Large Graph Algorithms for Massively Multithreaded Architectures

Pawan Harish, Vibhav Vineet and P. J. Narayanan
Center for Visual Information Technology
International Institute of Information Technology Hyderabad, INDIA
harishpk@research.iiit.ac.in, vibhavvinet@research.iiit.ac.in, pjn@iiit.ac.in
Technical Report Number IIIT/TR/2009/74

Modern Graphics Processing Units (GPUs) provide high computation power at low costs and have been described as desktop supercomputers. The GPUs expose a general, data-parallel programming model today in the form of CUDA and CAL. The GPU is presented as a massively multithreaded architecture by them. Several high-performance, general data processing algorithms such as sorting, matrix multiplication, etc., have been developed for the GPUs. In this paper, we present a set of general graph algorithms on the GPU using the CUDA programming model. We present implementations of breadth-first search, st-connectivity, single-source shortest path, all-pairs shortest path, minimum spanning tree, and maximum flow algorithms on commodity GPUs. Our implementations exhibit high performance, especially on large graphs. We experiment on random, scale-free, and real-life graphs of up to millions of vertices. Parallel algorithms for such problems have been reported in the literature before, especially on supercomputers. The approach has been that of divide-and-conquer, where individual processing nodes solve smaller sub-problems followed by a combining step. The massively multithreaded model of the GPU makes it possible to adopt the data-parallel approach even to irregular algorithms like graph algorithms, using $O(V)$ or $O(E)$ simultaneous threads. The algorithms and the underlying techniques presented in this paper are likely to be applicable to many irregular algorithms on them.

1. Introduction

Graphs are popular data representations in many computing, engineering, and scientific areas. Fundamental graph operations such as breadth first search, st-connectivity, shortest paths, etc., are building blocks to many applications. Implementations of serial fundamental graph algorithms exist [36,14] with computing time of the order of vertices and edges. Such implementations become impractical on very large graphs involving millions of vertices and edges, common in many domains like VLSI layout, phylogeny reconstruction, network analysis, etc. Parallel processing is essential to apply graph algorithms on large datasets. Parallel implementations of some graph algorithms on supercomputers are reported, but are accessible only to a few owing to the high hardware costs [6,8,52]. CPU clusters have been used for distributed implementations. Synchronization however becomes a bottleneck for them. All graph algorithms cannot scale to parallel hardware models. For example, there does not exist an efficient PRAM solution to the DFS problem. A suitable mix of parallel and serial hardware is required for efficient implementation in such cases.

Modern Graphics Processing Units (GPUs) provide high computation power at low costs and have been described as desktop supercomputers. The GPUs have been used for many general purpose computations due to their low cost, high computing power, and high availability. The latest GPUs, for instance, can deliver close to 1 TFLOPs of compute power at a cost of around \$400. The stages of the graphics pipeline were exploited for parallelism with the flow of execution handled serially using the pipeline in the earlier, GPGPU model. The GPUs expose a general, data-parallel programming model today in the form of CUDA and CAL. The recently adopted OpenCL standard [38] will provide a common computing model to not only all GPUs, but also to other platforms like multicore, manycore, and Cell/B.E. The Compute Unified Device Architecture (CUDA) from Nvidia presents a heterogeneous programming model where the parallel hardware can be used in conjunction with the CPU. This provides good control over sequential flow of execution which was absent from the earlier GPGPU. CUDA can be used to imitate a parallel random access machine (PRAM) if global memory alone is used. In conjunction with a CPU, it can be used as a bulk synchronous parallel (BSP) hardware with the CPU deciding the barrier for synchronization.

CUDA presents the GPU as a massively threaded parallel architecture, allowing upto millions of threads to run in parallel over its processors, with each having access to a common global memory. Such a tight architecture is a departure from the supercomputers, which typically have a small number of powerful cores. The parallelizing approach on them was that of divide-and-conquer, where individual processing nodes solve smaller sub-problems followed by a combining step. The massively multithreaded model presented by the GPU makes it possible to adopt the data-parallel approach even to irregular algorithms like graph algorithms, using $O(V)$ or $O(E)$ simultaneous threads. The multicore and manycore processors of the future are likely to support a massively multithreaded model. Each core will be simple with support for multiple threads in flight, as the number of cores exceeds a few dozens into the hundreds as has been proposed.

Several high-performance, general data processing algorithms such as sorting, matrix multiplication, etc., have been developed for the GPUs. In this paper, we present a set of general graph algorithms on the GPU, using the CUDA programming model. Specifically, we present implementations of breadth first search (BFS), st-connectivity (STCON), single source shortest path (SSSP), all pairs shortest path (APSP), minimum spanning trees (MST), and maximum flow (MF). Our implementations exhibit high performance, especially on large graphs. We experiment on random, scale-free, and real-life graphs of up to millions of vertices. Our algorithms don't just present high performance. They also provide strategies for exploiting the massively multithreaded architectures on future manycore architectures for graph theory as well as for other problems.

Using a single graphics card, we perform BFS in about half a second on a $10M$ vertex graph with $120M$ edges, and SSSP on it in 1.5 seconds. On the DIMACS full-USA graph of $24M$ vertices and $58M$ edges it takes less than 9 seconds for our implementation to compute the minimum spanning tree. We study different approaches to APSP and show a speed up by a factor of $2-4$ times over Katz and Kider [33]. A speed up of nearly $10-15$ times over CPU Boost graph library is achieved for all algorithms are reported in this paper for general large graphs.

2. Compute Unified Device Architecture

Programmability was introduced to the GPU with shader model 1.0 and since enhanced up to the current shader model 4.0 standard. A GPGPU solution poses an algorithm as a series of rendering passes following the graphics pipeline. Programmable shaders are used to interpret data in each pass and write the output to the frame buffer.

Amount of memory available on the GPU and its representation are restricting factors for GPGPU algorithms; a GPU cannot handle data greater than the largest texture supported by it. The GPU memory layout is also optimized for graphics rendering which restricts the GPGPU solution. Limited access to memory and processor anatomy makes it tricky to port general algorithms to this framework [40].

CUDA enhances the GPGPU programming model by not treating the GPU as a graphics pipeline but as a multicore co-processor. Further it improves the GPGPU model by removing memory restrictions for each processor. Data representation is also improved by providing programmer friendly data structures. All memory available on the CUDA device can be accessed by all processors with no restriction on its representation, though the access times may vary for different types of memory.

At the hardware level, CUDA is a collection of multiprocessors consisting of a series processors. Each multiprocessor contains a small shared memory, a set of 32-bit registers, texture, and

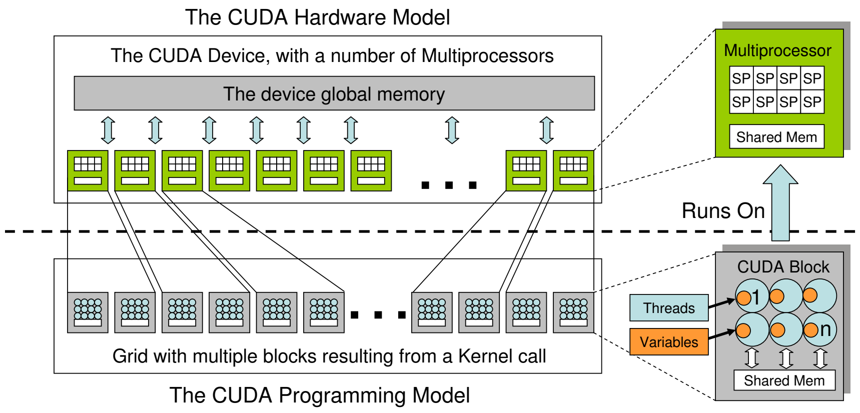


Figure 1. The CUDA hardware model (top) and programming model (bottom), showing the block to multiprocessor mapping.

constant memory caches common to all processors inside it. Each processor in the multiprocessor executes the same instruction on different data, which makes it a SIMD model. Communication between multiprocessors is through the device global memory, which is accessible to all processors within a multiprocessor.

As a software interface, CUDA API 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. For the programmer CUDA is a collection of *threads* running in parallel. Each thread can use a number of private registers for its computation. A collection of threads (called a *block*) runs on a multiprocessor at a given time. The threads of each block have access to a small amount of common shared memory. Synchronization barriers are also available for all threads of a block. A group of blocks can be assigned to a single multiprocessor but their execution is time-shared. The available shared memory and registers are split equally amongst all blocks that timeshare a multiprocessor. A single execution on a device generates a number of blocks. Multiple groups of blocks are also time shared on the multiprocessor for execution. A collection of all blocks in a single execution is called a *grid* (Figure 1).

Each thread executes a single instruction set called the *kernel*. Each thread and block is given a unique ID that can be accessed within the thread during its execution. These can be used by each thread to perform the kernel task on its part of the data, an SIMD execution. An algorithm may use multiple kernels, which share data through the global memory and synchronize their execution at the end of each kernel. Threads from multiple blocks can only synchronize at the end of the kernel execution by all threads.

