

# Accelerating large graph algorithms on the GPU using CUDA

Pawan Harish and P. J. Narayanan

Center for Visual Information Technology  
International Institute of Information Technology Hyderabad, INDIA  
{harishpk@research., pjn@}iiit.ac.in

**Abstract.** Graph algorithms are fundamental to many disciplines and application areas. Large graphs involving millions of vertices are common in scientific and engineering applications. Practical-time implementations using high-end computing resources have been reported but are accessible only to a few. Graphics Processing Units (GPUs) are fast emerging as inexpensive parallel processors due to their high computation power and low price. The G80 line of Nvidia GPUs provides the CUDA programming model that treats the GPU as a SIMD processor array. We present a few fundamental algorithms using this programming model on large graphs. We report results on breadth first search, single source shortest path, and all-pairs shortest path. Our implementation is able to compute single source shortest paths on a graph with 10 million vertices in about 1.5 seconds using the Nvidia 8800GTX GPU costing \$600. The GPU can then be used as an inexpensive co-processor to accelerate parts of the applications.

## 1 Introduction

Graph representations are common in many problem domains including scientific and engineering applications. Fundamental graph operations like breadth-first search, depth-first search, shortest path, etc., occur frequently in these domains. Some problems map to very large graphs, often involving millions of vertices. For example, problems like VLSI chip layout, phylogeny reconstruction, data mining, and network analysis can require graphs with millions of vertices. While fast implementations of sequential fundamental graph algorithms exist [4, 8] they are of the order of number of vertices and edges. Such algorithms become impractical on very large graphs. Parallel algorithms can achieve practical times on basic graph operations but at a high hardware cost. Bader et al. [2, 3] use CRAY MTA-2 supercomputer to perform BFS and single pair shortest path on very large graphs. While such methods are fast, the hardware used in them is very expensive.

Commodity graphics hardware has become a cost-effective parallel platform to solve many general problems. Many problems in the fields of linear algebra [6], image processing, computer vision, signal processing [11], etc., have benefited from its speed and parallel processing capability. GPU implementations of various graph algorithms also exist [9]. They are, however, severely limited by the memory capacity and architecture of the existing GPUs. GPU clusters have also been used [5] to perform compute

intensive tasks, like finite element computations [12], gas dispersion simulation, heat shimmering simulation [13], accurate nuclear explosion simulations, etc.

GPUs are optimized for graphics operations and their programming model is highly restrictive. All algorithms are disguised as graphics rendering with the programmable shaders interpreting the data. This was the situation until the latest model of GPUs following the Shader Model 4.0 were released late in 2006. These GPUs follow a unified architecture for all processors and can be used in more flexible ways than their predecessors. The G80 series of GPUs from Nvidia also offers an alternate programming model called Compute Unified Device Architecture (CUDA) to the underlying parallel processor. CUDA is highly suited for general purpose programming on the GPUs and provides a model close to the PRAM model. The interface uses standard C code with parallel features. A similar programming model called Close To Metal (CTM) is provided by ATI/AMD. Various products that transform the GPU technology to massive parallel processors for desktops are to be released in short time.

In this paper, we present the implementation of a few fundamental graph algorithms on the Nvidia GPUs using the CUDA model. Specifically, we show results on breadth-first search (BFS), single-source shortest path (SSSP), and all-pairs shortest path (APSP) algorithms on the GPU. Our method is capable of handling large graphs, unlike previous GPU implementations [9]. We can perform BFS on a 10 million vertex random graph with an average degree of 6 in one second and SSSP on it in 1.5 seconds. The times on a scale-free graph of same size is nearly double these. We also show that the repeated application of SSSP outscores the standard APSP algorithms on the memory restricted model of the GPUs. We are able to compute APSP on graphs with 30K vertices in about 2 minutes. Due to the restriction of memory on the CUDA device graphs above 12 million vertices with 6 degree per vertex cannot be handled using current GPUs.

The paper is organized as follows. An overview of GPU, general purpose programming (GPGPU) on it, and CUDA is given in Section 2. Section 3 presents the specific algorithms on the GPU using CUDA. Section 4 presents results of our implementation on various types of graphs. Conclusion and future work is discussed in Section 5.

