Parallel Processing Letters

SOME GPU ALGORITHMS FOR GRAPH CONNECTED COMPONENTS AND SPANNING TREE

Jyothish Soman, Kishore Kothapalli, and P J Narayanan
International Institute of Information Technology, Gachibowli
Hyderabad, 500032, India

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

Graphics Processing Units (GPU) are application specific accelerators which provide high performance to cost ratio and are widely available and used, hence places them as a ubiquitous accelerator. A computing paradigm based on the same is the general purpose computing on the GPU (GPGPU) model. The GPU due to its graphics lineage is better suited for the data-parallel, data-regular algorithms. The hardware architecture of the GPU is not suitable for the data parallel but data irregular algorithms such as graph connected components and list ranking.

In this paper, we present results that show how to use GPUs efficiently for graph algorithms which are known to have irregular data access patterns. We consider two fundamental graph problems: finding the connected components and finding a spanning tree. These two problems find applications in several graph theoretical problems. In this paper we arrive at efficient GPU implementations for the above two problems. The algorithms focus on minimising irregularity at both algorithmic and implementation level. Our implementation achieves a speedup of 11-16 times over a corresponding best sequential implementation.

Keywords: GPU,Graph algorithm, parallel, connected components

1. Introduction

The advent of General Purpose Computing on the GPU, also called as GPGPU, has placed GPUs as a viable general purpose co-processor. The architecture of GPUs fits the data parallel computing model best where a common processing kernel acts on a large data set. Several general purpose data parallel applications [5] and higher-order primitives such as parallel prefix sum (scan), reduction, and sorting [8, 12, 23] have been developed on the GPU in recent years. From all these applications, it can be observed that GPUs are more suited for applications that have a high arithmetic intensity and regular data access patterns. However, there are several important classes of applications which have either a low arithmetic intensity, or irregular data access patterns, or both. Examples include list ranking, an important primitive for parallel computing [21], histogram generation, and several graph algorithms [11, 26, 25]. The suitability of GPUs for such applications is still a subject of

study. Very recently, the list ranking problem on GPUs is addressed in [21] where a speed up of 10 is reported with respect to the CPU for a list of 64 million nodes.

Graphs are an important data structure in Computer Science because of their ability to model several problems. Some of the fundamental graph problems are graph traversals, graph connectivity, and finding a spanning tree of a given graph.

In this paper, we study the fundamental graph problem of finding connected components of a graph on the GPU. It finds application in several other graph problems such as bi-connected components, ear decomposition, and the like. We modify our method to generate a spanning tree of the graph. Our implementation of connected components achieves a speed-up of 10-13 over the best sequential CPU implementation and are highly scalable. Our work can thus lead to efficient implementations of other important graph algorithms on GPUs.

1.1. *Related Work*

There have been several PRAM algorithms for the graph connected components problem. Hirschberg et al. [13, 14] discuss a connected components algorithm that works in $O(\log^2 n)$ time using $O(n^2)$ operations. However, the input representation has to be in the adjacency matrix format, which is a limitation for large sparse graphs. For sparse graphs, Shiloach and Vishkin [24] presented an algorithm that runs in $O(\log n)$ time using $O((m+n)\log n)$ operations on an arbitrary CRCW PRAM model. The input for this algorithm can be in the form of an arbitrary edge list. A similar algorithm is presented by Awerbuch and Shiloach in [1]. However, it should be noted that the PRAM model is a purely algorithmic model and ignores several factors such as the memory hierarchy, communication latency and scheduling, among others. Hence, PRAM algorithms may not immediately fit novel architectures such as the GPU.

In [10], the authors presented a wide range of optimizations of popular PRAM algorithms for connected components of a graph along with empirical results on a Connection Machine 2 and a Cray YMP/C90. Their work includes optimizations for the Shiloach-Vishkin algorithm [24] and the Awerbuch-Shiloach algorithm [1]. Though the architectures on which they reported empirical results are dated, many algorithmic observations and inferences presented are relevant to our work also. Another attempt at implementing connectivity algorithms was by Hsu et al [15]. Their method shows good speedups on graphs that can be partitioned well. For random graphs, no major speedups were reported.

Graph connectivity on symmetric multiprocessors (SMP) is studied by Bader and Cong in [4]. Their idea is to give each processor a stub spanning tree to which unvisited nodes may be added iteratively. In a recent work, Harish and Narayanan [11] have implemented a modified BFS-style graph traversal on the GPU. Their implementation works with a sorted edge list as the input. There have also been other efforts to implement parallel BFS on different architectures (cf. [22, 27, 9]).

2. GPU Connected Components Algorithm

Connected components is considered an irregular memory access algorithm (irregular algorithm), which is not a good fit for the GPU computational model which relies heavily on regularity of memory access. The CUDA Programming Guide[20] provides a detailed description of the effects of irregular and regular memory access on GPU performance.