3. Representation and Algorithm Outline

Efficient data structures for graph representation have been studied in depth. Complex data structures like hash tables [32] have been used for efficiency on the CPU. The GPU memory layout is optimized for graphics rendering and cannot support user defined data structures efficiently. Creating an efficient data structure under the GPGPU memory model is a challenging problem [35,28]. However, the CUDA model treats memory as general arrays and can support more efficient data structures.

The adjacency matrix is a good choice for representing graphs on the GPU. However, it is not suitable for large graphs because of its $O(V^2)$ space requirements. This restricts the size of graphs that can be handled by the GPU. Adjacency list is a more practical representation for large graphs requiring $O(V + E)$ space. We represent graphs using a compact adjacency list representation with each vertex pointing to its starting edge list in a packed adjacency list of edges

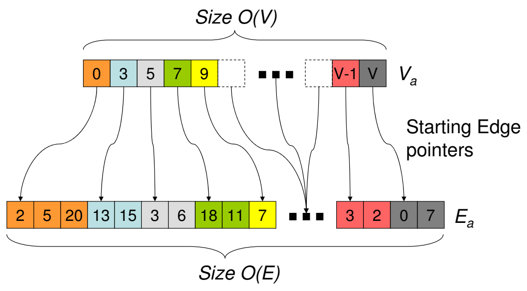


Figure 2. Graph representation is in terms of a vertex list that points to a packed edge list.

(Figure 2). Uneven array sizes can be supported on CUDA.

The vertex list V_a points to its starting index in the edge list E_a. Each entry in the edge list E_a points to a vertex in vertex list V_a. Since we deal with undirected graphs, each edge results in one entry for each of its end vertices. Cache efficiency is hard to achieve using this representation as the edge list can point to any vertex in V_a and can cause random jumps in memory. The problem of laying out data in memory for efficient cache usage is a variant of the BFS problem itself.

We use this representation for all algorithms reported in this paper except one all pairs shortest paths method. A block-divided adjacency matrix representation is used to exploit better cache efficiency there (explained in section 7.2.1). The output for APSP requires $O(V^2)$ space and thus adjacency matrix is a more suitable representation.

3.1. Algorithm Outline on CUDA

The CUDA hardware can be seen as a multicore/manycore co-processor in a bulk synchronous parallel mode when used in conjunction with the CPU. Synchronization of CUDA threads can be achieved with the CPU deciding the barrier for synchronization. Broadly a bulk synchronous parallel machine follows three steps: **(a)** *Concurrent computation*: Asynchronous computation takes place on each processing element (PE). **(b)** *Communication*: PEs exchange data between each other. **(c)** *Barrier Synchronization*: Each process waits for other processes to finish. Concurrent computation takes place at the CUDA device in the form of program kernels with communication through the global memory. Synchronization is achieved only at the end of each kernel. Algorithm 1 outlines the CPU code in this scenario. The skeleton code runs on the CPU while the kernels run on a CUDA device.

Algorithm 1 CPU_SKELETON

```
1:  Create and initialize working arrays on CUDA device.
2:  while NOT $Terminate$ do
3:      $Terminate \leftarrow$ true
4:      For each vertex/edge/color in parallel:
5:      Invoke Kernel1
6:      Synchronize
7:      For each vertex/edge/color in parallel:
8:      Invoke Kernel2
9:      Synchronize
10:     etc...
11:     For each vertex/edge/color in parallel:
12:     Invoke Kernel$n$ and modify $Terminate$
13:     Synchronize
14:     Copy $Terminate$ from GPU to CPU
15: end while
```

The termination of an operation depends on a consensus between threads. A logical OR operation needs to be performed over all active threads for termination. We use a single boolean variable (initially set to true) that is written over by all threads independently, typically by the last kernel during execution. Each non-terminating thread writes a false to this location in global memory. If no thread modifies this value, the loop terminates. The variable needs to be copied from GPU to CPU after each iteration to check for termination (Algorithm 1 line 2).

Algorithms presented differ from each other in the kernel code and the data structure requirements but the CPU skeleton pseudo-code given above applies to all algorithms reported in this paper.

3.2. Vertex List Compaction

We assign threads to an attribute of the graph (vertex, color etc.) in most implementations. This leads to $|V|$ threads executing in parallel. The number of active vertices, however, varies in each iteration of execution. Active vertices are typically indicated in an *activity mask*, which holds a 1 for each active vertex. During execution, each vertex thread confirms its status from the activity mask and continues execution if active. This can lead to poor load balancing on the GPU, as CUDA blocks have to be scheduled even when all vertices of the block are inactive, leading to an unbalanced SIMD execution. Performance will improve if we deploy only as many threads as the active vertices, reducing the number of blocks and thus time sharing on the CUDA device [46].

A scan operation on the activity mask can determine the number of active vertices as well as give each an ordinal number. This establishes a mapping between the original vertex index and its new index amongst the currently active vertices. Using the scan output, we compact all entries in the activity mask to a new active mask (Figure 3) creating the mapping of *new* thread IDs to *old* vertex IDs. While using active mask, each thread finds its vertex by looking at its active mask and thereafter executes normally.

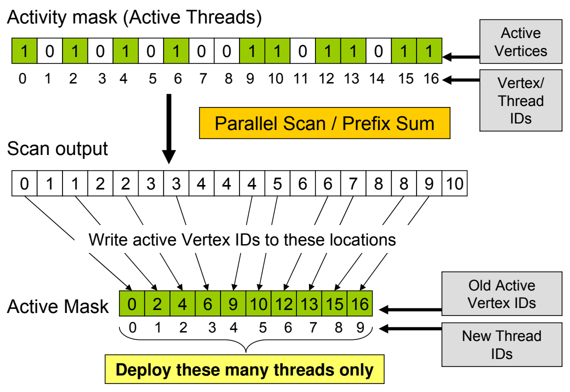


Figure 3. Vertex compaction is used to reduce the number of threads needed when not all vertices are active.

There is a trade-off between time taken by parallel thread execution and time taken for scan and compacting. For graphs where parallelism expands slowly, compaction makes most sense, as many threads will be inactive in a single grid execution. For faster expanding graphs, compacting can become an overhead. We report experiments where vertex compaction gives better performance than the non compacted version.

4. Breadth First Search (BFS)

The BFS problem is to find the minimum number of edges needed to reach every vertex in graph G from a source vertex s. BFS is well studied in serial setting with best time complexity reported as $O(V + E)$. Parallel versions of BFS algorithm also exist. A study of the BFS algorithm on Cell/B.E. processor using the bulk synchronous parallel model appeared in [45]. Zhang et al. [53] gave a heuristic search for BFS using level synchronization. Bader et al.[6] implement BFS for the CRAY MTA$-$2 supercomputer and Yoo et al. [52] on the BlueGene/L.

We treat the GPU as a bulk synchronous device and use level synchronization to implement BFS. BFS traverses the graph in levels, once a level is visited it is not visited again during execution. We use this as our barrier and synchronize threads at each level. A BFS *frontier* corresponds to all vertices at the current level, see Figure 4. Concurrent computation takes place at the BFS frontier where each vertex updates the cost of its neighboring vertices by assigning cost values to their respective indices in the global memory.

We assign one thread to every vertex, eliminating the need for queues in our implementation. This decision further eliminates the need to change grid configuration and reassigning indices in the global memory with every kernel execution, which incurs additional overheads and slows down the execution.

GPU Implementation

We keep two boolean arrays F_a and X_a of size $|V|$ for the frontier and visited vertices respectively. Initially, X_a is set to false and F_a contains the source vertex. In the first kernel (Algorithm 2), each thread looks at its entry in the frontier array F_a (Figure 4). If present, it updates the cost of its unvisited neighbors by writing its own cost plus one to its neighbor's index in the global cost array C_a.*

*It is possible for many vertices to write a value at one location concurrently while executing this step, leading to

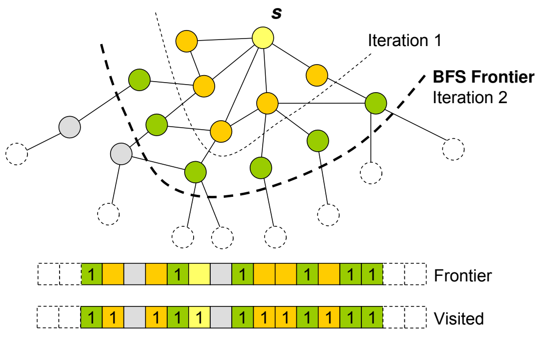


Figure 4. Parallel BFS: Vertices in the `frontier` list execute in parallel in each iteration. Execution stops when the `frontier` is empty.

Algorithm 2 KERNEL1_BFS

```
 1: $tid \leftarrow$ getThreadID
 2: if $F_a[tid]$ then
 3:    $F_a[tid] \leftarrow$ false
 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:          $F_{ua}[nid] \leftarrow$ true
 8:       end if
 9:    end for
10: end if
```

Each thread removes its vertex from the frontier array F_a and adds its neighbors to an alternate updating frontier array F_{ua}. This is needed as there is no synchronization possible between all CUDA threads. Modifying the frontier at the time of updation may result in read after write inconsistencies. A second kernel (Algorithm 3) copies the updated frontier F_{ua} to the actual frontier F_a. It adds the vertex in F_{ua} to the visited vertex array X_a and sets the termination flag as false.