## 2 GPU, GPGPU and CUDA

General purpose programming on Graphics processing Units (GPGPU) tries to solve a problem by posing it as a graphics rendering problem, restricting the range of solutions that can be ported to the GPU. A GPGPU solution is designed to follow the general flow of the graphics pipeline (consisting of vertex, geometry and pixel processors), with each iteration of the solution being one rendering pass. The GPU memory layout is also optimized for graphics rendering. This restricts the GPGPU solutions as an optimal data structure may not be available. Creating efficient data structures using the GPU memory model is a challenging problem in itself [7]. Memory size on GPU is another restricting factor. A single data structure on the GPU cannot be larger than the maximum texture size supported by it.

## 2.1 Compute Unified Device Architecture

On an abstract level, the Nvidia 8800 GTX graphics processor follows the shader model 4.0 design and implements the a 4-stage graphics pipeline. At the hardware level, however, it is not designed as 4 different processing units. All the 128 processors of the 8800 GTX are of same type with similar memory access speeds, which makes it a massive parallel processor. CUDA is a programming interface to use this parallel architecture for general purpose computing. This interface is a set of library functions which can be coded as an extension of the C language. A compiler generates executable code for the CUDA device. The CPU sees a CUDA device as a multi-core co-processor. The CUDA design does not have memory restrictions of GPGPU. One can access all memory available on the device using CUDA with no restriction on its representation though the access times vary for different types of memory. This enhancement in the memory model allows programmers to better exploit the parallel power of the 8800 GTX processor for general purpose computing.

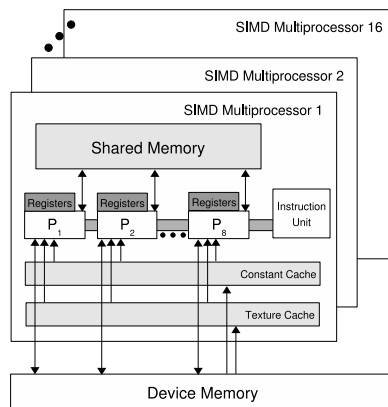


Fig. 1. CUDA Hardware interface

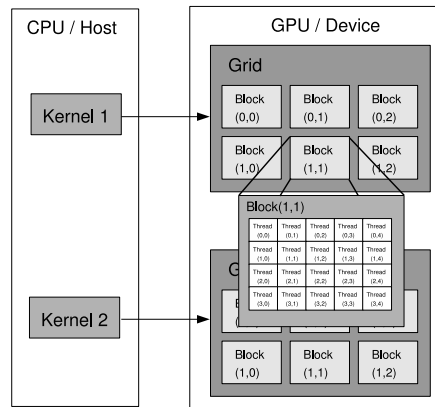


Fig. 2. CUDA programming model

**CUDA Hardware Model:** At the hardware level the 8800 GTX processor is a collection of 16 multiprocessors, with 8 processors each (Figure 1). Each multiprocessor has its own shared memory which is common to all the 8 processors inside it. It also has a set of 32-bit registers, texture, and constant memory caches. At any given cycle, each processor in the multiprocessor executes the same instruction on different data, which makes each a SIMD processor. Communication between multiprocessors is through the device memory, which is available to all the processors of the multiprocessors.

**CUDA Programming Model:** For the programmer the CUDA model is a collection of *threads* running in parallel. A *warp* is a collection of threads that can run simultaneously on a multiprocessor. The warp size is fixed for a specific GPU. The programmer decides the number of threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the multiprocessor. A collection of

threads (called a *block*) runs on a multiprocessor at a given time. Multiple blocks can be assigned to a single multiprocessor and their execution is time-shared. A single execution on a device generates a number of blocks. A collection of all blocks in a single execution is called a *grid* (Figure 2). All threads of all blocks executing on a single multiprocessor divide its resources equally amongst themselves. Each thread and block is given a unique ID that can be accessed within the thread during its execution. Each thread executes a single instruction set called the *kernel*.

