Spanning Large Graphs by Combining Work-stealing with Multiple Parallel Granularities on GPU

Hao Peng, Jifeng Chen, Qiao Li, Cheng Zhang
School of Information Science and Engineering, Hunan International Economics University, Changsha 410205, Hunan, China

Abstract
Graphs are fundamental representations which are extensively used in many kinds of computation domains. Despite its irregular memory accesses and unbalanced workload graph spanning on GPU has recently shown promising results in accelerating those challenging graph problems. In this paper we propose a policy spanning large graphs by combining work-stealing with multiple parallel granularities on GPU. Among them warp-based spanning and threadblock-based spanning can achieve spanning acceleration for graphs with higher degrees skew. Moreover, this coarse-grained spanning can reduce branch divergence and scatter memory accesses. On the other hand work-stealing mechanism can keep workload balanced among stream multi-processors inside GPU and maintain efficient execution of each multi-processor. The relevant experiments of graphs instances have confirmed our method can gain 10x speedup over baseline implementation. Meanwhile our tests also have confirmed that the performance gap between GPU and other architectural machines is predominantly due to memory bandwidth bottleneck.

Key words: Breadth First Spanning, GPU, Work-stealing, Warp-based Spanning.

1. INTRODUCTION

Now large graphs applications are drawing many scientists’ eyes in social network analysis or biological information processing. However, it is challenging to efficiently process large graphs because of their lack of spatial and temporal locality in memory access pattern exhibited in graph spanning. To accurately define the computational requirements of large graphs applications a benchmark named Graph500 has been created (Mark, 1977).

For application of large graphs, breadth-first spanning (BFS) is an important branch. But when considering its scalable implementation, it is hard to achieve good parallel performance. Frequent accesses to irregular and dynamic data structures result in tangle some synchronization and much latency. For working out these harsh terms, a great amount of literature may be referred to, either based on commodity processor (Virat et al., 2010; Andyet et al., 2005; Venkata et al., 1997) or special purpose hardware (Michael et al., 2006; Daniele et al., 2008). For an example, Agarwal et al. (Virat et al., 2010) proposed a scalable BFS algorithm which saves memory bandwidth by bitmaps and coherence traffic avoidance methods for multi-core CPU. Their experiment evaluation showed their graph exploration algorithm could run 2.4 times faster than a Cray XMT with 128 processors for some random graphs with 64 million vertices and 512 million edges.

Despite the graphics processing units (GPU) wide spread in parallel acceleration in recent years, accelerating graph algorithms on GPU challenges us. The speedup of GPU derives from massive fine-grained threads. For BFS massive synchronization can cause deadlock because of graphs spanning’s dynamic workload characteristics. Within such mechanism it is not easy to implement parallel algorithms which dynamically allocate and relocate data. Some recent papers indicated that GPU solution to BFS had worked poorly due to the workload imbalance between threads caused by the irregularity of graphs instance. For the moment, we at least are aware of Harish and Narayanan’s pioneering BFS on the GPU (Pawan et al., 2007) which has greater impact in high performance computation and graph algorithms. Despite the algorithm’s claimed efficiency, for some real-world irregular large graph applications, this work is still slower than the traditional CPU graph’s breadth-first spanning (Virat et al., 2010). In (Honget et al., 2011) a warp parallel granularity method has been proposed. The method makes full use of coarser parallel granularity to balance workload between SIMD ALU inside GPU. Duane Merril et al put forward finer-grained task management mechanism to achieve an asymptotically optimal $O(|V|+|E|)$ time complexity for BFS inside GPU (Duane et al., 2012). In (Lijuan et al., 2010) its BFS implementation adopted a hierarchical queue management technique to achieve 10 time speedup of sequential version implementation.

In this paper we first manifest that performance reduction of BFS derives from significant diversity of number of edges of nodes which leads to workload imbalance between stream multiprocessors, branch divergence which idles SIMD cores and scatter memory accesses which waste memory bandwidth. Then we come up with different graph spanning granularities on the basis of threshold in number of edges of nodes: graph nodes spanning...
based on threads blocks, warp-based parallel nodes spanning and fine-grained thread-based nodes spanning. To further balance workload between stream multiprocessors we delicately adopt work-stealing method based on threads blocks of CUDA which is widely used to balance workload in multi-core CPU.