The process is repeated until the frontier array is empty and the while loop in Algorithm 1 line 2 terminates. In the worst case, the algorithm needs the order of the diameter of the graph $G(V, E)$ iterations. Results for this implementation are summarized in Figure 11.

clashes in the global memory. We do not lock memory for concurrent write operations because all frontier vertices write the same value at their neighbor's index location in C_a. CUDA guarantees at least one of them will succeed which is sufficient for our BFS cost propagation.

Algorithm 3 KERNEL2_BFS

```
1: $tid \leftarrow$ getThreadID
2: if $F_{ua}[tid]$ then
3:    $F_a[tid] \leftarrow$ true
4:    $X_a[tid] \leftarrow$ true
5:    $F_{ua}[tid] \leftarrow$ false
6:    $Terminate \leftarrow$ false
7: end if
```

5. ST-Connectivity (STCON)

The st-Connectivity problem resembles the BFS problem closely. Given an unweighted directed graph $G(V, E)$ and two vertices, s and t, find a path if one exists from s to t. The problem falls under NL-complete category with a non-deterministic time complexity requiring log-space. The undirected version of this algorithm falls in SL category with same space complexity. Trifonov [48] gives a theoretical deterministic algorithm requiring $O(\log(n)\log\log(n))$ space for the undirected case. Reingold [44] provides the optimal deterministic $O(\log(n))$ space algorithm for the undirected case. These results however are not implemented in practice. Bader et al. [6] implement STCON by extending their BFS implementation; they find the smallest distance between s and t by keeping track of all expanded frontier vertices. We also modify BFS to find the smallest number of edges needed to reach t from s for the undirected case.

Our approach starts BFS concurrently from s and t with initial colors assigned to them. In each iteration, colors are propagated to neighbors along with the BFS cost. Termination is when both colors meet. Evidently, both frontiers hold the smallest distance to current vertices from their respective source vertices, the smallest path between s and t is reached when frontiers come in contact with each other. Figure 5 depicts two termination condition due to merging of frontiers, either at a vertex or an edge.

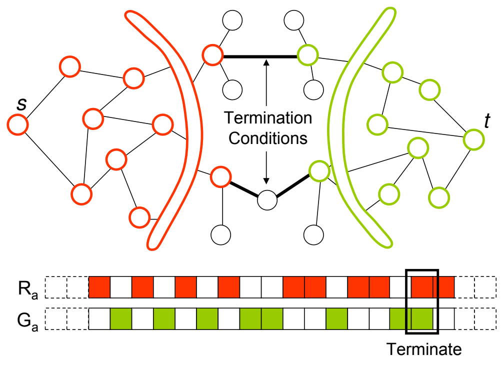


Figure 5. Parallel st-connectivity with colors expanding from s and t vertices.

GPU Implementation

Along with V_a, E_a, F_a, and C_a we keep two boolean arrays R_a and G_a, one for red color and the other for green, of size $|V|$ as the vertices visited by s and t frontiers respectively. Initially R_a and G_a are set to false and F_a contains the source and target vertices. To keep the state of variables intact, alternate updating arrays R_{ua}, G_{ua} and F_{ua} of size $|V|$ are used in each iteration.

Each vertex, if present in F_a, reads its color in both R_a and G_a and sets its own color to one of the two. This is exclusive as a vertex can only exist in one of the two arrays as an overlap is a termination condition for the algorithm. Each vertex updates the cost of its unvisited neighbors by adding 1 to its own cost and writing it to the neighbor's index in C_a. Based on its color, the vertex also adds its neighbors to its own color's visited vertices by adding them to either R_{ua} or G_{ua}. The algorithm terminates if any unvisited neighbor of the vertex is of the opposite color. The vertex removes itself from the frontier array F_a and adds its neighbors to the updating frontier array F_{ua}. Kernel1 (Algorithm 4) depicts these steps.

The second Kernel (Algorithm 5) copies the updating arrays F_{ua}, R_{ua}, G_{ua} to actual arrays F_a, R_a and G_a for all newly visited vertices. It also checks the termination condition due to merging of frontiers and terminates the algorithm if frontiers meet at any vertex. Figure 12 summarizes results for this implementation.

Algorithm 4 KERNEL1_STCON

```
1: $tid \leftarrow$ getThreadID
2: if $F_a[tid]$ then
3:     $F_a[tid] \leftarrow$ false
4:     for all neighbors $nid$ of $tid$ do
5:         if $(R_a[tid] \& G_a[nid]) \mid (G_a[tid] \& R_a[nid])$ then Terminate
6:         if NOT $(G_a[nid] \mid R_a[nid])$ then
7:             if $G_a[tid]$ then $G_{ua}[nid] \leftarrow$ true
8:             if $R_a[tid]$ then $R_{ua}[nid] \leftarrow$ true
9:             $F_{ua}[nid] \leftarrow$ true
10:            $C_a[nid] \leftarrow C_a[tid]+1$
11:        end if
12:    end for
13: end if
```

Algorithm 5 KERNEL2_STCON

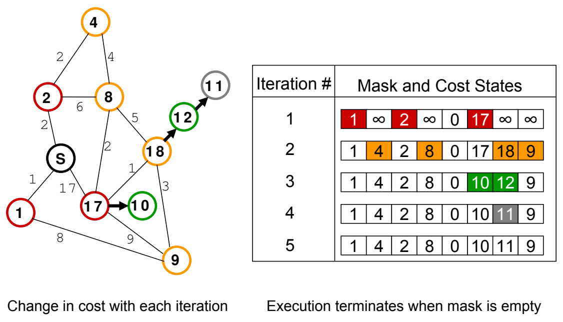
```
1: $tid \leftarrow$ getThreadID
2: if $F_{ua}[tid]$ then
3:     if $G_{ua}[tid]$ $\& R_{ua}[tid]$ then Terminate
4:     $F_a[tid] \leftarrow$ true
5:     if $R_{ua}[tid]$ then $R_a[tid] \leftarrow$ true
6:     if $G_{ua}[tid]$ then $G_a[tid] \leftarrow$ true
7:     $F_{ua}[tid] \leftarrow$ false
8:     $R_{ua}[tid] \leftarrow$ false
9:     $G_{ua}[tid] \leftarrow$ false
10: end if
```

6. Single Source Shortest Path (SSSP)

The sequential solution to single source shortest path problem comes from Dijkstra [22]. Originally the algorithm required $O(V^2)$ time but was later improved using Fibonacci heap to $O(V \log V + E)$. A parallel version of Dijkstra's algorithm on a PRAM given in [18] introduces a $O(V^{1/3} \log V)$ algorithm requiring O$(V \log V)$ work. Nepomniaschaya et al. [41] parallelized Dijkstra's algorithm for associative parallel processors. Narayanan [39] solves the SSSP problem for processor arrays. Although parallel implementations of the Dijkstra's SSSP algorithm are reported [17], an efficient PRAM algorithm does not exist.

Single source shortest path does not traverse a graph in levels, as cost of a visited vertex may change due to a low cost path being discovered later in the execution. Simultaneous updates are triggered by vertices undergoing a change in cost values. These vertices constitute an *execution mask*. Termination condition is reached with equilibrium when there is no change in cost for any vertex.

We assign one thread to every vertex. Threads in the execution mask execute in parallel. Each vertex updates the cost of its neighbors and removes itself from the execution mask. Any vertex whose cost is updated is put into the execution mask for next iteration of execution. This process is repeated until there is no change in cost for any vertex. Figure 6 shows the execution mask (shown as colors) and cost states for a simple case, costs are updated in each iteration, with vertices undergoing re-execution if their cost changes.


Change in cost with each iteration
Execution terminates when mask is empty

Iteration #	Mask and Cost States							
1	1	∞	2	∞	0	17	∞	∞
2	1	4	2	8	0	17	18	9
3	1	4	2	8	0	10	12	9
4	1	4	2	8	0	10	11	9
5	1	4	2	8	0	10	11	9

Figure 6. SSSP execution: In each iteration, vertices in the mask update costs of their neighbors. A vertex whose cost changes is put in the mask for execution in the next iteration.

GPU Implementation

For our implementation (Algorithm 6 and Algorithm 7) we keep a boolean mask M_a and cost array C_a of size $|V|$. W_a holds the weights of edges and an updating cost array C_{ua} is used for intermediate cost values. Initially the mask M_a contains the source vertex. Each vertex looks at its entry in the mask M_a. If true, it fetches its own cost from C_a and updates the cost of its neighbors if greater than $C_a(u)+W_a(u,v)$ in the updating cost array C_{ua} (where u is the current vertex index and v is the neighboring vertex index). The alternate cost array C_{ua} is used to resolve read after write inconsistencies in the global memory. Update in $C_{ua}(v)$ needs to lock its memory location for modifying the cost as many threads may write different values at the same location concurrently. We use the *atomicMin* function supported on CUDA 1.1 hardware (lines $5-9$, Algorithm 6) to resolve this.