The kernel is the core code to be executed on each thread. Using the thread and block IDs each thread can perform the kernel task on different set of data. Since the device memory is available to all the threads, it can access any memory location. The CUDA programming interface provides an almost Parallel Random Access Machine (PRAM) architecture, if one uses the device memory alone. However, the multiprocessors follow a SIMD model, the performance improves with the use of shared memory which can be accessed faster than the device memory. The hardware architecture allows multiple instruction sets to be executed on different multiprocessors. The current CUDA programming model, however, cannot assign different kernels to different multiprocessors, though this may be simulated using conditionals.

With CUDA, the GPU can be viewed as a massive parallel SIMD processor, limited only by the amount of memory available on the graphics hardware. The 8800 GTX graphics card has 768 MB memory. Large graphs can reside in this memory, given a suitable representation. The problem needs to be partitioned appropriately into multiple grids for handling even larger graphs.

### 3 Graph Algorithms and CUDA Implementation

As an extension of the C language, CUDA provides a high level interface to the programmer. Hence porting algorithms to the CUDA programming model is straight forward. Breadth first search (Section 3.2) and single source shortest path (Section 3.3) algorithms reported in this paper use one thread per vertex. All pairs shortest path implementations (Section 3.4) use  $V^2$  threads for the Floyd Warshall algorithm and  $V$  threads for other implementations. All threads in these implementations are multiplexed on 128 processors by the CUDA programming environment.

In our implementations of graph algorithms we do not use the device shared memory, as the data required by each vertex can be present anywhere in the global edge array (explained in the following section). Finding the locality of data to be collectively read into the shared memory is as hard as the BFS problem itself.

Denser graphs with more degree per vertex will benefit more using the following algorithms. Each iteration will expand the number of vertices being processed in parallel. The worst case will be when the graph is linear which will result in one vertex being processed every iteration.

#### 3.1 Graph Representation on CUDA

A graph  $G(V, E)$  is commonly represented as an adjacency matrix. For sparse graphs such a representation wastes a lot of space. Adjacency list is a more compact representa-

tion for graphs. Because of variable size of edge lists per vertex, its GPU representation may not be efficient under the GPGPU model. CUDA allows arrays of arbitrary sizes to be created and hence can represent graph using adjacency lists.

We represent graphs in compact adjacency list form, with adjacency lists packed into a single large array. Each vertex points to the starting position of its own adjacency list in this large array of edges. Vertices of graph  $G(V, E)$  are represented as an array  $V_a$ . Another array  $E_a$  of adjacency lists stores the edges with edges of vertex  $i + 1$  immediately following the edges of vertex  $i$  for all  $i$  in  $V$ . Each entry in the vertex array  $V_a$  corresponds to the starting index of its adjacency list in the edge array  $E_a$ . Each entry of the edge array  $E_a$  refers to a vertex in vertex array  $V_a$  (Figure 3).

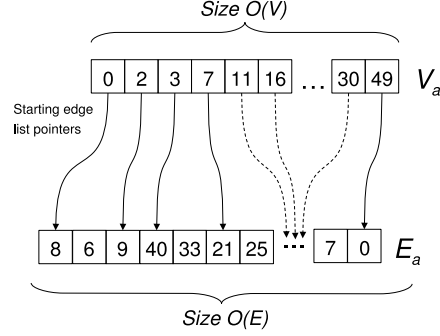


Fig. 3. Graph representation with vertex list pointing to a packed edge list.

### 3.2 Breadth First Search

BFS finds use in state space searching, graph partitioning, automatic theorem proving, etc., and is one of the most used graph operation in practical graph algorithms. The BFS problem is, given an undirected, unweighted graph  $G(V, E)$  and a source vertex  $S$ , find the minimum number of edges needed to reach every vertex  $V$  in  $G$  from source vertex  $S$ . The optimal sequential solution for this problem takes  $O(V + E)$  time.

**CUDA implementation of BFS:** We solve the BFS problem using level synchronization. BFS traverses the graph in levels; once a level is visited it is not visited again. The BFS *frontier* corresponds to all the nodes being processed at the current level. We do not maintain a queue for each vertex during our BFS execution because it will incur additional overheads of maintaining new array indices and changing the grid configuration at every level of kernel execution. This slows down the speed of execution on the CUDA model.