In summary, the main contributions of our work are as follows:

- Warp-based graph spanning by which we can reduce branch divergence and scatter memory accesses.
- Threadblock-based graph spanning by which a graph instance with larger degrees skewing can be efficiently spanned.
- Work-stealing mechanism which can balance workload among different stream multi-processors.
- A detailed performance mechanism which shows that our graph spanning mechanism is superior to other implementations and memory accesses bandwidth is the bottleneck of speedup.

The rest of this paper is organized as follows: in section 2, we provide some necessary background for GPU and BFS. Then we present our multiple granularities spanning in section 3 and work-stealing mechanism in section 4. In section 5 a delicate performance evaluation is given. At last, we introduce the related work in section 6 and make a conclusion in section 7.

2. BACKGROUND

2.1 GPU and CUDA Programming Model

GPU consists of tens of multiprocessor cores named stream multiprocessors (SM) which are the units of computation to which groups of threads, namely threads block, are allocated by the CUDA runtime for parallel execution. Every SM is made up of instruction fetch units, multiple SIMD cores (i.e. stream processors), a shared memory accessible by all threads in the SM and a large register file private by every hardware thread. SM contains only enough SIMD core to execute a few of active warps (fixed-size group of threads), yet schedules all warps to mask memory latency and pipeline hazard. CUDA (CUDA, 2012) is a general-purpose graphics processing unit (GPGPU) solution of NVIDIA which offers direct access to the GPU hardware interface through a C-like language, rather than the traditional approach that reckons on the graphics programming interface. It organizes virtual threads groups to map to physical stream multiprocessor cores by two-level hierarchy. Among them a grid is a collection of homogenous threadblocks and forms the executable kernel, a threadblock is composed of a group of threads which should be collocated on the same SM and have a sharable memory space.

GPU is originally oriented for maximum throughput of orderly-structured computation. Once the computation becomes dynamic and irregular GPU performance declines significantly. These performance penalty can be classified as irregular memory access patterns that cannot be coalesced or can lead to bank conflicts; control flow divergences between threads of warp which can lead to serial execution of threads of warp and load imbalance which can lead to resource underutilization.

2.2 Parallel Graph Breadth-first Algorithm on GPU

For graph of the form \( G = (V, E) \) where the sets \( V \) and \( E \) are respectively made up of \( n \) vertices and \( m \) directed edges, we introduce the well-known compressed sparse row (CSR) (Nathan et al., 2012) format to store the graph in memory. This data structure includes arrays of nodes and edges respectively, where each element of nodes arrays stores the start index of the corresponding edges outgoing from each node and edges array stores destination nodes of each edge, as illustrated in Figure 1.

Figure 2 gives a sample algorithm aimed at the above data structures modified from (Pawan et al., 2007). The algorithm performs a breadth-first spanning from the given source node. The \textit{cost} represents the minimum number of edges spanning from the source node to this node; \textit{num_edges} represents number of edges of nodes. In its execution the kernel must be called many times for visiting all reachable nodes. At every invocation, every thread visits a node which has the same value of \textit{cost} and marks all of unvisited neighbor's nodes with the updated value incremented by one. minimum number of edges spanning from the source node to this node; \textit{num_edges} represents number of edges of nodes. In its execution the kernel must be called many times for visiting
all reachable nodes. At every invocation, every thread visits a node which has the same value of \textit{cost} and marks all of unvisited neighbor nodes with the updated value incremented by one.

```c
struct graph {
    int edges[m];
    int nodes[n+1];
    int cost[n];
}
void main_BFS (graph * G, int source) {
    cost_initial(g->cost, source);
    while (!ended) {
        kernel_begin(g, temp++, &ended);
    }
}
_device_ void bfs_kernel(int temp, int n, int *cost, int * edges, in * nodes, bool * ended)
{
    int v = threadIdx.x;
    if (cost[v] == temp) {
        int num_edges = nodes[v+1] - nodes[v];
        int * neighbor = &edges[nodes[v]];
        for (int i = 0; i < num_edges; i++) {
            int s = neighbor[i];
            if (cost[s] == infinity) {
                * ended = false;
                cost[s] = temp + 1;
            }
        }
    }
}
```