Algorithm 6 KERNEL1_SSSP
```
 1: $tid \leftarrow$ getThreadID
 2: **if** $M_a[tid]$ **then**
 3:     $M_a\ [tid] \leftarrow$ false
 4:     **for all** neighbors $nid$ of $tid$ **do**
 5:         **Begin Atomic**
 6:         **if** $C_{ua}[nid] > C_a[tid] + W_a[nid]$ **then**
 7:             $C_{ua}[nid] \leftarrow C_a[tid]+W_a[nid]$
 8:         **end if**
 9:         **End Atomic**
10:     **end for**
11: **end if**
```

Algorithm 7 KERNEL2_SSSP
```
1: $tid \leftarrow$ getThreadID
2: **if** $C_a[tid] > C_{ua}[tid]$ **then**
3:     $C_a[tid] \leftarrow C_{ua}[tid]$
4:     $M_a[tid] \leftarrow$ true
5:     $Terminate \leftarrow$ false
6: **end if**
7: $C_{ua}[tid] \leftarrow C_a[tid]$
```

Atomic functions resolve concurrent writes by assigning exclusive rights to one thread at a time. The clashes are thus serialized in an unspecified order. The function compares the existing $C_{ua}(v)$ cost with $C_a(u)+W_a(u,v)$ and updates the value if necessary. A second kernel (Algorithm 7) is used to reflect updating cost C_{ua} to the cost array C_a. If C_a is greater than C_{ua} for any vertex, it is set for execution in the mask M_a and the termination flag is toggled to continue execution. This process is repeated until the mask is empty. Experimental results for this implementation are reported in figure 13.

7. All Pairs Shortest Paths (APSP)

Warshall defined boolean transitive closure for matrices that was later used to develop the Floyd Warshall algorithm for the APSP problem. The algorithm had $O(V^2)$ space complexity and $O(V^3)$ time complexity. Numerous parallel versions for the APSP problem have been developed to date [47,43,29]. Micikevicius [37] reported a GPGPU implementation for the same, but due to $O(V^2)$ space requirements he reported results on small graphs.

The Floyd Warshall parallel CREW PRAM algorithm (Algorithm 8) can be easily extended to CUDA if the graph is represented as an adjacency matrix. The kernel implements line 4 of Algorithm 8 while the rest of the code runs on the CPU. This approach however requires entire matrix to be present on the CUDA device. In practice this approach performs slower as compared to approaches outlined below. Please see [30] for a comparative study.

Algorithm 8 Parallel-Floyd-Warshall

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

7.1. APSP using SSSP

Reducing space requirements on the CUDA device directly helps in handling larger graphs. A simple space conserving solution to the APSP problem is to run SSSP from each vertex iteratively using the graph representation given in Figure 2. This implementation requires $O(V+E)$ space on the GPU with a vector of $O(V)$ copied back to the CPU memory in each iteration. However for dense graphs this approach proves inefficient. We implemented this approach for general graphs and found it to be a scalable solution for low degree graphs. See the results in Figure 14.

7.2. APSP as Matrix Multiplication

Katz and Kider [33] formulate a CUDA implementation for APSP on large graphs using a matrix block approach. They implement the Floyd Warshall algorithm based on transitive closure with a cache efficient blocking technique (extension of method proposed by Venkataraman [49]), in which the adjacency matrix (broken into blocks) present in the global memory is brought into the multiprocessor shared memory intelligently. They handle larger graphs using multiple CUDA devices by partitioning the problem across the number of devices. We take a different approach and use streaming of data from the CPU to GPU memory for handling larger matrices. Our implementation uses a modified parallel matrix multiplication with blocking approach. Our times are slightly slower as compared to Katz and Kider for fully connected small graphs. For general large graphs however we gain $2-4$ times speed over the method proposed by Katz and Kider.

A simple modification to the matrix multiplication algorithm yields an APSP solution (Algorithm 9). Lines $4-11$ is the general matrix multiplication algorithm with the multiplication and addition operations replaced by addition and minimum operations respectively, line 7. The outer loop (line 3) utilizes the transitive property of matrix multiplication and runs $\log V$ times.

Algorithm 9 MATRIX_APSP

```
1: $D^1 \leftarrow A$
2: for $m \leq \log V$ do
3:     for $i \leftarrow 1$ to $V$ do
4:         for $j \leftarrow 1$ to $V$ do
5:             $D^m_{i,j} \leftarrow \infty$
6:             for $k \leftarrow 1$ to $V$ do
7:                 $D^m_{i,j} \leftarrow min(D^m_{i,j},\ D^{(m-1)}_{i,j} + A_{k,j})$
8:             end for
9:         end for
10:    end for
11: end for
```

We modify the parallel version of matrix multiplication proposed by Volkov and Demmel [51] for our APSP solution.

7.2.1. Cache Efficient Graph Representation

For matrix multiplication based APSP, we use an adjacency matrix to represent graph. Figure 7 depicts a cache efficient conflict free blocking scheme used for matrix multiplication by Volkov and Demmel. We present two new ideas over the basic matrix multiplication scheme. The first is the modification to handle graphs larger than the device memory by streaming data as required from the CPU. The second is the lazy evaluation of the minimum finding which results in a huge boost in performance.

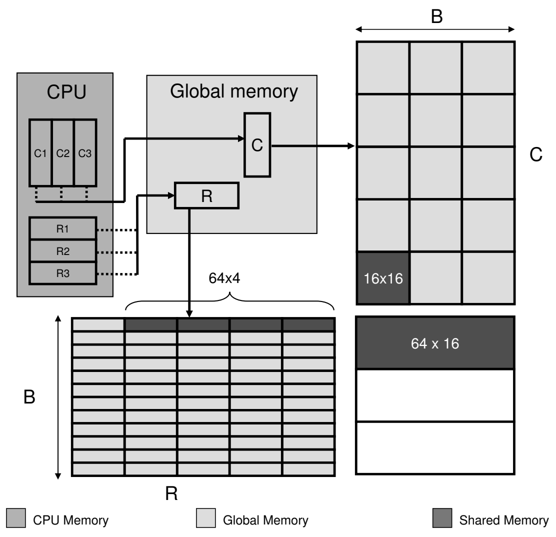


Figure 7. Blocks for matrix multiplication by Volkov and Demmel [51] modified to stream from CPU to GPU.

Streaming Blocks: To handle large graphs, the adjacency matrix present in the CPU memory is divided into rectangular row and column submatrices. These are streamed into the device global memory and a matrix-block D^m based on their values is computed. Let R be the row and C the column submatrices of the original matrix present in the device memory. For every row submatrix R we iterate through all column submatrices C of the original matrix. We assume CPU memory is large enough to hold the adjacency matrix, though our method can be easily extended to secondary storage with slight modification.

Let the size of available device memory be GPU_{mem}. We divide the adjacency matrix into rows R and column C submatrices of size $(B \times V)$ and $(V \times B)$ respectively such that

$$\text{size}\left(R_{B\times V} + C_{V\times B} + D^m_{B\times B}\right) \leq \text{GPU}_{mem},$$

where B is the block size. A total of

$$\log V\left(\frac{V^3}{B} + V^2\right) \equiv O\left(\log V\left(\frac{V^3}{B}\right)\right)$$

elements are transferred between CPU and GPU for a $V \times V$ adjacency matrix for our APSP computation, with $V^3 \log V/B$ reads and $V^2 \log V$ writes. Time taken for this data transfer is negligible compared to the computation time, and can be easily hidden using asynchronous read and write operations supported on current generation CUDA hardware.

For example, for a $18K \times 18K$ matrix with integer entries and 1GB device memory, a block size $B \simeq 6K$ can be used. At a PCI-e×16 practical transfer rate of 3.3 GB/s, data transfer takes nearly 16 seconds. This time is negligible as compared to $\simeq 800$ seconds of computation time taken on Tesla for a $18K \times 18K$ matrix without streaming (result taken from Table 4).

Lazy Minimum Evaluation: The basic step of Floyd's algorithm is similar to matrix multiplication with multiplication replaced by addition and addition by minimum finding. However, for sparse-degree graphs, the connections are few and the other entries of the adjacency matrix are infinity. When an entry is infinity, additions involving it and subsequent minimum finding can be skipped altogether without affecting correctness. We, therefore, evaluate the minimum in a lazy manner, skipping all paths involving a non-existent edge. This results in a speedup of 2 to 3 times over complete evaluation on most graphs. This also makes the running time degree-dependent.

GPU Implementation

Let R be the row and C be the column submatrices of the adjacency matrix. Let D^i denote a temporary matrix variable of size $B \times B$ used to hold intermediate values. In each iteration of outer loop (Algorithm 9, line 2) D^i is modified using C and R. Lines $3-10$ of Algorithm 9 are executed on the CUDA device while the rest of the code executes on the CPU.