For our implementation we give one thread to every vertex. Two boolean arrays, *frontier* and *visited*,  $F_a$  and  $X_a$  respectively, of size  $|V|$  are created which store the BFS frontier and the visited vertices. Another integer array, *cost*,  $C_a$ , stores the minimal number of edges of each vertex from the source vertex  $S$ . In each iteration, each vertex

---

**Algorithm 1** CUDA\_BFS (Graph  $G(V, E)$ , Source Vertex  $S$ )

---

```
1: Create vertex array  $V_a$  from all vertices in  $G(V, E)$ ,
2: Create edge array  $E_a$  from all edges in  $G(V, E)$ 
3: Create frontier array  $F_a$ , visited array  $X_a$  and cost array  $C_a$  of size  $V$ .
4: Initialize  $F_a$ ,  $X_a$  to false and  $C_a$  to  $\infty$ 
5:  $F_a[S] \leftarrow \text{true}$ 
6:  $C_a[S] \leftarrow 0$ 
7: while  $F_a$  not Empty do
8:   for each vertex  $V$  in parallel do
9:     Invoke CUDA_BFS_KERNEL( $V_a, E_a, F_a, X_a, C_a$ ) on the grid.
10:  end for
11: end while
```

---

---

**Algorithm 2** CUDA\_BFS\_KERNEL ( $V_a, E_a, F_a, X_a, C_a$ )

---

```
1:  $tid \leftarrow \text{getThreadID}$ 
2: if  $F_a[tid]$  then
3:    $F_a[tid] \leftarrow \text{false}$ ,  $X_a[tid] \leftarrow \text{true}$ 
4:   for all neighbors  $nid$  of  $tid$  do
5:     if NOT  $X_a[nid]$  then
6:        $C_a[nid] \leftarrow C_a[tid] + 1$ 
7:        $F_a[nid] \leftarrow \text{true}$ 
8:     end if
9:   end for
10: end if
```

---

looks at its entry in the *frontier* array  $F_a$ . If true, it fetches its cost from the *cost* array  $C_a$  and updates all the costs of its neighbors if more than its own cost plus one using the edge list  $E_a$ . The vertex removes its own entry from the *frontier* array  $F_a$  and adds to the *visited* array  $X_a$ . It also adds its neighbors to the *frontier* array if the neighbor is not already visited. This process is repeated until the *frontier* is empty. This algorithm needs iterations of order of the diameter of the graph  $G(V, E)$  in the worst case.

Algorithm 1 runs on the CPU while algorithm 2 runs on the 8800 GTX GPU. The while loop in line 7 of Algorithm 1 terminates when all the levels of the graph are traversed and *frontier* array is empty. Results of this implementation are given in Figure 4.

### 3.3 Single Source Shortest Path

Single source shortest path (SSSP) problem is, given weighted graph  $G(V, E, W)$  with positive weights and a source vertex  $S$ , find the smallest combined weight of edges that is required to reach every vertex  $V$  from source vertex  $S$ . Dijkstra's algorithm is an optimal sequential solution to SSSP problem with time complexity  $O(V \log V + E)$ . Although parallel implementations of the Dijkstra's SSSP algorithm are available [10], an efficient PRAM algorithm does not exist.

---

**Algorithm 3** CUDA\_SSSP (Graph  $G(V, E, W)$ , Source Vertex  $S$ )

---

```
1: Create vertex array  $V_a$  from all vertices in  $G(V, E, W)$ ,
2: Create edge array  $E_a$  from all edges in  $G(V, E, W)$ 
3: Create weight array  $W_a$  from all weights in  $G(V, E, W)$ 
4: Create mask array  $M_a$ , cost array  $C_a$  and Updating cost array  $U_a$  of size  $V$ 
5: Initialize mask  $M_a$  to false, cost array  $C_a$  and Updating cost array  $U_a$  to  $\infty$ 
6:  $M_a[S] \leftarrow \text{true}$ 
7:  $C_a[S] \leftarrow 0$ 
8:  $U_a[S] \leftarrow 0$ 
9: while  $M_a$  not Empty do
10:   for each vertex  $V$  in parallel do
11:     Invoke CUDA_SSSP_KERNEL1( $V_a, E_a, W_a, M_a, C_a, U_a$ ) on the grid
12:     Invoke CUDA_SSSP_KERNEL2( $V_a, E_a, W_a, M_a, C_a, U_a$ ) on the grid
13:   end for
14: end while
```