The focus of any algorithm designed for the GPU relies on regular/coalesced memory accesses and increasing computation, focusing on data movement in the shared memory. The requirements for connected components and GPU computational model are thus orthogonal to each other. Thus mapping the connected components algorithm on to the GPU is non trivial.

The work presented here tries to reduce irregular memory access based on the guidelines of algorithms of [1, 24, 10]. While designing the algorithm, the following principles were considered:

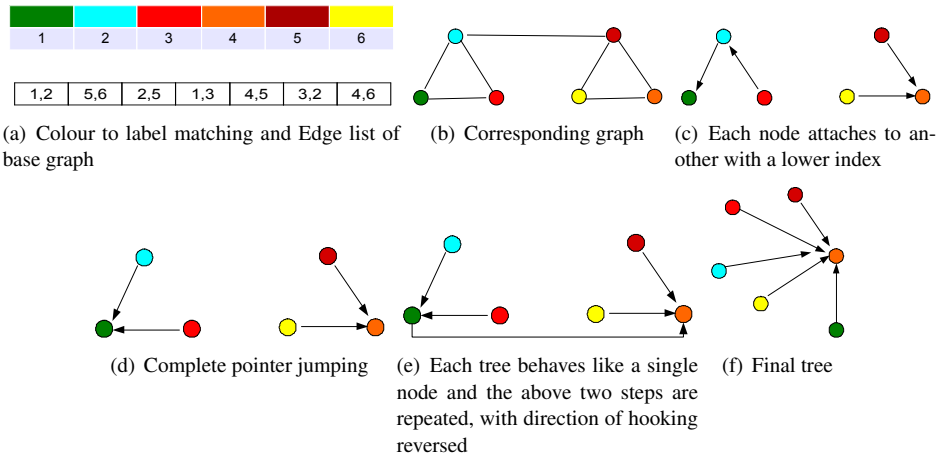
- Removing atomic operations
- Reducing the overhead caused by the 64 bit reads for the end points of the edges
- Allowing partial results from previous iterations to be carried forward to reduce redundant computations
- Minimising the number of times an edge makes an active contribution to the algorithm

We have focused on developing an algorithm that reduces irregular memory accesses, and allows transfer of information from one iteration to the next, hence streamlining the algorithm.

The Shiloach-Vishkin algorithm as proposed [24] may not be quite suitable on modern architectures such as the GPU. One of the reasons for this is the excessive number of irregular reads during the grafting and the pointer jumping steps. An important point to be noted here is that the average number of iterations over which an edge is active is large. Also, if an edge has taken part in a grafting step once, it is not necessary that it cannot participate in another iteration of grafting. Hence, we require a method that requires an edge to succesfully participate in grafting at-most once. The number of reads of the full edgelist should be minimal, and the pointer jumping step is further optimized. In this section, we devise ways to handle the given constraints, thereby improving the performance on the GPU.

Our adaptation follows three main steps: 1) Hooking 2) Pointer Jumping 3) Graph Contraction. Hooking is the time intensive operation in the algorithm. It tries to connect elements of the same component together. Pointer jumping tries to find the root node for each component after the hooking step. The fundamental building blocks of our algorithm are as follows:

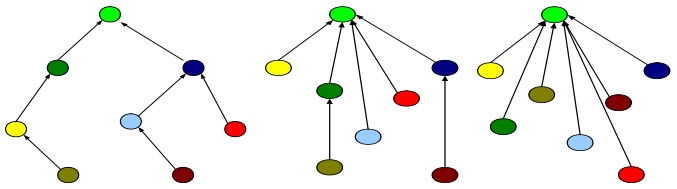
(1) **Hooking:** Hooking is the process of selecting a vertex as the parent of another. For each edge, if the two ends lie in different components, it tries to connect the root nodes of the two components. The orientation of the connection is based upon the labels of the indexes. The orientation is varied across iterations. As the hooking process is ran-



(a) Colour to label matching and Edge list of base graph (b) Corresponding graph (c) Each node attaches to another with a lower index

(d) Complete pointer jumping (e) Each tree behaves like a single node and the above two steps are repeated, with direction of hooking reversed (f) Final tree

Fig. 1. A simple example of our algorithm is shown in the Figures. The base graph goes through the iterations as shown in the Figures (c) through (f). The first graft (c), Pointer jumping (d), second iteration follows the same method as shown in (e) and (f), where all the nodes in the graph are connect to a single node



(a) Initial tree (b) Partial pointer jumping (c) Complete pointer jumping

Fig. 2. Effect of different degrees of pointer jumping

domized, the sensitivity to vertex labels is reduced. We use the alternating orientation hooking process as mentioned in HY1 of [10]. In even iterations, the node with lower label selects the node with higher label as its parent and the reverse happens in odd iterations.

