

Figure 3: Hierarchical frontiers

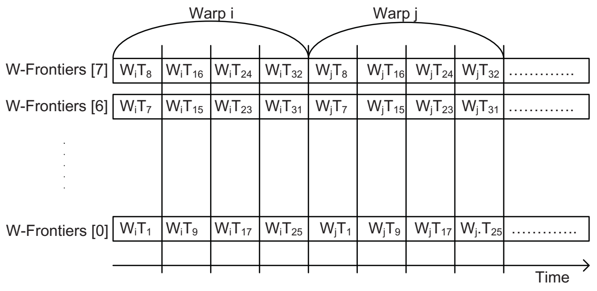
Warp i
Warp j
W-Frontiers [7]
W_iT_8 W_iT_{16} W_iT_{24} W_iT_{32} W_jT_8 W_jT_{16} W_jT_{24} W_jT_{32} ....
W-Frontiers [6]
W_iT_7 W_iT_{15} W_iT_{23} W_iT_{31} W_jT_7 W_jT_{15} W_jT_{23} W_jT_{31} ....
W-Frontiers [0]
W_iT_1 W_iT_9 W_iT_{17} W_iT_{25} W_jT_1 W_jT_9 W_jT_{17} $W_j.T_{25}$ ....
Time

Figure 4: Thread Scheduling and W-Frontier design

synchronization.

GPU synchronization is a synchronization technique without kernel termination. It includes intra-block synchronization and inter-block synchronization. In the following discussion, we assume the largest possible number of threads – 512 – in each block.

Intra-block synchronization uses the CUDA barrier function to synchronize threads within one block. Hence it only applies to the levels with frontiers of no larger than 512 vertices. Note that once a kernel is launched, the number of threads in this kernel cannot be changed anymore. Therefore, when launching such a single-block kernel from the host CPU, we always launch 512 threads regardless of the size of the current frontier. This kernel can propagate through multiple levels with an intra-block synchronization at the end of each level. Now that there is only one working block, no G-Frontier is needed and global memory access is also reduced.

Once the frontier outgrows the capacity of one block, we use the second GPU synchronization strategy, which can handle frontiers as large as 15 360 vertices. *Inter-block synchronization* synchronizes threads across blocks by communicating through global memory. Xiao and Feng introduced the implementation details of inter-block synchronization [10]. To apply this synchronization strategy, the number of blocks should not be larger than the number of multiprocessors. In GTX280, this means at most $512 \times 30 = 15\ 360$ threads in the grid. We use G-Frontier at this level.

Only when the frontier has more than 15 360 vertices, we call the top-layer kernel. This kernel depends on kernel termination and re-launch for synchronization. With more than 15 360 vertices, the kernel-launch overhead becomes acceptable compared to the propagation work performed.

Table 1: BFS results on regular graphs

#Verte	IIIT-BFS	CPU-BFS	UIUC-BFS	Sp.
1M	462.8ms	146.7ms	67.8ms	2.2
2M	1129.2ms	311.8ms	121.0ms	2.6
5M	4092.2ms	1402.2ms	266.0ms	5.3
7M	6597.5ms	2831.4ms	509.5ms	5.6
9M	9170.1ms	4388.3ms	449.3ms	9.8
10M	11019.8ms	5023.0ms	488.0ms	10.3

Table 2: BFS results on real world graphs

	#Vertex	IIIT-BFS	CPU-BFS	UIUC-BFS	Sp.
New York	264,346	79.9ms	41.6ms	19.4ms	2.1
Florida	1,070,376	372.0ms	120.7ms	61.7ms	2.0
USA-East	3,598,623	1471.1ms	581.4ms	158.5ms	3.7
USA-West	6,262,104	2579.4ms	1323.0ms	236.6ms	5.6

Table 3: BFS results on scale-free graphs

#Vertex	IIIT-BFS	CPU-BFS	UIUC-BFS
1M	161.5ms	52.8ms	100.7ms
5M	1015.4ms	284.0ms	302.0ms
10M	2252.8ms	506.9ms	483.6ms