Algorithm 10 KERNEL_APSP

```
1: $d[1:16] \leftarrow$ 16 values of $64 \times 16$ block of $D^i$
2: for loop over (dimension of $C$)/16 times do
3:     $c[1:16][1:16] \leftarrow$ next $16 \times 16$ block of $C$
4:     $p \leftarrow 0$
5:     for 1 to 4 do
6:         $r[1:4] \leftarrow$ 4 values of next $64 \times 16$ block of $R$
7:         for $j \leftarrow 1$ to 4 do
8:             for $k \leftarrow 1$ to 16 do
9:                 if $d[k] > r[j]+c[p][k]$ then
10:                    $d[k] \leftarrow r[j]+c[p][k]$
11:                end if
12:            end for
13:            $p \leftarrow p+1$
14:        end for
15:    end for
16: end for
17: Merge $d[1:16]$ with $64 \times 16$ block of $D^i$
```

Our kernel modifies the matrix multiplication routine given by Volkov and Demmel [51] by replacing the multiplication and addition operations with addition and minimum operations. Shared memory is used as a user managed cache to improve performance. Volkov and Demmel bring sections of matrices R, C and D^i into shared memory in blocks: R is brought in 64×4 sized blocks, C in 16×16 sized blocks and D^i in 64×16 sized blocks. These values are selected to maximize throughput of the CUDA device. During execution, each thread computes 64×16 values of D^i. Algorithm 10 describes the modified matrix multiplication kernel. Please see [51] for full details on the matrix multiplication kernel.

Here d, r and c are shared memory variables. The above kernel produces a speed up of up to 2 times over the earlier parallel matrix multiplication kernel [42]. Lazy minimum evaluation goes further; if either entry of R or C is infinity due to a non-existent edge, we do not update the output matrix. We achieve much better performance on general graphs using lazy min evaluation. However, computation time is no more independent of the degree of the graph.

7.3. Gaussian Elimination Based APSP

In a parallel work that came to light very recently, Buluc et al. [11] formulate a fast recursive APSP algorithm based on Gaussian elimination. They cleverly extend the R-Kleene [20] algorithm for in place APSP computation on global memory. They split each APSP step recursively into 2 APSPs involving graphs of half the size, 6 matrix multiplications and 2 matrix additions. The base-case is when there are 16 or fewer vertices; Floyd's algorithm is applied in that case by modifying the CUDA matrix multiplication kernel proposed by Volkov and Demmel [51]. They also use the fast matrix multiplication for other steps. Their implementation is degree independent and fast; they achieve a speed up of $5-10$ times over the APSP implementation presented above.

While the approach of Buluc et al. is the fastest APSP implementation on the GPU so far, our key ideas can extend it further. We incorporated the lazy minimum evaluation into their code and obtained a speed up of 2 over their native approach. Their approach is memory heavy and is best suited when the adjacency matrix can fit completely in the GPU device memory. Their approach involves several matrix multiplications and additions. Extending this to stream the data from CPU to the GPU for matrix operations in terms of blocks that fit in the device memory will involve many more communications between the CPU and the GPU and many more computations. The CPU to GPU communication bandwidth has not at all kept pace with the increase in the number of cores or computation power of the GPU. Thus, our non-matrix approach is likely to scale better to arbitrarily high graphs than the Gaussian Elimination based approach by Buluc et al.

Comparison of the matrix multiplication approach with APSP using SSSP and Gaussian elimination approach is summarized in Figure 14. Comparison of matrix multiplication approach with Katz and Kider is given in Figure 15. Behavior of the matrix approach with varying degree is reported in Table 1.

8. Minimum Spanning Tree (MST)

Best time complexity for a serial solution to the MST problem, proposed by Bernard Chazelle [13], is $O(E\alpha(E,V))$, where α is the functional inverse of Ackermann's function. Borůvka's algorithm [10] is a popular solution to the MST problem. In a serial setting it takes $O(E \log V)$ time. However, numerous parallel variations of this algorithm also exist [34]. Chong et al. [15] report a EREW PRAM algorithm requiring $O(\log V)$ time and $O(V \log V)$ work. Bader et al. [4] design a fast algorithm for symmetric multiprocessors with $O((V+E)/p)$ lookups and local operations for a p processor machine. Chung et al. [16] efficiently implement Borůvka's algorithm on a asynchronous distributed memory machine by reducing communication costs. Dehne and Götz implement three variations of Borůvka's algorithm using the BSP model [21].

We implement a modified parallel Borůvka algorithm on CUDA. We create colored partial spanning trees from all vertices, grow individual trees, and merge colors when trees come in contact. Cycles are removed explicitly in each iteration. Connected components are found via color propagation, an approach similar to our SSSP implementation (section 6).

We represent each supervertex in Borůvka's algorithm as a color. Each supervertex finds the minimum weighted edge to another supervertex and adds it to the output MST array. Each newly added edge in the MST edge list updates the colors of both its supervertices until there is no change in color values for all supervertices. Cycles are removed from the newly created graph and each vertex in a supervertex updates its color to the new color of the supervertex. This processes is repeated and the number of supervertices keep on decreasing. The algorithm terminates when exactly one supervertex remains.

GPU Implementation

We use colors array C_a, color index array Ci_a (per vertex color index to which the vertex belongs to), active colors array Ac_a and newly added MST edges $NMst_a$ of size $|V|$. Output is a

Algorithm 11 Minimum Spanning Tree

```
1: Create $V_a$, $E_a$, $W_a$ from $G(V,E,W)$
2: Initialize $C_a$ and $Ci_a$ to vertex id.
3: Initialize $Mst_a$ to false
4: while More than 1 supervertex remains do
5:    Clear $NMst_a$, $Ac_a$, $Deg_a$ and $Cy_a$
6:    Kernel1 for each vertex: Finds the minimum weighted outgoing edge from each supervertex to the lowest outgoing color by working at each vertex of the supervertex, sets the edge in $NMst_a$.
7:    Kernel2 for each supervertex: Each supervertex sets its added edge in $NMst_a$ as part of output MST, $Mst_a$.
8:    Kernel3 for each supervertex: Each added edge, in $NMst_a$, increments the degrees of both its supervertices in $Deg_a$ using color as index. Old colors are saved in $PrevC_a$.
9:    while no change in color values $C_a$ do
10:      Kernel4 for each supervertex: Each edge in $NMst_a$ updates colors of supervertices by propagating the lower color to the higher.
11:   end while
12:   while 1 degree supervertex remains do
13:      Kernel5 for each supervertex: All 1 degree supervertices nullify their edge in $NMst_a$, and decrement their own degree and the degree of its outgoing supervertex using old colors from $PrevC_a$.
14:   end while
15:   Kernel6 for each supervertex: Each remaining edge in $NMst_a$ adds itself to $Cy_a$ using new colors from $C_a$.
16:   Kernel7 for each supervertex: Each entry in $Cy_a$ is removed from the output MST, $Mst_a$, resulting in cycle free MST.
17:   Kernel8 for each vertex: Each vertex updates its own colorindex to the new color of its new supervertex.
18: end while
19: Copy $Mst_a$ to CPU memory as output.
```

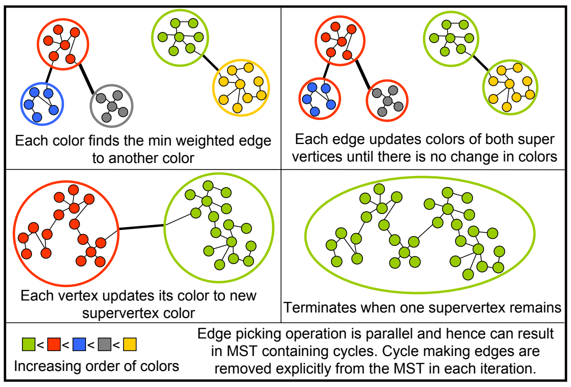

Each color finds the min weighted edge to another color
Each edge updates colors of both super vertices until there is no change in colors
Each vertex updates its color to new supervertex color
Terminates when one supervertex remains
🟩< 🟥< 🟦< ⬛< 🟨
Increasing order of colors
Edge picking operation is parallel and hence can result in MST containing cycles. Cycle making edges are removed explicitly from the MST in each iteration.

Figure 8. Parallel minimum spanning tree.

set of edges present in the MST, let Mst_a of size $|E|$ denote this. Further, we keep degrees array Deg_a and cycle edges array Cy_a of size $|V|$ for cycle finding and elimination. Arrays V_a, E_a and W_a retain their previous meanings. Initially, C_a holds the vertex id as color and each vertex points to this color in Ci_a, Ac_a and Mst_a are set to false and $NMst_a$ holds null. We assign one color to each vertex in the graph initially, eliminating uncolored vertices and thus race conditions due to them. An overview of the algorithm using steps presented in following sections is given in Algorithm 11.

8.1. Finding Minimum Weighted Edge

Every vertex knows its color, i.e., the supervertex it belongs to. Each vertex finds its minimum weighted edge using edge weights W_a. The index of this edge is written atomically to the color index of the supervertex in global memory. If multiple edges in a supervertex have minimum weight, the one with minimum outgoing color is selected. Algorithm 12 finds the minimum weighted edge for each supervertex. Please note lines $10 - 14$ in the pseudo code (Algorithm 12) are implemented as multiple atomic operations in practice.

Algorithm 13 adds the minimum weighed edge from each supervertex to the final MST output array Mst_a. This kernel is important as we cannot add an edge to MST array until all vertices belonging to a single supervertex have voted for their lowest weighted edge. This Kernel executes for all supervertices (or active colors) after Kernel1 executes for every vertex of the graph.