(2) **Multilevel pointer jumping:** Instead of one step of pointer jumping as originally proposed in the Shiloach-Vishkin algorithm [24], we perform complete pointer jumping. In complete pointer jumping, each tree is reduced to a star in every iteration. Figure 2 illustrates the difference. We add that the HY1 algorithm of Greiner [10] also uses complete pointer jumping.

Further, when we use complete pointer jumping, notice that nodes in a tree are at only two levels: root node and internal nodes or leaf nodes. Also, after a tree is grafted onto another tree, only the root nodes in one tree have to perform pointer jumping. This helps in improving the performance on the GPU.

Though the number of iterations required remains the same as Shiloach Vishkin

[24] and Awerbuch Shiloach [1], we observe that the number of read write operations decrease significantly.

```
Begin
    Initialize Parent[i] = i;
    while All edges are not marked do
        for each edge (u, v)
            if edge is unmarked and Parent[u] ≠ Parent[v]
                max=max{Parent[u], Parent[v]};
                min=min{Parent[u], Parent[v]};
                if iteration is even Parent[min] = max;
                else Parent[max] = min;
            else Mark edge (u, v);
        end-for
    end-while
    Multilevel_pointer_jumping();
end.
```

Fig. 3. Algorithm for Connected Components

(3) **Graph contraction using edge hiding:** Explicit graph contraction requires large amount of data movement across the global memory of the GPU, which is a costly operation. The time taken by such a process only increases the run time of the algorithm. Hence an implicit method of hiding edges from the graph is used. As edges are only active in the hooking stage, edges which are known to be in the same component are inactivated in the next stage of hooking.

The complete algorithm is presented in Algorithm 3. Figure 1 provides an illustration of the running of our algorithm. Our algorithm can be seen as a modification of HY1 of [10] for the GPU. The algorithm presented here runs in worst case of $O(\log n)$ iterations, and $O(\log^2 n)$ time, doing $O((m+n)\log n)$ work on a PRAM model. The pointer jumping step takes a total of $O(n \log n)$ of work, using $O(\log^2 n)$ time.

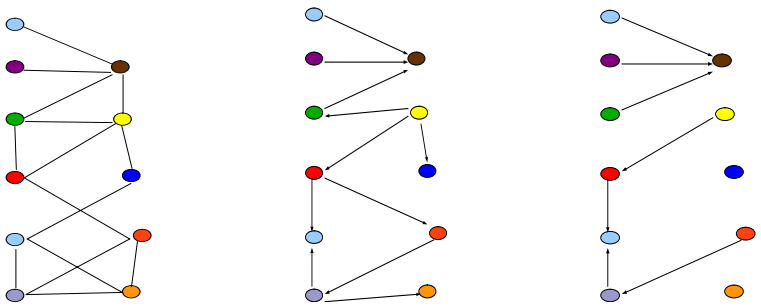
2.1. *Modifying the PRAM algorithm for Shared memory*

The algorithm presented in the previous section can be a practical PRAM algorithm. The behaviour of the GPU though, is slightly skewed from the PRAM model. The difference, at thread level, is due to the grouping of threads into blocks of cooperating threads and the lack of global synchronisation across blocks. Additionally, each thread block has access to limited shared memory. Hence further optimisations can be presented for the same.

In the hooking phase, our algorithm shows variable properties to sorted and unsorted edge list data. For sorted edge list, additional space is provided in the shared memory. In the hooking phase, instead of directly writing data in the global memory, we write to

the shared memory. The data in the shared memory is filtered, so that only one write to a memory location is allowed per thread block. This is done by performing a de-duplication and redundant data removal. Thus each thread block only passes unique information to the global memory. To improve coalescing, and to reduce occupancy of warps on the SM, only the first N threads, where N is the number of unique data are allowed to copy data to the global memory.

As can be seen, the same cannot be extended to unsorted data. For unsorted data, the probability of duplication within a thread block is low. We suggest modifications by mapping the unsorted edge list to a sorted edge list problem. It can be noted that, in the hooking phase of our algorithm, each edge tries to hook its end points. The number of writes is thus equal to m, m=$|E'|$, E' is the number of active edges in the iteration. Out of the m writes, only $k \leq n$, n=$|V|$ will succeed. This step is highly irregular given an unsorted edge list. We can see that for each node, its neighbours try to hook to it. We suggest that if we store some of the nodes which try to connect to it, we can form an adjacency list. To perform the given task, we suggest that adding an additional set of dummy vertices $D(v)$, to which nodes can hook to, then writes are spread across a larger set of destinations. Hence, we will be able to form an adjacency list, which is a subset of the real adjacency list of the graph. We argue that $D(v)$ can then be used for optimisations we have suggested for sorted edge list can hence be used for the new list. An example iteration is shown in figure 2.1, the original Graph G shown in sub figure a. A subsection of the graph G, G' is taken at random, as shown in sub figure b. In the next step, our hooking algorithm is run on G', as shown in sub figure c.