---

---

**Algorithm 4** CUDA\_SSSP\_KERNEL1 ( $V_a, E_a, W_a, M_a, C_a, U_a$ )

---

```
1:  $tid \leftarrow \text{getThreadID}$ 
2: if  $M_a[tid]$  then
3:    $M_a[tid] \leftarrow \text{false}$ 
4:   for all neighbors  $nid$  of  $tid$  do
5:     if  $U_a[nid] > C_a[tid] + W_a[nid]$  then
6:        $U_a[nid] \leftarrow C_a[tid] + W_a[nid]$ 
7:     end if
8:   end for
9: end if
```

---

**CUDA implementation of SSSP:** The SSSP problem does not traverse the graph in levels. The cost of a visited vertex may change due to a low cost path being discovered later. The termination is based on the change in cost.

In our implementation, we use a *vertex* array  $V_a$  an *edge* array  $E_a$ , boolean *mask*  $M_a$  of size  $|V|$ , and a *weight* array  $W_a$  of size  $|E|$ . In each iteration each vertex checks if it is in the *mask*  $M_a$ . If yes, it fetches its current cost from the *cost* array  $C_a$  and its neighbor's weights from the *weight* array  $W_a$ . The cost of each neighbor is updated if greater than the cost of current vertex plus the edge weight to that neighbor. The new cost is not reflected in the *cost* array but is updated in an alternate array  $U_a$ . At the end of the execution of the kernel, a second kernel compares *cost*  $C_a$  with *updating cost*

---

**Algorithm 5** CUDA\_SSSP\_KERNEL2 ( $V_a, E_a, W_a, M_a, C_a, U_a$ )

---

```
1:  $tid \leftarrow \text{getThreadID}$ 
2: if  $C_a[tid] > U_a[tid]$  then
3:    $C_a[tid] \leftarrow U_a[tid]$ 
4:    $M_a[tid] \leftarrow \text{true}$ 
5: end if
6:  $U_a[tid] \leftarrow C_a[tid]$ 
```

---

$U_a$ . It updates the *cost*  $C_a$  only if it is more than  $U_a$  and makes its own entry in the *mask*  $M_a$ . The *updating cost* array reflects the *cost* array after each kernel execution for consistency.

The second stage of kernel execution is required as there is no synchronization between the CUDA multiprocessors. Updating the cost at the time of modification itself can result in read after write inconsistencies. The second stage kernel also toggles a flag if any *mask* is set. If this flag is not set the execution stops.

Newer version of CUDA hardware (ver 1.1) supports atomic read/write operations in the global memory which can help resolve inconsistencies. 8800 GTX is CUDA version 1.0 GPU and does not support such operations. Timings for SSSP CUDA implementations are given in Figure 4.

### 3.4 All Pairs Shortest Path

All pairs shortest path problem is, given a directed, weighted graph  $G(V, E, W)$  with positive weights, find the least weighted path from every vertex to every other vertex in the graph  $G(V, E, W)$ . Floyd Warshall's all pair shortest path algorithm requires  $O(V^3)$  time and  $O(V^2)$  space. Since APSP requires  $O(V^2)$  space it is impossible to go beyond a few thousand vertices for a graph on the GPU, due to the limited memory size. We show results on smaller graphs for this implementation. An implementation of Floyd Warshall's algorithm on SM 3.0 GPU can be found in [9]. Another approach for all pair shortest path is running SSSP from all vertices sequentially, this approach does not require  $O(V^2)$  space as can be seen by SSSP implementation in section 3.3. For this approach, we show results on larger graphs.