Figure 2. The primary algorithm of BFS on GPU

The primary algorithm of Figure 2 suffers several severe performance penalty. First for irregular graphs, i.e. when the distribution of degrees (number of edges per node) is highly varied, the workload among stream processors is greatly differentiated. This performance bottleneck of mainly derives from the fact when a high-degrees node gives rise to many times iterations at line 18. On top of that, lines 16, 17, 20 and 22 do not appear to access memory by coalesced pattern (David et al., 2010) for all threads. All of these access operations reflect no spatial locality to waste memory bandwidth significantly. At last the execution path divergence (Wilson et al., 2009) of line 15, 18 and 20 degrades performance.

2.3 Spanning Graph with Fine-grained Parallel

In this fine-grained parallel spanning (Duane et al., 2012), a shared array of column indices offsets was constructed for all threads in each threadblock. Then by computing prefix sum (Merrill et al., 2009) neighbors of nodes can be located in the compacted shared array and at the same time no SIMD lanes are unutilized during the global reads from column indices.

For this spanning scheme, all threads read the column indices. Any workload imbalance between threads is not exacerbated by expensive global memory access. Instead any imbalance stemming from non-uniformly-sized neighbors list is produced in the form of underutilized cycles during the local offset-sharing. In spite of the larger shared buffer which can be used to reduce imbalance, the worst case may take place, where a single node has more degrees than the shared buffer can be loaded.

2.4 Work-stealing Principle

In work stealing each computation unit has its own queue of tasks to deal with. Whenever the computation unit has finished its tasks, it tries to steal more tasks from another computation unit. Of course each new-created task needs adding to its own queue. So work-stealing mechanism causes tasks to redistribute evenly over different cores while minimizing the communication costs.

With the recent advent of scatter operations and atomic hardware primitives it is now possible to bring the elaborate work-stealing principle original to dynamically keep balanced workload among processors of the conventional SMP to GPU. Cederman et.al. (Daniele et al., 2008) applied the task of creating an octree partitioning to compare four different policies for dynamic load balancing. In (Miguel et al., 2010) the shortest path problem is used as an irregular application to evaluate a framework for dynamic work scheduling based on Blumofe and Leiserson’s algorithm (Blumofe et al., 1999).

In the work-stealing optimization it is important to weigh up to have the work queue stored on the global memory or on the shared memory. Storing the work queue on the shared memory can reduce the number of
accesses to the global memory. Nevertheless the work queue would only be stolen by blocks running on the same multiprocessor, what is more, because of the popular use of large blocks (512 thread per block) to benefit more from the SIMD parallelism there exist few concurrent blocks sharing the same multiprocessor.

3. SPANNING GRAPH WITH MULTIPLE GRANULARITIES

With great diversity of degrees of nodes on graph, conventional spanning in Figure 2 suffers from load imbalance between threads. Fine-grained spanning in (Duane et al., 2012) lead to overblown shared buffer. Coarser-grained warp-based spanning like (Hong et al., 2011) suffers from insufficient concurrency, exacerbated underutilization of warp SIMD lanes and uncoalesced warp read. So we may adopt a scheme of synthesizing multiple different grains to avoid the above bottleneck.

![Figure 3. BFS based on warps](image)

When degrees of some nodes have a medium value we may take warp-based breadth-first spanning to reduce significant load imbalance between threads, as illustrated by Figure 3. First of all, the threshold value of degrees of node is tested. Then for each offset of the warp once the thread has updated all neighbors of current node, the corresponding node will be spanned. There at least exist two strong points for our spanning policy. Firstly within one warp, the data structures of neighbors of node can be accessed continuously (Nathan et al., 2012). So memory access patterns can be more coalesced than the conventional thread-level task allocation in applications where concurrent memory accesses within a task exhibit much higher spatial locality than across different tasks. On the other hand since warps execute independently and times of loop in line 11 decrease significantly, unlike the primary algorithm in Figure 2, warp-based spanning algorithm reduces workload divergence among threads, in other words, it is less likely that many threads within a warp will remain idle while the thread with the largest degrees continues iterating.