(a) Original Graph G

(b) Graph G'

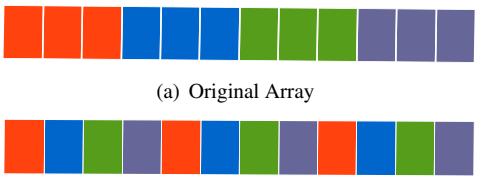
(c) Sample hooking using G'

Fig. 4. A random sampling of a graph G (a) and the resultant graph G' (b) A possible hooking caused by G'

Our algorithm when implemented on the new adjacency list will have access to a stronger data structure as compared to the sorted edgelist, as the degree of each node in the new graph is constant. The major optimisation we suggest is related to memory access. The GPU is better suited for intra warp-inter thread memory access locality. If each thread in a warp individually access continuous memory locations, then the coalescing is weak. In comparison, if all threads in a warp put together, access continuous memory locations,

the coalescing effect is better found. Hence, to make use of inter thread locality instead of intra thread locality, for each set of 16 nodes, their adjacency list is interleaved. The stride is hence kept at a constant value of 16 for each thread. An example of the same, with 4 nodes such that $|D(v)| = 3$, is shown in figure 5. Each colour represents the adjacency list of a vertex.

It can be seen that each iteration takes $\Theta(m+n)$ order of additional work and $\Theta(\log(n))$ time to complete. To maintain the algorithmic complexity, and for practical purposes, the number of iterations for our optimisation for an unsorted list is kept constant. The number of edges reduce over iterations substantially, hence using this method exhaustively causes performance degradation. We present that $|D(v)|$=4 is sufficient in practise.



(a) Original Array

(b) Interleaved Array

Fig. 5. The interleaving of an array, each colour represents adjacency list of a node

3. Spanning Tree on the GPU

A direct application of the modified algorithm is to find the spanning tree of a given graph. With additional, low overhead, book keeping, we can use the above algorithm to find the spanning tree of a graph. The modifications can be explained as below:

Mark edge which caused a hook: A simple bookkeeping step which differentiates between edges that caused a hook and other edges, can give the spanning tree for a given graph. That is, each edge which connects two different trees tries to hook the two trees together. For a given tree only one edge is allowed to hook another tree to it. This is performed by competitively selecting an edge, wherein multiple edge tries to hook a tree onto a given tree, but only one succeeds. This edge is stored and then used to hook the two trees.

Coalesced termination check: If more than two subtrees occur in the induced graph, then there will exist two indexes, i and i + 1 such that nodes i and i + 1 are in different subtrees. Hence their parents will be different. Hence if such an index exists, then the algorithm should not terminate. The reads will be fully coalesced. And since the same value is being written to the same memory space, the write will be optimized.

3.1. *Spanning Forest*

The spanning forest algorithm is an extension of the spanning tree algorithm, wherein the termination is based on whether there exists any edge such that its ends are in different

components. The checks are done while selecting edges for hooking, which is inherently part of the connected components algorithm.

4. Implementation Details

The implementation on the GPU for the algorithm given in Figure 3 has scope for further implementation optimisation, which we describe in this section.

4.1. *Input Representation*

An edge list, seen as an arbitrarily ordered array of edges, is a natural choice for implementing our algorithm. When a connected components algorithm is used as a subroutine for another graph problem, such as finding the bi-connected components of a graph in parallel (cf. [16, Chapter 5]), an algorithm that can work with a random edge list is important. Further, compared to other standard representations such as adjacency lists, an edge list representation is not any stronger. In the GPU context, however, an edge list representation does not support information locality.

4.2. *GPU Implementation*

Each edge (u, v) is represented by a 3-triple $\langle u, v, \texttt{state}\rangle$. For an edge (u, v), the variable `state` shows whether u and v belong to the same component or not. Each edge is stored as a 64-bit integer. The `state` flag is stored in a separate array. This is done so that `state` information can be read in a fully coalesced manner. In the hooking step, the edgelist needs to be read, for each half warp the edges can be read in a single memory transaction.

Global memory bound GPU algorithms generally focus on two features, we have managed them as given below:

- **Reducing uncoalesced reads:** In the pointer jumping phase of the algorithm, we can identify which nodes have to participate in this operation by distinguishing between leaf nodes and internal nodes. See also Figure 2. This allows one to reduce the number of uncoalesced reads by an early case distinction.