4. EXPERIMENTAL RESULTS

All experiments were conducted on a host machine with a dual socket, dual core 2.4 GHz Opteron processor and 8 GB of memory. A single NVIDIA GeForce GTX280 GPU was used to run CUDA applications. We first tested the programs on degree-6 "regular" graphs. (For simplicity, we used grid-based graphs. Therefore, only the vertices inside the grids are of degree 6. The very few vertices on the boundaries of the grids have degrees less than 6. And the source vertices are always at the centers of the grids.) The experimental results are shown in Table 1. Here UIUC-BFS represents our GPU implementation of BFS, and CPU-BFS is the BFS implementation outlined in [8]. The last column shows the speedup of UIUC-BFS over CPU-BFS. We can see that IIIT-BFS (The source code was obtained from the original authors.) is slower than CPU-BFS just as we discussed in Section 2. On the other hand, UIUC-BFS is faster than CPU-BFS on all the benchmarks. On the largest one, up to 10 times speedup was achieved. We also downloaded the same four graphs as [2] from the DIMACS challenge site [9]. All these graphs have average vertex degree of 2, and the maximum degree can go up to 8 or 9. Again IIIT-BFS is slower than CPU-BFS, and UIUC-BFS achieves speedup over CPU-BFS (Table 2). Finally, we tested the BFS programs on scale-free graphs. Each graph was built in the following way. We made 0.1% of the vertices have degrees equal to 1000. The remaining vertices have the average degree of 6 and the maximum degree of 7. Table 3 shows the running times of the three BFS programs. Although UIUC-BFS is still much faster than IIIT-BFS, its running time is close to or even longer than the running time of CPU-BFS. The result shows that the highly imbalanced problems cannot benefit as much from the parallelism as the relatively balanced ones

Note that our GPU implementation pre-allocates the share memory space for W-Frontiers and B-Frontiers based on the maximum node degree. If the maximum degree is much larger than the average degree, the memory usage becomes very inefficient. Thus, when handling very irregular graphs, we first converted them into near-regular graphs by splitting the big-degree nodes. A similar idea was used in [11]. This is a legitimate solution because most of the applications run BFS on the same graph a great number of times (if only run once, the BFS process is not really slow), and we only need to do the conversion once.

5. CONCLUSION

We have presented a BFS implementation on the GPU. This work is most suitable for accelerating sparse and near-regular graphs, which are widely seen in the field of EDA. Both methods proposed in this paper – hierarchical queue management and hierarchical kernel arrangement – are potentially applicable to the GPU implementations of other types of algorithms, too.

6. REFERENCES

[1] C. Rodrigues, D. Hardy, J. Stone, K. Schulten, and W. Hwu, "GPU acceleration of cutoff pair potentials for molecular modeling applications," in *ACM International Conference on Computing Frontiers*, 2008, pp. 273-282.

[2] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *IEEE High Performance Computing*, 2007, pp 197-208.

[3] Y. Deng, B. Wang, and S. Mu, "Taming irregular EDA applications on GPUs," in *ICCAD*, 2009, pp. 539-546.

[4] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs," *Computer Graphics Forum*, vol. 28, no. 2., pp. 375-384, 2009.

[5] A. Yoo, E. Chow, K. Henderson, W. Mcledon, B. Hendrickson, and *Ü C*atalyürek, "A scalable distributed parallel breadth-first search algorithm on Bluegene/L," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2005, pp. 25-32.

[6] D.A. Bader and K. Madduri, "Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2," in *ICPP*, 2006, pp. 523-530.

[7] D.P. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the Cell/BE processor," *IEEE Trans. on Parallel Distributed Systems*, vol.19, no. 10, pp. 1381-1395, 2008.

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

[9] http://www.dis.uniroma1l.it/∼challenge9/

[10] S. Xiao and W. Feng, "Inter-block GPU communication via fast barrier synchronization," Technical Report TR-09-19, Dept. of Computer Science, Virginia Tech.

[11] U. Meyer, "External memory BFS on undirected graphs with bounded degree," in *SODA*, 2001, pp. 87-88.