![Figure 4. BFS based on threadblocks](image)
While the warp-based spanning reduces the amount of load imbalance among threads there exists opportunities for significant imbalance among warps. So we shall take coarser granularities to acquire BFS, as showed in Figure 4. Like warp-based spanning once the threshold value of the thread is tested the domain of neighbors of node can be broadcast to all threads through shared memory so as to achieve the most adjacency spanning using the width of threadblock.

4. DYNAMICALLY SCHEDULING THREADBLOCK BY WORK-STEALING

In our implementation of spanning the large graph by work-stealing, a task pool (the data structures of graph node) is defined for each threadblock which tries to repeatedly get the task to perform from its own task pool’s tail. If the new task is created during the execution it is added to tail of its own task pool. If the threadblock fails to acquire a task, it checks a condition to see if all task has been done. If it is not, the threadblock tries to steal task from the other threadblock’s task queue. In its all execution the synchronization and memory operation plays an important part for right spanning.

4.1. Synchronization and Atomic Memory Operation

During stealing multiple threadblocks may access the same task pool. The central data structures should be able to synchronize all the concurrent threads. Surely the spinlock can be used to synchronize the threads. Nevertheless, it is very expensive to use the spinlock for scalable synchronization especially on graph processors. So it is better choice to make avail of lock-free technique [Maurice et al., 2008; Arora et al., 2001] Lock-freedom is a advance warranty for algorithms that presents that at any given time, at least one thread block will always makecomputational advance,regardlessofthe advancer statusofanyotherthreadblock. This implies that a thread block never has to wait for a lock to be released, so whichever thread block is scheduled, at least one thread block will always be able to finish its operation in a bounded amount of time. One common design method to make an algorithm lock-free is to take all the changes that need to be performed inmutualexclusiveandrewritethem sothat they canbe performed with only one atomic instruction.

It is limited only to make use of read and write operation for non-blocking synchronization. So atomic primitive was introduced into the instruction set of graphic processors. In the set of all atomic primitives, the Compare-And-Swap (CAS) operation is among the most powerful. The CAS is used to atomically change the value of the variable, if and only if the value of some variable is given as a parameter to some operation. For hardware supported CAS operation the memory bus needs to be locked to warrant that no other threadblock concurrently writes in the same memory location. All of these overhead is expensive and this leads the atomic operation to be many times slower than a normal read or write operation. Therefore, reducing the use of CAS operation primitives for our scalable spanning is advisable choice.

4.2. Algorithm of Work-stealing

A double-directed queue for task pools of threadblock is used for the algorithm. Tasks (the data structure of graph) are added or displaced from the tail of queue in a manner of Last-in-first-out (LIFO). Once task pool of threadblock is empty, the threadblock tries to steal from the head of the other threadblock’s queue. Expensive synchronization is not needed for threadblock’s access to the tail of its own queue. However several different threadblock might try to steal at the same time. Then the demanding synchronization is required for this case.

The data structure of double-directed queue is depicted in Figure 5. There are tasks and a head and tail pointer which points to the first task and the last task respectively in the data structure. In the head structure there are two domain: index and counter which can be used to avoid the ABA-problem [Damianet et al., 2010]. As soon as the head pointer has been moved to the beginning of the queue the counter is increased by one. When the head pointer is updated by two different threadblocks the ABA-problem can take place.

```c
struct head {
    2 unsigned int index;
    3 unsigned int count;
    4 }
struct queue {
    6 head head;
    7 unsigned int tail;
    8 task tasks[max];
    9 }
Queue queues[max_thread_blocks];
```

Figure 5. Data structures of graphs
Because each threadblock has its own queue array we have to allocate sufficient memory for as many queue arrays as the threadblocks. We cannot use the shared memory to store the queue arrays for the stealing among threadblocks.

Every new task can be added to the tail of the corresponding threadblock. As the tail is only updated by the own threadblock there is no need for any synchronization. The task is simply written to the array which the tail pointer points to and afterwards tail is incremented to point the next empty array.