  But the hooking stage does not support any functional method which can be used to reduce uncoalesced reads. Each iteration stores whether an edge was found to connect vertices in the same component, if so it was marked inactive, hence would not participate in future iterations. This state information directs the algorithm whether or not to perform a hooking operation involving the given edge. This state information can be read by a thread block of 512 threads in a single transaction of 64 bytes. Also in the first iteration of the algorithm, nodes hook to their neighbours, rather than the parents of the neighbours, this reduced irregular reads and writes appreciably.
- **Reducing thread divergence penalty:** Thread serialization is present in both the major components of the algorithm, namely pointer jumping and hooking. It is the second major bottleneck for performance. To reduce the effect of thread divergence, work load was increased on a single branch of the divergent branch. The other branches were not

required to do any other operations. Thus the ratio of run time of the two divergent threads is large. Hence thread serialization penalty was minimized. Another factor to be noticed is that, given two warps, each with different number of divergent threads there was little difference in the SIMD code running on the SMs for a kernel.

The algorithm presented here can be categorised as an unordered algorithm [18]. Hence atomic operations are not required in both hooking and pointer jumping, as random execution is supported by the algorithm.

The relevant kernels in the connected component algorithm are the hooking kernel and the pointer jumping kernels. Pointer jumping is divided into two kernels to assist global synchronization. The number of threads per block is kept at 512 to maximize the GPU utility. This configuration is hardware specific and can be considered to be selected for Tesla S1070/C1060.

A practical issue comes up in the first step of the multistep pointer jumping, which requires multiple iterations. As all threads cannot be active at the same time, the method requires external synchronisation. This can add an additional overhead in terms of both algorithmic complexity as well as runtime, as data needs to be loaded before each iteration from the global memory of the device to the SMs. Thus increasing work and time taken. But as our method only requires nodes which are not connected to the root node to jump in the next iteration, the current state information of the nodes can be stored so that redundant reads do not take place. As each thread block can retrieve relevant state information in a small number of coalesced memory transactions, the overhead added is hidden by the uncoalesced reads/writes caused by pointer jumping. Also, as reading the parent information only duplicates the effort in the previous iteration, the algorithmic complexity is not changed in practice.

5. Experimental Environment and Results

In this section, we report the results of our experiments. The experiments were run on the following systems:

- **CPU:** An Intel Core i7 920, with 8 MB cache, 4 GB RAM and a 4.8 GT/s Quick path interface, with maximum memory bandwidth of 25 GB/s.
- **GPU:** A Tesla C1060 which is one quarter of a Tesla S1070 computing system with 4 GB memory and 102 GB/s memory bandwidth. It is attached to an Intel Core i7 CPU, running CUDA Toolkit/SDK version 2.2. [20].

5.1. *Experimental Datasets*

We have conducted our experiments on four different data sets. These are described below.

- Random graphs: We used random graphs from the $\mathcal{G}(n, m)$ family generated using GTgraph generator suite [3]. Random graphs serve to study the effect of unstructured topology on the runtime of our algorithm.

- Synthetic social networks: Social networks are gaining lot of research attention in recent years due to their large application base. Hence, we studied our implementation on social networks focussing on the effect of the presence of high degree nodes and clusters. These instances are generated using the R-MAT generator [6], an implementation of which is present in GTGraph.
- Real world instances: The above two datasets are synthetically generated. Hence, we also considered real world instances from the repository maintained by Leskovec, which have also been used in [2],[17].
- k Regular geometric graphs : Finally, to study the effect of regular topology on the runtime of our algorithm, we considered k-regular geometric graphs. These graphs are constructed by choosing n nodes over a two dimensional Cartesian space, and for each node, its k nearest neighbors are chosen as neighbors. The k nearest neighbors are found using the *ANN: Approximate Nearest Neighbors package* [19]. In our experiments, we used $k = 6$.

5.2. *Experimental Results*

We compare the performance of our implementation to that of the best known sequential algorithm for the same problem. In the case of connected components, a good choice for the best known sequential algorithm is the depth first search of a graph [7, 10]. A speedup of 9-12 compared to a sequential (single-core implementation) implementation of DFS is noted. Figure 6 shows a comparison of DFS with our algorithm on social networks of size 1 million to 7 million and of degree 5. In Figure 6, the label DFS_CPU refers to a single-threaded implementation of DFS on an Intel Core i7 CPU, and the label Connected_GPU refers to the implementation of our connected components algorithm on one quarter of a Tesla GPU. Connected_CPU shows an 8 threaded implementation, on an Intel i7 CPU, of the same algorithm.

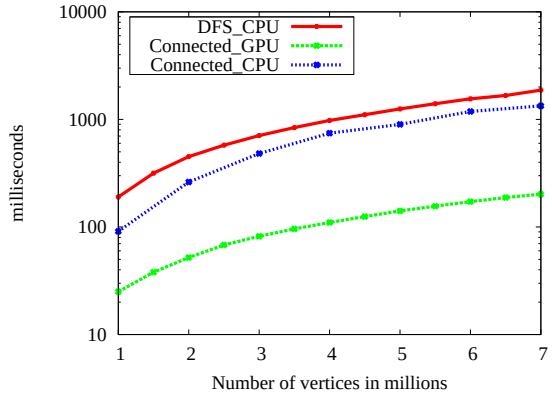


Fig. 6. Comparison of connected components on GPU versus DFS on CPU, with constant degree of 5.