Algorithm 12 KERNEL1_MST

```
1:  $tid \leftarrow$ getThreadID
2:  $cid \leftarrow Ci_a[tid]$
3:  $col \leftarrow C_a[cid]$
4:  for all edges $eid$ of $tid$ do
5:      $col2 \leftarrow C_a[Ci_a[E_a[eid]]]$
6:      if NOT $Mst_a[eid]$ & $col \neq col2$ then
7:          $Ieid \leftarrow$ Index($min(W_a[eid]$ &$col2$))
8:      end if
9:  end for
10: Begin Atomic
11: if $W_a[Ieid] > W_a[NMst_a[col]]$ then
12:     $NMst_a[col] \leftarrow Ieid$
13: end if
14: End Atomic
15: $Ac_a[col] \leftarrow$ true
```

Algorithm 13 KERNEL2_MST

```
1: $col \leftarrow$ getThreadID
2: if $Ac_a[col]$ then
3:     $Mst_a[NMst_a[col]] \leftarrow$ true
4: end if
```

8.2. Finding and Removing Cycles

As C edges are added for C colors, atleast one cycle is expected to be formed in the new graph of supervertices. Multiple cycles can also form for disjoint components of supervertices. Figure 9 shows such a case. It is easy to see that each such component can have atmost one cycle consisting of exactly 2 supervertices with both edges in the cycle having equal weights. Identifying these edges and removing one edge per cycle is crucial for correct output.

In order to find these edges, we assign degrees to supervertices using newly added edges $NMst_a$. We then remove all 1-degree supervertices iteratively until there is no 1-degree supervertex left, resulting in supervertices that are part of cycles.

Each added edge increments the degree of both its supervertices using color of the supervertex as its index in Deg_a (Algorithm 14). After color propagation, i.e., merger of supervertices (Section 8.3), all 1-degree supervertices nullify their added edge in $NMst_a$. They also decrement their own degree and the degree of thier added edge's outgoing supervertex in Deg_a (Algorithm 16). This process is repeated until there is no 1-degree supervertex left, resulting in supervertices whose edges form a cycle.

Incrementing the degree array needs to be done before propagating colors, as the old color is used as index in Deg_a for each supervertex. Old colors are also needed after color propagation to identify supervertices while decrementing the degrees. To this end, we preserve old colors before propagating *new* colors in an alternate color array $PrevC_a$ (Algorithm 14).

After removing 1-degree supervertices, resulting supervertices write their edge from $NMst_a$ to their new color location in Cy_a (Algorithm 17), after new colors have been assigned to supervertices of each disjoint component using Algorithm 15. One edge of the two, per disjoint component, survives this step. Since both edges have equal weights, there is no preference for any edge. Edges in Cy_a are then removed from the output MST array Mst_a (Algorithm 18) resulting in cycle free partial MST.

Algorithm 14 KERNEL3_MST
```
1: $col \leftarrow$ getThreadID
2: if $Ac_a[col]$ then
3:     $col2 \leftarrow C_a[Ci_a[E_a[NMst_a[col]]]]$
4:     Begin Atomic
5:     $Deg_a[col] \leftarrow Deg_a[col]+1$
6:     $Deg_a[col2] \leftarrow Deg_a[col2]+1$
7:     End Atomic
8: end if
9: $PrevC_a[col] \leftarrow C_a[col]$
```

8.3. Merging Supervertices

Each added edge merges two supervertices. Lesser color of the two is propagated by assigning it to the higher colored supervertex. This process is repeated until there is no change in color values for any supervertex. Color propagation mechanism is similar to the SSSP step. Kernel4 (Algorithm 15) executes for each added edge and updates the colors of both the vertices to the lower one. As in the SSSP implementation, we use an alternate color array to store intermediate values and to resolve read after write inconsistencies.

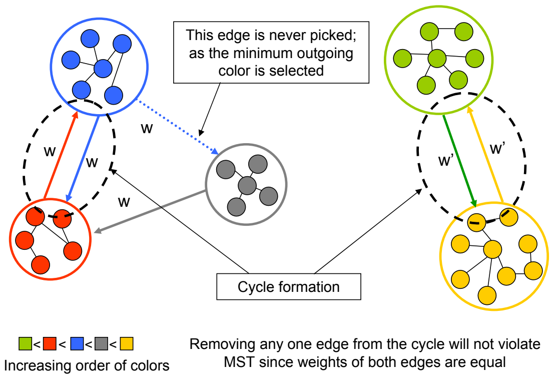

This edge is never picked; as the minimum outgoing color is selected
w
w
w
w
w'
w'
Cycle formation
🟩<🟥<🟦<⬛<🟨
Increasing order of colors
Removing any one edge from the cycle will not violate MST since weights of both edges are equal

Figure 9. For C colors, C edges are added, resulting in multiple cycles. One edge per cycle must be removed.

Algorithm 15 KERNEL4_MST
```
1: $cid \leftarrow$ getThreadID
2: $col \leftarrow C_a[cid]$
3: if $Ac_a[col]$ then
4:     $cid2 \leftarrow Ci_a[E_a[NMst_a[col]]]$
5:     Begin Atomic
6:     if $C_a[cid] > C_a[cid2]$ then
7:         $C_a[cid] \leftarrow C_a[cid2]$
8:     end if
9:     if $C_a[cid2] > C_a[cid]$ then
10:        $C_a[cid2] \leftarrow C_a[cid]$
11:    end if
12:    End Atomic
13: end if
```

8.4. Assigning Colors to Vertices

Each vertex in a supervertex must know its color; merging of colors in the previous step does not necessarily end with all vertices in a component being assigned the minimum color of that component. Rather, a link in color values is

Algorithm 16 KERNEL5_MST

```
1: $cid \leftarrow$ getThreadID
2: $col \leftarrow PrevC_a[cid]$
3: if $Ac_a[col]$ & $Deg_a[col] = 1$ then
4:     $col2 \leftarrow PrevC_a[Ci_a[E_a[NMst_a[cid]]]]$
5:     Begin Atomic
6:     $Deg_a[col] \leftarrow Deg_a[col] - 1$
7:     $Deg_a[col2] \leftarrow Deg_a[col2] - 1$
8:     End Atomic
9:     $NMst_a[col] \leftarrow \phi$
10: end if
```

Algorithm 17 KERNEL6_MST

```
1: $cid \leftarrow$ getThreadID
2: $col \leftarrow PrevC_a[cid]$
3: if $Ac_a[col]$ & $NMst_a[col] \neq \phi$ then
4:     $newcol \leftarrow C_a[Ci_a[E_a[NMst_a[col]]]]$
5:     $Cy_a[newcol] \leftarrow NMst_a[col]$
6: end if
```

Algorithm 18 KERNEL7_MST

```
1: $col \leftarrow$ getThreadID
2: if $Cy_a[col] \neq \phi$ then
3:     $Mst_a[Cy_a[col]] \leftarrow$ false
4: end if
```

Algorithm 19 KERNEL8_MST

```
1: $tid \leftarrow$ getThreadID
2: $cid \leftarrow Ci_a[tid]$
3: $col \leftarrow C_a[cid]$
4: while $col \neq cid$ do
5:     $col \leftarrow C_a[cid]$
6:     $cid \leftarrow C_a[col]$
7: end while
8: $Ci_a[tid] \leftarrow cid$
9: if $col \neq 0$ then
10:     $Terminate \leftarrow$ false
11: end if
```

established during the previous step. This link must be traversed by each vertex to find the lowest color it should point to. Since the color is same as the index initially, the color and the index must be same for all active colors. This property is used while updating colors for each vertex. Each vertex in Kernel8 (Algorithm 19) finds its colorindex *cid* and traverses the colors array C_a until coloridnex is not equal to color.

Timings for MST implementation on various types of graphs are reported in Figure 16.

9. Maximum Flow (MF)/Min Cut

Maxflow tries to find the minimum weighed cut that separates a graph into two disjoint sets of vertices, one containing the source s and the other target t. The fastest serial solution due to Goldberg and Rao takes $O(Emin(V^{2/3}, \sqrt{E}) \log(V^2/E) \log(U))$ time [26], where U is the maximum capacity of the graph.

Popular serial solutions to the max flow problem include Ford-Fulkerson's algorithm [25], later improved by Edmond and Karp [23], and the Push-Relabel algorithm [27] by Goldberg and Tarjan. Edmond-Karp's algorithm repeatedly computes augmenting paths from s to t using BFS, through which flows are pushed, until no augmented paths exist. The Push-Relabel algorithm works by pushing flow from s to t by increasing heights of nodes farther away from t. Rather than examining the entire residual network to find an augmenting path, it works locally, looking at each vertex's neighbors in the residual graph.

Anderson and Setubal [3] first gave a parallel version of the Push-Relabel algorithm. Bader and Sachdeva implemented parallel cache efficient variation of the push-relabel algorithm using an SMP [9]. Alizadeh and Goldberg [2] implemented the same on a massively parallel Connection Machine CM$-$2. GPU implementations of the push-relabel algorithm are also reported [31]. A CUDA implementation for grid graphs specific to vision applications is reported in [50]. We implement the parallel push-relabel algorithm using CUDA for general graphs.