The steal operation illustrated in Figure 6 tries to extract a half queue size (Danny et al., 2002) tasks from the victim queue. Because multiple stealers might steal tasks of victim queue at the same time and victim itself might pop task to perform it a synchronization primitive \texttt{atomicCAS()} is used to guarantee only one stealer succeeds in acquiring the tasks.

```
Steal(int size) {
    head, head, newhead, oldhead;
    task task, steal_task;
    oldhead=head;
    for (i=0;i<size;i++)
        if(tail<oldhead.index+i)
            return null;
    steal_task[i]=task[oldhead.index+i];
    newhead=oldhead;
    newhead.index=oldhead.index+size;
    if (atomicCAS(&head, oldhead, newhead))
        return steal_task;
    return null;
}
```

**Figure 6.** Stealing operation of work-stealing

The extracted task from the tail of queue is executed locally as showed in Figure 7. When comparing the pointer between head and tail there are three different cases:

- Tail is strictly larger than the copied \texttt{oldhead} on line 9. As any new steal operation that is called can find that the task which the tail pointer pointed to is out of stolen domain it will not try to steal the task.
- Tail is strictly smaller than the copied \texttt{oldhead} on line 9. This case may occur when a stealer has extracted the last task in some victim queue before victim performs the \texttt{pop} operation of Figure 8. For this case, both tail and head are changed to the beginning queue array in lines 11 – 14 and line 18 in order to provide memory for pushing the new tasks locally. Once the local queue array is empty the operation return null in line 19.
- Tail is equal to the copied \texttt{oldhead} on line 9. This indicates there is only one task in the local queue. For this case we make use of primitive \texttt{atomicCAS()} to move \texttt{head} to the beginning of array on lines 15-17.

```
pop() {
    head oldhead, newhead, head;
    unsigned int oldtail, tail;
    task task;
    if (tail==0) return null;
    tail--;
    task = task[tail];
    oldhead = head;
    if(tail>oldhead.index)
        return task;
    oldtail=oldhead.index;
    return task;
    oldtail=tail;
    tail=0;
    newhead.index=0;
    newhead.counter=newhead.counter+1;
    if(oldtail==oldhead.index)
        return task;
    head=newhead;
    return null;
}
```

**Figure 7.** Pop operation of work-stealing
Figure 8 shows the main kernel that implements the work-stealing scheduler. From line 4 the execution path of thread 0 of every threadblock forks to pop task or steal tasks. This execution converges in line 13 when all threads in the threadblock before actually executing the SIMD task on line 12.

1 steal_scheduler()
2 {int* local_queue=queues[blockIdx.x];
3   while (1) {
4     if (threadIdx.x==0)
5       if (pop()==null)
6         {int* victim_queue=queues[rand()];
7           int steal_size= length(victim_queue)/2;
8           steal(steal_size)
9           steal(steal_size)
10       }
11   _synthreads();
12   Task_nodes_neighbors();
13   _synthreads();
14   if (terminate) break;
15 }}

Figure 8. Work-stealing scheduler

5. EXPERIMENTAL EVALUATION

In this section we evaluate the performance in two aspects. First, we investigate the effectiveness of our spanning mechanism which we projected to work out workload imbalance. Second, we study how different architecture interacts with breadth first spanning.

5.1. Effect of Spanning Methods

In the aforementioned sections, we introduced combined granularities and work-stealing mechanism to avoid workload imbalance for efficient breadth first spanning. In this section, we evaluate these schemes by graph instances from different sources. Table 1 is the different evaluation platform used for our experiment. For all fair contrast speedup is measured against single-threaded execution time of Intel Xeon E5345 illustrated in Table 1. The corresponding baseline implementation adopts algorithm in (Virat et al., 2010) because it has recently been shown to be the fastest multicore graph spanning implementation. The codes of CPU BFS and GPU BFS were compiled using gcc with the parameters `-O3 -m32` flags and nvcc with –O3 flags respectively.

For our experimental dataset characteristics of graph instances are simply illustrated in Table 2. Among them Memetracker and Wiki-talk derive from the public real world dataset collection (SNAP, 2012). Both have non uniform degree distribution among nodes. R-MAT graph is produced by the GTgraph suite (David, 2006) with a few high degree nodes and many low degree nodes like real-world large-scale networks. Every node of Random graph has a randomly chosen degrees indicating somewhat regularity. The remaining graph instances come from the 10th DIMACS implementation challenge (David et al., 2006).