We now move to comparison of our implementation for various instances of random matrices and scale-free networks. We considered three different ways to compare our implementations. For each of the graph classes we used, we kept the average degree fixed and varied the number of vertices. The results of this comparison are plotted in Figure 7. We also compared by keeping the number of vertices fixed and varying the number of edges. The results are shown in Figure 8. Finally, we kept the number of vertices fixed and varied the number of edges. The results of this comparison are show in Figure 9. In all these figures, the label DFS_CPU refers to the single-core implementation of DFS on a CPU. The label Connected_GPU refers to our implementation of the connected components algorithm on the GPU. SV_GPU refers to Shiloach Vishkin Algorithm on the GPU[24]. Spanning_GPU refers to our spanning tree algorithm.

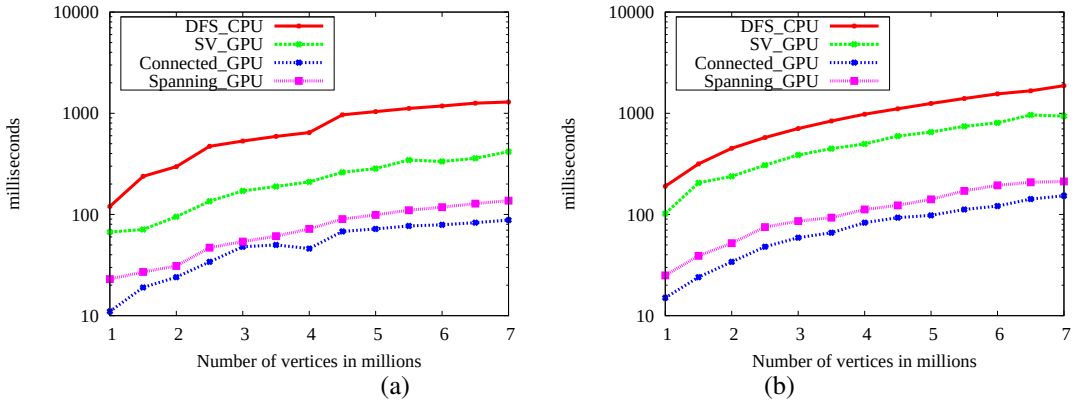


Fig. 7. Connected Components with constant average degree of 5 with (a)R-MAT and (b) Random

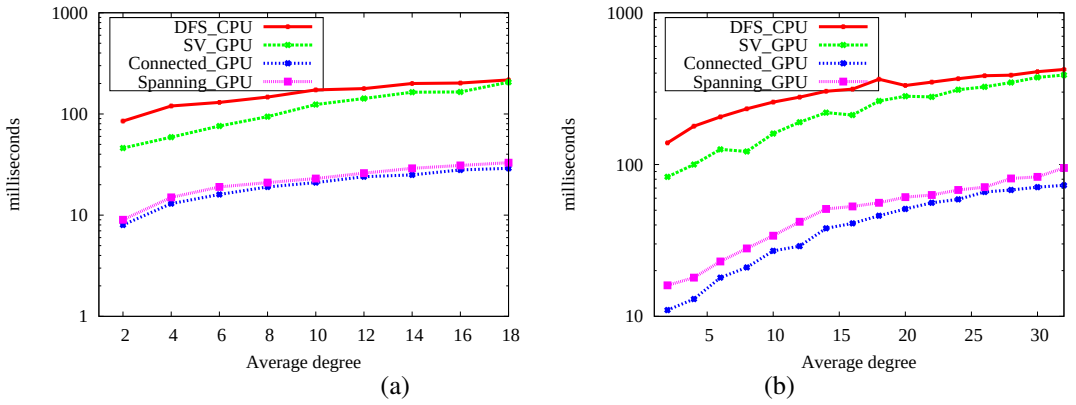


Fig. 8. Connected Components with constant number of vertices (1M) with (a)R-MAT and (b) Random