The Push-Relabel Algorithm

The push-relabel algorithm constructs and maintains a residual graph at all times. The residual graph G_f of the graph G has the same topology, but consists of the edges which can admit more flow. Each edge has a current capacity in G_f, called its residual capacity which is the amount of flow that it can admit currently. Each vertex in the graph maintains a reservoir of flow (excess flow) and a height. Based on its height and excess flow either push or relabel operations are undertaken at each vertex. Initially height of s is set to $|V|$ and height of t to 0. Height at all times is a conservative estimate of the vertex's distance from the source.

- **Push**: The push operation is applied at a vertex if its height is one more than any of its neighbor and it has excess flow in its reservoir. The result of push is either saturation of an edge in G_f or saturation of vertex, i.e., empty reservoir.
- **Relabel**: Relabel operation is applied to change the heights. If any vertex has excess flow but there is a height mismatch and it cannot flow, the relabel operation makes its height one more than the minimum height of its neighbor.

Better estimates of height values can be obtained using global or gap relabeling [9]. Global relabeling uses BFS to correctly assign distances from the target whereas gap relabeling finds gaps using height mismatches in the entire graph. However, both are expensive operations, especially when executed on parallel hardware. The algorithm terminates when neither push nor relabeling can be applied. The excess flows in the nodes are then pushed back to the source and the saturated nodes of the final residual graph gives the maximum flow/minimal cut.

GPU Implementation

We keep e_a and h_a arrays representing excess flow and height per vertex. An activity mask M_a holds three distinct sates per vertex, 0 corresponding to the relabeling state ($e_a(u)> 0$, $h_a(v) \geq h_a(u)$ $\forall$ neighbors $v \in G_f$), 1 for the push state ($e_a(u) > 0$ and $h_a(u) = h_a(v)+1$ for any neighbor $v \in G_f$) and 2 for saturation. Based on these values the push and relabel operations are undertaken. Initially activity mask is set to 0 for all vertices. We use both local relabeling and global relabeling. We apply multiple pushes before applying the relabel operation. Multiple local relabels are also applied before applying a single global relabel step. Algorithm 20 describes this scenario. Backward BFS from the sink is used for global relabeling.

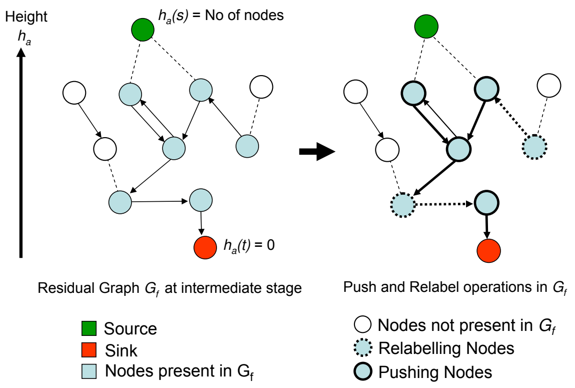


Figure 10. Parallel maxflow, showing push and relabel operations based on e_a and h_a

Algorithm 20 Max Flow Step

```
1: for 1 to $k$ times do
2:     Apply $m$ pushes
3:     Apply Local Relabel
4: end for
5: Apply Global Relabel
```

Relabel: Relabels are applied as given in Algorithm 20. Local relabel operation is applied at a vertex if it has positive excess flow but no push is possible to any neighbor due to height mismatch. The height of vertex is increased by setting it to one more than the minimum height of its neighboring nodes.

For local relabeling, each vertex updates its height to one more than the minimum height of its neighbors in the residual graph. Kernel1 (Algorithm 21) explains this operation. For global relabeling we use backward BFS from sink, which propagates the height values to each vertex in the residual graph based on its actual distance from sink.

Algorithm 21 KERNEL1_MAXFLOW

```
1: $tid \leftarrow$ getThreadID
2: if $M_a[tid] = 0$ then
3:    for all neighbors $nid$ of $tid$ do
4:       if $nid \in G_f \& minh > h_a[nid]$ then
5:          $minh \leftarrow h_a[nid]$
6:       end if
7:    end for
8:    $h_a[tid] \leftarrow minh + 1$
9:    $M_a[tid] \leftarrow 1$
10: end if
```

Push: The push operation can be applied at a vertex if it has excess flow and its height is equal to 1 more than one of its neighbor's. After the push, either vertex is saturated (i.e., excess flow is zero) or the edge is saturated (i.e., capacity of edge is zero).

Each vertex looks at its activity mask M_a, if 1 it pushes the excess flow along the edges present in residual graph. It atomically subtracts the flow from its own reservoir and adds it to the neighbor's reservoir. For every edge (u, v) of u in residual graph it atomically subtracts the flow from the residual capacity of (u, v) and adds (atomically) it to the residual capacity of (v, u). Kernel2 (Algorithm 22) performs the push operation.

Algorithm 23 changes the state of each vertex. The activity mask is set to either 0, 1 or 2 states reflecting relabel, push and saturation states based on the excess flow, residual edge capacities and height mismatches at each vertex. Each vertex sets the termination flag to false if its state undergoes a change.

The operations terminate when there is no change in the activity mask. This does not necessarily occur when all nodes have saturated. Due to saturation of edges, some unsaturated nodes may get cutoff from sink, these nodes do not contribute any further and thus are not actively taking part in the process, consequently their state does not change which can lead to an infinite loop if termination is based on saturation of nodes. Results of this implementation are given in Figure 17. Figure 18 shows the behavior of our implementation with varying m and k in accordance to Algorithm 20.

Algorithm 22 KERNEL2_MAXFLOW

```
1: $tid \leftarrow$ getThreadID
2: if $M_a[tid] = 1$ then
3:    for all neighbors $nid$ of $tid$ do
4:       if $nid \in G_f \& h_a[tid] = h_a[nid]+1$ then
5:          $minflow \leftarrow min(e_a[tid], W_a[nid])$
6:          Begin Atomic
7:          $e_a[tid] \leftarrow e_a[tid] - minflow$
8:          $e_a[nid] \leftarrow e_a[nid] + minflow$
9:          $W_a(tid,nid) \leftarrow W_a(tid,nid) - minflow$
10:         $W_a(nid,tid) \leftarrow W_a(nid,tid) + minflow$
11:         End Atomic
12:      end if
13:   end for
14: end if
```

Algorithm 23 KERNEL3_MAXFLOW

```
1: $tid \leftarrow$ getThreadID
2: for all neighbors $nid$ of $tid$ do
3:    if $e_a[tid] \leq 0$ OR $W_a(tid,nid) \leq 0$ then
4:       $state \leftarrow 2$
5:    else
6:       if $e_a[tid] > 0$ then
7:          if $h_a[tid] = h_a[nid] + 1$ then
8:             $state \leftarrow 1$
9:          else
10:            $state \leftarrow 0$
11:         end if
12:      end if
13:   end if
14: end for
15: if $M_a[tid] \neq state$ then
16:    $Terminate \leftarrow$ false
17:    $M_a[tid] \leftarrow state$
18: end if
```

10. Performance Analysis

We choose graphs representatives of real world problems. Our graph sizes vary from $1M$ to $10M$ vertices for all algorithms except APSP. The focus of our experiments is to show fast processing of general graphs. Scarpazza et al. [45] focus on improving the throughput of the Cell/B.E. for BFS. Bader and Madduri [6,8,4] use CRAY MTA−2 for BFS, STCON and MST implementations. Dehne and Götz [21] use CC−48 to perform MST. Edmonds et al. [24] use Parallel Boost graph library and Crobak et al. [19] use CRAY MTA−2 for their SSSP implementations. Yoo et al. [52] use the BlueGene/L for a BFS implementation. Though our input sizes are not comparable with the ones used in these implementations, of orders of billions of vertices and edges, we show implementations on a hardware several orders less expensive. Because of the large difference in input sizes, we do not compare our results with these implementations directly. We show a comparison of our APSP approach with Katz and Kider [33] and Buluc et al. [11] on similar graph sizes as the implementations are directly comparable.

10.1. Types of Graphs

We tested our algorithms on various types of synthetic and real world large graphs including graphs from the ninth DIMACS challenge [1]. Primarily, three generative models were used for performance analysis, using the Georgia Tech. graph generators [7]. These models approximate real world datasets and are good representatives for graphs commonly used in real world domains. We assume all graphs to be connected with positive weights. Graphs are also assumed to be undirected with a complementary edge present for every edge in the graph. Models used for graph generation are:

- Random Graphs: Random graphs have a short band of degree where all vertices lie, with a large number of vertices having similar degrees. A slight variation from the average degree results in a drastic decrease in number of such vertices in the graph.
- R-MAT [12]/Scale Free/Power law: A large number of vertices have small degree with a few vertices having large degree. This model best approximates large graphs found in real world. Practical large graphs models including, Erdös-Rényi, power-law and its derivations follow this generative model. Due to its small degree distribution over most vertices and uneven degree distribution these graphs expand slowly in each iteration and exhibit uneven load balancing. These graphs therefore are a worst case scenario for our algorithms as verified empirically.
- SSCA#2 [5]: These graphs are made up of random sized *cliques* of vertices with a hierarchical distribution of edges between cliques based on a distance metric.