| Table 1. Parameters Of Different Systems |
|------------------|------------------|------------------|------------------|------------------|
|                  | Intel Xeon E5345 | Intel Xeon X5580 | Sun UltraSparc T2 | Nvidia Tesla GTX260 | Nvidia Tesla GTX275 |
| sockets           | 2                | 2                | 4                | 1                | 1                |
| Cores             | 4                | 4                | 8                | 24               | 30               |
| SMT/core          | 1                | 2                | 8                | 16               | 16               |
| cache             | 8MB              | 8MB              | 4MB              | -                | -                |
| Clock freq        | 2.3Ghz           | 3.2Ghz           | 1.6G             | 1.2Ghz           | 1.4Ghz           |
| memory            | 32G              | 16G              | 128GB            | 1GB              | 1GB              |
Table 2. Characteristics of Graph Instances

<table>
<thead>
<tr>
<th>Graph name</th>
<th>nodes</th>
<th>edges</th>
<th>shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memetracker (SNAP, 2012)</td>
<td>96 million</td>
<td>418 million</td>
<td>irregular</td>
</tr>
<tr>
<td>Wiki-talk (SNAP, 2012)</td>
<td>2394285</td>
<td>5021410</td>
<td>irregular</td>
</tr>
<tr>
<td>R-MAT (David et al., 2006)</td>
<td>200 million</td>
<td>1 billion</td>
<td>irregular</td>
</tr>
<tr>
<td>Random (David et al., 2009)</td>
<td>64 million</td>
<td>512 million</td>
<td>regular</td>
</tr>
<tr>
<td>nlpkkt160 (DIMACS, 2012)</td>
<td>8.3 million</td>
<td>221.2 million</td>
<td>irregular</td>
</tr>
<tr>
<td>Grid7pt300 (DIMACS, 2012)</td>
<td>27 million</td>
<td>188.5 million</td>
<td>irregular</td>
</tr>
</tbody>
</table>

Figure 9. Effect of optimizations on different benchmark

Figure 9 displays the comparable performance difference of benchmark in Table 2 when combining work-stealing with different parallel granularities. For all irregular graphs in Figure 9 we find that the more the number of optimized levels the more the performance improves. This reveals advantage of our combined policy for irregular graphs instance. On the other hand, in contrast, dynamic work-stealing allows for further speedup, especially for severely imbalanced graph instance such as nlpkkt160, nevertheless, this dynamic policy yields more overhead than benefit for regular instance Random.

Figure 10. Effect of average degrees of graph

The number of edges of node inside graph has an effect on performance of BFS. Figure 10 shows the result for different average degrees of R-MAT graphs by the use of GTtool suite (David et al., 2006). When the average degrees are less than 32 edges the spanning policy is mainly fine-grained. So in this interval the speedup is improved little by little. Roughly at intervals (Daniele, et al., 2008; SNAP, 2012) the coarse-grained spanning is adopted the performance is ideally perfect. Once the average degrees are greater than 1024 edges because the workload imbalance becomes more severe due to the increased level of degree skew the comparable speedup maintains. However, the speedup decreases slightly because of the overhead of dynamic scheduling of threads blocks.

The chart of Figure 11 shows the scalability of different policies with input size by the transversal of generated R-MAT graphs. The relative speedup for the small graph examples in all of GPU-based spanning mechanism is low because of CPU’s higher cache hit rates, but for the larger graphs instances the plot visibly shows superior speedup.
5.2. Architecture-affected Performance

As shows in previous subsection, the GPU spanning algorithm significantly outperform their CPU counterparts. We now investigate which GPU architecture characteristics produce this result.

As it is known to all, the following architectural traits account for performance improvements delivered by the execution of GPU CUDA codes:

- A great number of threads perform to hide memory latency.
- Many physical steam processors enables massive parallel execution.
- The speedup of GPU makes full use of the directly-attached GDDR3 memory, which boasts higher bandwidth and lower latency than FB-DIMM based main memory.
- GPU does not consider coherence traffic to consume massive bandwidth.

To explore the architectural characteristics of BFS, we compare performance between GPU and SMT machine revealed in Figure 12. The presented single thread speedup in T1 bar is very low compared to single-threaded execution of Intel Xeon E5345. It indicates that the out-of-order execution CPU successfully makes full use of the high degree of instruction level parallel. As soon as a increased number of threads has been put to use, the SMT machine outperforms the out-of-order machine due to higher throughput as figured in 64-threads T64 bar of Figure 13. But more important, exploring more sockets of the SMT machine such as four sockets for T256 does not surely manifest better performance. This demonstrates that the communication overhead across chip boundaries is the predominant limiting factor for scalable spanning on SMT machine.