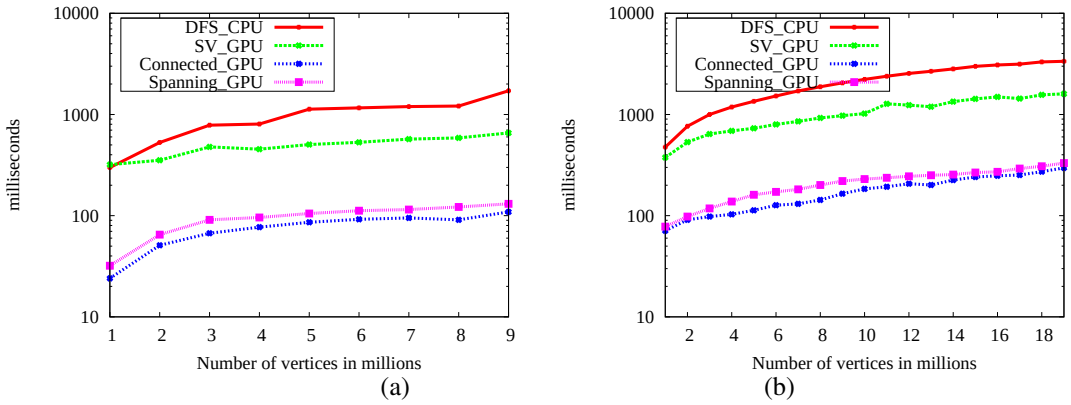


Fig. 9. Connected Components with constant number of edges 15M with (a)R-MAT and (b) Random

Some observation about the plots in Figures 7–9 are given below. In almost all cases our implementation achieves a speed-up of about 10 compared to DFS_CPU. The figures clearly show that the algorithm performs well in low diamter graphs. Also in graphs with higher diameters, the number of iterations was larger than low diameter graphs. The speedup due to multilevel pointer jumping was more evident in such graphs. A rough comparison is given in the Table 10(c).

To study the scalability of our algorithm, we considered another set of experiments where large graphs are taken as test cases. The results of these experiments are given in Table 10(b). It can be observed from Table 10(b) that even for a graph of 10 million vertices and 100 million edges, our algorithm can finish in less than a second. Though the results show practicality of the solution, we are bottlenecked by limited memory of a GPU system.

5.3. *Real World Instances*

In this section, we consider experiments on real-world instances obtained from the repository maintained by Jure Leskovec [2]. These instances have sizes that vary from 1 million vertices to 5 million vertices. We ran our GPU connected components algorithm on these instances and the results are shown in Table 10(a).

6. Comparison With Existing Work

One of the ways to consider our work is to study the ability of GPUs, which are ubiquitous and easily available, to efficiently implement graph algorithms. Hence, we mainly focused on comparing our run-time with respect to the best known CPU implementation, that is DFS in this case.

However, one has to also consider the efficacy of our implementation with respect to popular existing parallel implementations for finding the connected components of a graph. To this end, we make the following observations.

Data set	# vertices, #edges (in M)	Run Time (in ms)
Live journal	4.8, 69	191
Wiki Talk	2.4, 5	11
Citation n/w	3.7, 16.5	116
Road Networks		
California	2, 5.5	24
Pennsylvania	1.0, 3.0	13
Texas	1.4, 3.8	16

(a) Run time of our algorithm on various real-world instances.

Number of Edges in M	Time taken in ms
20	187
30	274
40	312
50	392
60	413
80	547
100	668

(b) Performance our algorithm on a sorted edge list for large random graphs, number of vertices 10 M

Number of vertices (in million)	Random Graph (ms)	Social Networks (ms)	6-regular graphs (ms)
1	15	11	19
2	34	25	45
3	52	43	71
4	83	49	103
5	109	71	142
6	132	81	167
7	148	90	191

(c) Comparison of our algorithm on the GPU with different family of graphs. The average degree of the graphs is fixed at 6 in all cases for these experiments.

Fig. 10. Comparison of our algorithm on the GPU across different datasets

Bader and Cong's SMP implementation [4] were on dated systems. Hence a direct comparison is not possible. The speedups they achieved over sequential is 4.5-5.5 with 8 processors. Theoretically, GPU thus provides a speedup of 2-3 over an 8 threaded Core i7.

Another widely accepted and popular parallel implementation is that of parallel Boost Graph Library [9]. Parallel BGL is implemented on a cluster system. We do note significant speedup of our implementation with respect to reported results for parallel BGL on a 64 node Opteron cluster. However, one cannot compare cluster based implementations with that of co-processor based implementations, like GPUs. For example, while a GPU based implementation is limited by its available memory, cluster based implementations can scale to very large instances. Also, the latency over the interconnect network will be large for a cluster based systems, which in the case of GPUs is very low.

There are a few related works on the GPU. The authors of [11] implement a variation of BFS on the GPU. Our implementations offer a reasonable speedup of about 4-5 compared to their implementation. Moreover, their implementation requires a sorted adjacency list based input representation. Our method is less sensitive to the topology of the graph.

To test the feasibility of computing connected components on the GPU instead of the CPU, we implemented an OpenMP based multithreaded implementation of our method. The GPU implementation gave a speedup of 8-9X. It should be noted that our method is focussed on improvements for a random edge list format, hence little cache based performance could be achieved for the CPU implementation. Compared to a DFS implementation, our CPU implementation achieved a speedup of around 1.2-1.5 X. Also, the time taken for format conversion from edge list to adjacency list, required for DFS on the CPU is not included in the timing for the CPU.

7. Conclusion

We have presented here an algorithm that is practical on the GPU, and gives good performance across input types. Thus further applications such as bi-connected components can be implemented using our method.