**CUDA implementation of APSP:** Since the output is of  $O(V^2)$ , we use an adjacency matrix for graphs rather than the one given in section 3.1. We use  $V^2$  threads, each running the classic CREW PRAM parallelization of Floyd Warshall algorithm (Algorithm 6). Another approach of implementing Floyd Warshall algorithm is to use  $O(V)$  threads, each further running a loop of  $O(V)$  inside it. We found this approach to be much slower because of the sequential access of entire vertex array by each thread. For example on a 1K graph it took around 9 seconds as compared to 1 second taken by Algorithm 6. The CUDA kernel code implements line 4 of Algorithm 6. The rest of the

---

#### Algorithm 6 Parallel-Floyd-Warshall( $G(V, E, W)$ )

---

```

1: Create adjacency Matrix  $A$  from  $G(V, E, W)$ 
2: for  $k$  from 1 to  $V$  do
3:   for all Elements in the Adjacency Matrix  $A$ , where  $1 \leq i, j \leq V$  in parallel do
4:      $A[i, j] \leftarrow \min(A[i, j], A[i, k] + A[k, j])$ 
5:   end for
6: end for
```

---

code is executed on the CPU. Results on various graphs for all pair shortest path are given in Figure 6.



---

**Algorithm 7** APSP\_USING\_SSSP( $G(V, E, W)$ )

---

```
1: Create vertex array  $V_a$ , edge array  $E_a$ , weight array  $W_a$  from  $G(V, E, W)$ ,
2: Create mask array  $M_a$ , cost array  $C_a$  and updating cost array  $U_a$  of size  $V$ 
3: for  $S$  from 1 to  $V$  do
4:    $M_a[S] \leftarrow \text{true}$ 
5:    $C_a[S] \leftarrow 0$ 
6:   while  $M_a$  not Empty do
7:     for each vertex  $V$  in parallel do
8:       Invoke CUDA_SSSP_KERNEL1( $V_a, E_a, W_a, M_a, C_a, U_a$ ) on the grid
9:       Invoke CUDA_SSSP_KERNEL2( $V_a, E_a, W_a, M_a, C_a, U_a$ ) on the grid
10:    end for
11:  end while
12: end for
```

---

Another alternative to find all pair shortest paths is to run SSSP algorithm from every vertex in graph  $G(V, E, W)$  (Algorithm 7). This will require only the final output size to be of  $O(V^2)$ , all intermediate calculations do not require this space. The final output could be stored in the CPU memory. Each iteration of SSSP will output a vector of size  $O(V)$ , which can be copied back to the CPU memory. This approach does not require the graph to be represented as an adjacency matrix, hence the representation given in section 3.1 can be used, which makes this algorithm suitable for large graphs. We implemented this approach and the results are given in Figure 6. This runs faster than the parallel Floyd Warshall algorithm because it is a single  $O(V)$  operation looping over  $O(V)$  threads. In contrast, the Floyd Warshall algorithm requires a single  $O(V)$  operation looping over  $O(V^2)$  threads which creates extra overhead for context switching of threads on the SIMD processors. Thus, due to the overhead for context switching of threads, the Floyd Warshall algorithm exhibits a slow down.

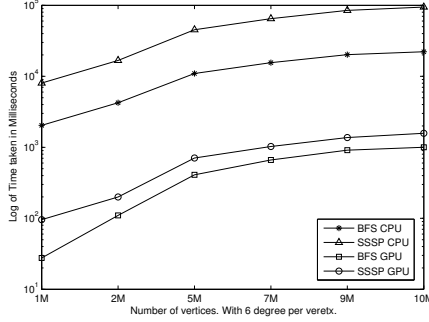
## 4 Experimental Results

All CUDA experiments were conducted on a PC with 2 GB RAM, Intel Core 2 Duo E6400 2.3GHz processor running Windows XP. A single Nvidia GeForce 8800 GTX GPU was used to run CUDA applications. The graphics card has 768 MB RAM on board. For the CPU implementation a PC with 3 GB RAM and an AMD Athlon 64 3200+ running 64 bit version of Fedora Core 4 was used. Applications were written in CUDA version 0.8.1 and C++ using Visual Studio 2005. Nvidia Graphics driver version 97.73 was used for CUDA compatibility. CPU applications were written in C++ using standard template library.