To further evaluate the effect of bandwidth exploitation of GPU some traits can be found in Figure 13. The related experiment involves varying the number of warps executing in each SM when performing BFS. From the chart we find that the speedup increasement becomes milder at the nine warps size. It reveals performance gain is limited by the memory bandwidth after this warp size.
To sum up, there exists massive parallelism in BFS, prevalingly in memory accesses rather than in computation. This parallelism can be optimized by the exploration of out-of-order CPU, SMT CPU or SIMT GPU. Regardless of architectural type chosen BFS needs to consume much memory bandwidth for its peak performance. GPU is indicative of largest bandwidth of all machines and consequently shows the best performance.

There is no enough memory in GPU for the large graphs which is bottleneck for graph spanning. So it would be attractive to consider how to span the larger graphs through directly attaching FB-DIMM based main memory rather than GDDR memory. This could accommodate larger graphs instances at the expense of a reduction in memory bandwidth. In such a deployment the GPU performance decreases, as per decreased memory bandwidth, although the massive parallel of GPU might make full use of bandwidth efficiently. Similarly the advantage of CPU’s cache will also diminish as graphs scale larger.

6. RELATED WORK

So far there exists numerous implementation of breadth first spanning graph aiming at different computers architecture including distributed memory supercomputers(Andyet et al., 2005), shared memory supercomputers(David et al., 2006), multi-core SMP computers(David et al., 2011; Virat et al., 2010; Masahiro et al., 2012), heterogeneous multi-core computers(Danieleet al., 2008) and GPUs(Pawan et al., 2007; Honget al., 2011; Duaneet al., 2012; Lijuannet al., 2010; Deng, 2009).

For scalable graphs distributed processing is often considered to be challenging due to the natural irregularity of underlying graph. However there is the case that a single machine’s memory can’t be enough for the graph. So distributed processing is necessary sometimes. At present there are a few frameworks or libraries which aim to simplify graph processing in distributed environment. Pregel(Grzegorzet al., 2010) is a distributed framework encapsulating message passing and fault-tolerance like the MapReduce framework. Of course conventional graph spanning should be expressed in a description suitable for MapReduce.

Supercomputers indicate its multi-threads and high memory bandwidth or capacity. Graph algorithm including BFS certainly showed impressive performance on these machines(Andyet al., 2005; David et al., 2006). Unfortunately, such machines are rare and costly.

For efficient parallel graph spanning work-stealing scheme can dynamically balance workload among processors. In the parallel language cilk++ Chareles Leiserson et.al.(Charleset al., 2010) achieved near-perfect linear speedup in the number of processing cores by the use of underlying work-stealing scheduler as long as the machine had sufficient memory bandwidth. Moreover X10 work-stealing framework(Guojinget al., 2008) further adopted global termination detection and phased computation to efficiently implement graphs algorithms. In addition, in (Masahiro et al., 2012) a backtracking work-stealing framework featuring on-demand concurrency was proposed to perform probabilistically balanced divide-and-conquer graphs spanning by authors.

7. CONCLUSION

Parallel graph spanning inside GPU suffers severe performance degradation for many real-world graph instances. For working out this issue we come up with a perfect solution by combining work-stealing with multiple granularities, in which different grain may tailor to different degrees of nodes and work-stealing further balances workload among stream multi-processors. In addition by warp-grained spanning we have reduced significant branch divergence and increased memory merge accesses.

By utilizing combined graph spanning method we have achieved about 10x speedup over baseline implementation for many kinds of graphs instances. Furthermore our experiments have confirmed combined
graph spanning method can achieve more speedup at larger degrees and more nodes. Finally our relevant tests demonstrates that BFS based on GPU outperforms BFS on other architectures is due largely to substantially larger memory bandwidth.

Acknowledgements

This work was supported by the Scientific Research Project (NO. 14B104) of Department of Education of Hunan Province, china and Higher Scientific Research Project of Hunan Province of China(15C0779).

REFERENCES