References

[1] AWERBUCH, B., AND SHILOACH, Y. New connectivity and msf algorithms for ultracomputer and pram. In *ICPP* (1983), pp. 175–179.

[2] BACKSTROM, L., HUTTENLOCHER, D., KLEINBERG, J., AND LAN, X. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (2006), ACM, p. 54.

[3] BADER, D., AND MADDURI, K. GTgraph: A suite of synthetic graph generators. url: http://wwwstatic. cc. gatech. edu/kamesh.

[4] BADER, D. A., AND CONG, G. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *Journal of Parallel and Distributed Computing 65*, 9 (2005), 994–1006.

[5] BUCK, I. GPU Computing with NVIDIA CUDA. In SIGGRAPH '07: ACM SIGGRAPH 2007 courses, 2007.

[6] CHAKRABARTI, D., ZHAN, Y., AND FALOUTSOS, C. R-MAT: A recursive model for graph mining. In *SIAM Data Mining* (2004), vol. 6.

[7] CORMEN, T., LEISERSON, C., RIVEST, R., AND STEIN, C. Introduction to algorithms, 1990.

[8] DOTSENKO, Y., GOVINDARAJU, N., SLOAN, P., BOYD, C., AND MANFERDELLI, J. Fast scan algorithms on graphics processors. In *In Proceedings of ICS* (2008), pp. 205–213.

[9] GREGOR, D., AND LUMSDAINE, A. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proceedings of the OOPSLA* (2005), vol. 40, ACM New York, NY, USA, pp. 423–437.

[10] GREINER, J. A comparison of parallel algorithms for connected components. In *Symposium on Parallel Algorithms and Architectures* (1994), Press, pp. 16–25.

[11] HARISH, P., AND NARAYANAN, P. J. Accelerating Large Graph Algorithms on the GPU using CUDA. In *Proc. of HiPC* (2007).

[12] HARRIS, M., OWENS, J., SENGUPTA, S., ZHANG, Y., AND DAVIDSON, A. Cudpp: Cuda data parallel primitives library. http://gpgpu.org/developer/cudpp., 2008.

[13] HIRSCHBERG, D. S. Parallel algorithms for the transitive closure and the connected components problem. In *Proc. of the ACM STOC* (1976), pp. 55–57.

[14] HIRSCHBERG, D. S., CHANDRA, A. K., AND SARWATE, D. V. Computing connected components in parallel computers. *Communications of the ACM 22*, 8 (1979), 461–464.

[15] HSU, T., RAMACHANDRAN, V., AND DEAN, N. Parallel implementation of algorithms for finding connected components in graphs. *Parallel algorithms: third DIMACS implementation challenge, October 17-19, 1994*, 20.

[16] JAJA, J. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

[17] LESKOVEC, J., KLEINBERG, J., AND FALOUTSOS, C. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of ACM SIGKDD* (New York, NY, USA, 2005), ACM, pp. 177–187.

[18] MÉNDEZ-LOJO, M., NGUYEN, D., PROUNTZOS, D., SUI, X., HASSAAN, M. A., KULKARNI, M., BURTSCHER, M., AND PINGALI, K. Structure-driven optimizations for amorphous data-parallel programs. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing* (2010), ACM, pp. 3–14.

[19] MOUNT, D., AND ARYA, S. ANN: A library for approximate nearest neighbor searching. In *CGC 2nd Annual Fall Workshop on Computational Geometry* (1997).

[20] NVIDIA, C. Cuda: Compute unified device architecture programming guide. Technical report, NVIDIA, 2007.

[21] REHMAN, M. S., KOTHAPALLI, K., AND NARAYANAN, P. J. Fast and Scalable List Ranking on the GPU. In *In Proceedings of ICS* (2009), pp. 152–160.

[22] SCARPAZZA, D. P., VILLA, O., AND PETRINI, F. Efficient breadth-first search on the cell/be processor. *IEEE Transactions on Parallel and Distributed Systems 19*, 10 (2008), 1381–1395.

[23] SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2007), pp. 97–106.

[24] SHILOACH, Y., AND VISHKIN, U. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms 3*, 1 (1982), 57–67.

[25] VINEET, V., HARISH, P. K., PATIDAR, S., AND NARAYANAN, P. J. Fast minimum spanning tree for large graphs on the gpu. In *In Proc. of High Performance Graphics* (2009).

[26] VINEET, V., AND NARAYANAN, P. J. CUDA Cuts: Fast Graph Cuts on the GPU. In *Proceedings of the CVPR Workshop on Visual Computer Vision on GPUs* (2008).

[27] YOO, A., CHOW, E., HENDERSON, K., MCLENDON, W., HENDRICKSON, B., AND CATALYUREK, U. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Proceedings of ICS* (2005), IEEE Computer Society, p. 25.