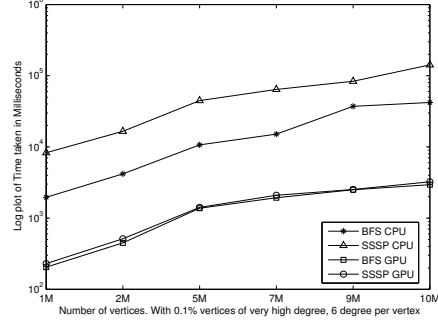
The results for CUDA BFS implementation and SSSP implementations are summarized in Figure 4 for random general graphs. As seen from the results, even for graphs with millions of vertices and edges the GPU is capable of performing BFS at high speeds. Implementation of Bader et al. of BFS for a 400 million vertex, 2 billion edges graph takes less than 5 seconds on a CRAY MTA-2, the 40 processor supercomputer [2], which costs 5–6 orders more than a CUDA hardware. We also implemented

BFS on CPU, using C++ and found BFS on GPU to be 20–50 times faster than its CPU counterpart.

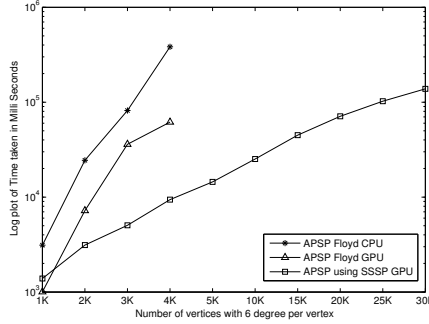
SSSP timings are comparable to that of BFS for random graphs given in Figure 4, due to the randomness associated in these graphs. Since the degree per vertex is 6–7 and the weights vary from 1–10 in magnitude it is highly unlikely to have a less weighted edge coming back from a far away level. We compare our results with the SSSP CPU implementation, our algorithm is 70 times faster than its CPU counterpart on an average.



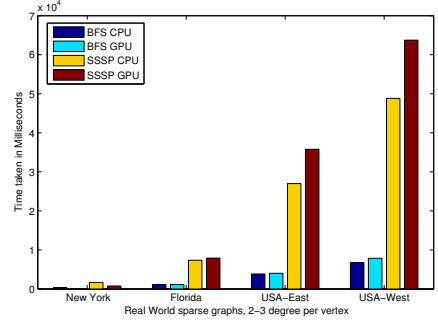
**Fig. 4.** BFS and SSSP times with weights ranging from 1-10



**Fig. 5.** BFS and SSSP times for Scale Free graphs, weights ranging from 1-10



**Fig. 6.** APSP timings for various graphs, weights ranging from 1-10



**Fig. 7.** Real world graphs timings, weights ranging from 1-300K, avg degree per vertex 2-3

Many real world networks fall under the category of scale free graphs. In such graphs a few vertices are of high degree while the rest are of low degree. For these graphs we kept the maximum degree of any vertex to be 1000 and average degree per vertex to be 6. 0.1% of the total number of vertices were given very high degree. The results are summarized in Figure 5. As seen from the results, BFS and SSSP are slower for scale free graphs as compared to random graphs. Because of the large degree at some vertices, the loop inside the kernel (line 4 of Algorithm 2 and line 4 of Algorithm 4) increases, which results in more lookups to the device memory slowing down the kernel execution time. Loops of non-uniform lengths are inefficient on a SIMD architecture.

Figure 6 summarizes results of all pair shortest path implementation on the CUDA architecture. The SSSP implementation of all pair shortest path requires only one vector of  $O(V)$  to be copied to the CPU memory in each iteration, it does not require adjacency matrix representation of the graph and hence only  $O(V)$  processors are required for its operation. We also implemented CPU version of the Floyd Warshall algorithm and found an average improvement of a factor of 3 for the Floyd Warshall CUDA algorithm and a factor of 17 for the all pair shortest path using SSSP CUDA implementation. As shown by the results APSP using SSSP is faster than Floyd Warshall's APSP algorithm on the GPU, it was found to be orders of magnitude slower when implemented on the CPU.

Figure 7 summarizes the results for BFS and SSSP for real world data. The graphs were downloaded from the DIMACS challenge [1] site. The New York graph has 250K vertices and 730K edges. Florida graph has 1M vertices with 2.7M edges. USA-East and USA-West have 3M vertices, 8M edges and 6M vertices, 15M edges respectively. All graphs have average branching factor of 2–3. The edge weights vary from 1–300K.

The results show that for both BFS and SSSP the GPU has not gained much in terms of speed. This is due to the low average degree of these graphs. A degree of 2–3 makes these graphs almost linear. In the case of linear graphs parallel algorithms cannot gain much as it becomes necessary to process every vertex in each iteration and hence the performance decreases.

## 5 Conclusions and Future Work

In this paper, we presented fast implementations of a few fundamental graph algorithms for large graphs on the GPU hardware. These algorithms have wide practical applications. We presented fast solutions of BFS, SSSP, and APSP on very large graphs at very high speeds using a GPU instead of expensive supercomputers. The Nvidia 8800GTX costs \$600 today and will be much cheaper before this article comes to print. The CUDA model can exploit the GPU hardware as a massively parallel co-processor.

The size of the device memory limits the size of the graphs handled on a single GPU. The CUDA programming model provides an interface to use multiple GPUs in parallel using multi-GPU bridges. Up to 2 synchronized GPUs can be combined using the SLI interface. Nvidia QuadroPlex is a CUDA enabled graphics solution with two Quadro 5600 cards each. Two such systems can be supported by a single CPU to give even better performance than the 8800GTX. Nvidia has announced its Tesla range of GPUs, with up to four 8800 cores and higher memory capacity, targeted at high performance computing. Further research is required on partitioning the problem and streaming the data from the CPU to GPU to handle even larger problems. External memory approaches can be adapted to the GPUs for this purpose.

Another drawback of the GPUs is the lack of double or higher precision, a serious limitation for scientific applications. The regular graphics rendering applications and games – which drive the GPU market – do not require high precisions. Graphics hardware vendors have announced limited double precision support to make their hardware more appealing to high performance computing community. The use of GPUs as economical, high-performance co-processors can be a significant driving force in the

future. It has the potential to bring double precision support to the GPU hardware in the future.

## References

1. Ninth DIMACS implementation challenge - Shortest paths  
<http://www.dis.uniroma1.it/challenge9/download.shtml>.
2. David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *ICPP*, pages 523–530, 2006.
3. David A. Bader and Kamesh Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 539–550, Washington, DC, USA, 2006. IEEE Computer Society.
4. Jun-Dong Cho, Salil Raje, and Majid Sarrafzadeh. Fast approximation algorithms on maxcut, k-coloring, and k-color ordering for vlsi applications. *IEEE Transactions on Computers*, 47(11):1253–1266, 1998.
5. Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.
6. Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003.
7. Aaron Lefohn, Joe M. Kniss, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, January 2006.
8. Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.
9. Paulius Micikevicius. General parallel computation on commodity graphics hardware: Case study with the all-pairs shortest paths problem. In *PDPTA*, pages 1359–1365, 2004.
10. Anna S. Nepomniaschaya and Maria A. Dvoskina. A simple implementation of dijkstra’s shortest path algorithm on associative parallel processors. *Fundam. Inf.*, 43(1-4):227–243, 2000.
11. John D. Owens, Shubhabrata Sengupta, and Daniel Horn. Assessment of Graphic Processing Units (GPUs) for Department of Defense (DoD) Digital Signal Processing (DSP) Applications. Technical Report ECE-CE-2005-3, Department of Electrical and Computer Engineering, University of California, Davis, October 2005.
12. Wen Wu and Pheng Ann Heng. A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting: Research Articles. *Comput. Animat. Virtual Worlds*, 15(3-4):219–227, 2004.
13. Ye Zhao, Yiping Han, Zhe Fan, Feng Qiu, Yu-Chuan Kuo, Arie E. Kaufman, and Klaus Mueller. Visual simulation of heat shimmering and mirage. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):179–189, 2007.