10.2. Experimental Setup

Our testbed consisted of a single Nvidia GTX 280 graphics adapter with 1024MB memory on board (30 multiprocessors, 240 stream processors) controlled by a Quad Core Intel processor ($Q6600$ @ 2.4GHz) with 4GB RAM running Fedora Core 9. For CPU comparison we use the Boost C++ graph library (with the exception of BFS) compiled using *gcc* at optimization setting $-O4$. We use our own BFS implementation on CPU as it proved faster than Boost. BFS was implemented using STL and C++, compiled with *gcc* using $-O4$ optimization. A quarter Tesla S1070 1U was used for graphs larger than $6M$ in most cases, it has the same GPU as GTX 280 with 4096MB of memory.

10.3. Summary of Results

In experiments, we observed lower performance on low degree/linear graphs. This behavior is not surprising as our implementations cannot exploit parallelism on such graphs. Only two vertices are processed in parallel in each iteration for a linear graph. We also observed that vertex list compaction helps reduce running time by factor of 1.5 in such cases. Another hurdle for efficient parallel execution is large variation in degree, which slows down the execution on an SIMD machine owing to uneven load per thread. This behavior is seen in all algorithms on R-MAT graphs. Larger degree graphs benefit more using our implementations as the expansion per iteration is more, resulting in better expansion of data and thus better performance. We show relative behavior of each algorithm in the following sections, detailed timings of plots given in each section are listed in Table 3 and Table 4 in the Appendix.

10.4. Breadth First Search (BFS)

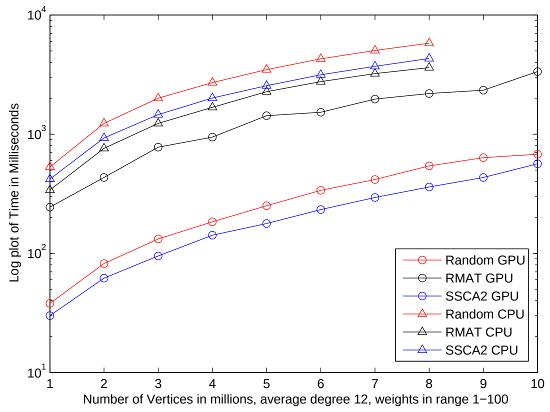


Figure 11. Breadth first search on varying number of vertices for synthetic graphs.

Figure 11 summarizes the results for our BFS implementation using the compaction process. R-MAT graphs perform poorly on the GPU as the expansion of frontier at every level is low owing to their linear nature. This prevents optimal utilization of the available computing resources as discussed in Section 10.3. Further, because of few high degree vertices, the loop in Algorithm 2 line 4 results in varying loads on different threads. Loops of non-uniform length are inefficient on SIMD architecture. Our implementation gains a speed up of about nearly 5 times for the R-MAT graphs and nearly 15 times for Random and SSCA#2 graphs over the CPU implementation.

Throughput, millions of edges processed per second (ME/S) is used as a BFS performance measure in [45]. It is, however, not a good measure for our BFS implementation as we exploit parallelism over vertices. However, for a $1M$ node graph with $12M$ edges we obtain a throughput of 315 ME/S as oppose to 101 ME/S obtained on the Cell/B.E.

10.5. ST-Connectivity (STCON)

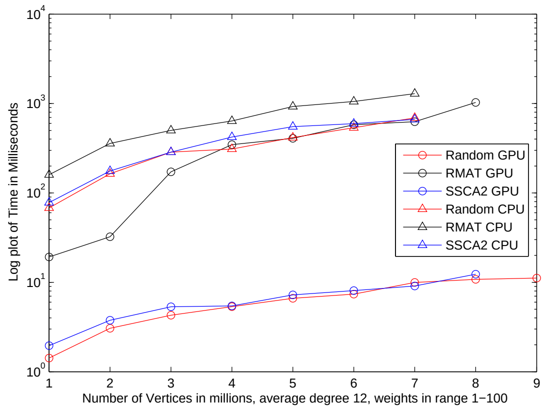


Figure 12. st-Connectivity for varying number of vertices for synthetic graph models.

Figure 12 shows results of our STCON implementation. In the worst case, our STCON implementation takes half the time of our BFS implementation, since the maximum distance between s and t can be the diameter of the graph and we start BFS from s and t concurrently for STCON. For fair comparison we averaged times of our results over 100 iterations of randomly selected s and t vertices. Because of the linear nature of R-MAT graphs we see a slower expansion of frontier vertices with uneven loops leading to load imbalance and thus poor timings.

10.6. Single Source Shortest Path (SSSP)

Single source shortest path results are summarized in Figure 13. Vertex list compaction was used to reduce number of threads executed in each kernel call for R-MAT graphs. A 40% reduction in times was observed for R-MAT graphs using compaction over the non-compacted version. R-MAT graphs however, even after compaction, perform badly on the GPU as compared to other types of graphs. On the CPU however they perform better compared to other graph models. We gain $15-20$ times speed up against boost for our SSSP implementation.

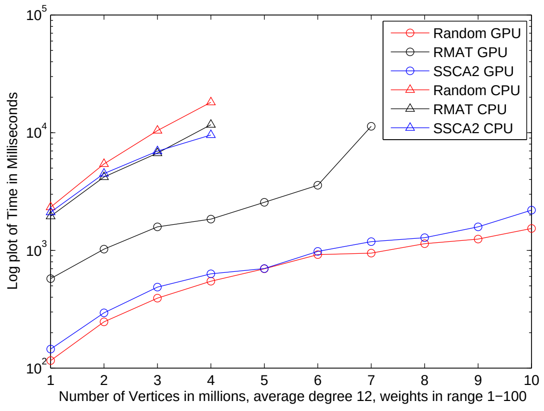


Figure 13. Single source shortest path on varying number of vertices for synthetic graph models.

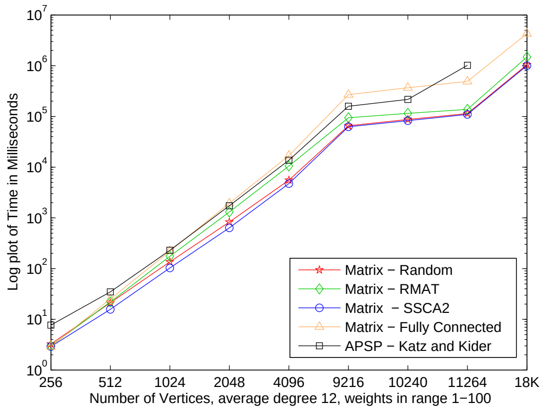


Figure 15. Comparing our matrix-based APSP approach with Katz and Kider [33] on a Quadro FX5600

10.7. All Pairs Shortest Paths (APSP)

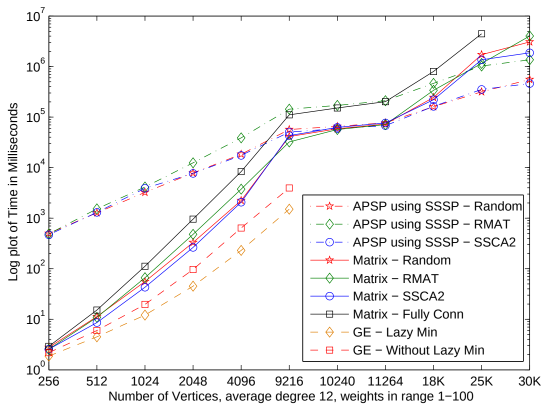


Figure 14. Comparing APSP using SSSP, APSP using matrix multiplications and APSP using Gaussian elimination [11] approaches on a GTX280 and Tesla

The SSSP-based, matrix multiplication-based and Gaussian elimination-based APSP implementations are compared in Figure 14 on GTX 280 and Tesla. Matrix multiplication APSP uses graph representation outlined in Section 7.2.1. We stream data from CPU to GPU for graphs larger than 18K for this approach. As seen from the experiments, APSP using SSSP performs badly on all types of graph, but is a scalable solution for large, low-degree graphs. For smaller graphs, matrix approach proves much faster. We do not use lazy min for fully connected graphs as it becomes an overhead for them. We are able to process a fully connected 25K graph using streaming of matrix blocks in nearly 75 minutes on a single unit of Tesla S1070, which is equivalent in computing power to the GTX280, but has 4 times the memory.

For direct comparison with Katz and Kider [33], we also show results on Quadro FX 5600. Figure 15 summarizes the results of these experiments. In case of fully connected graphs we are 1.5 times slower than Katz and Kider up to the 10K graph. We achieve a speed up of $2-4$ times over Katz and Kider for larger general graphs. The Gaussian elimination based APSP by Buluc et al. [11] is the fastest among the approaches. However, introducing the lazy minimum evaluation to their approach provides a further speed up of $2-3$ as can be seen from Figure 14 and Table 4.

10.8. Minimum Spanning Tree (MST)

Timings for minimum spanning tree implementation are summarized in Figure 16 for synthetic graphs. Our MST algorithm is not affected by the linearity of the graph, as each supervertex is processed in parallel independent to other supervertices and there is no expansion of frontier. However for R-MAT graphs we see a slowdown due to uneven loops over vertices with high de-

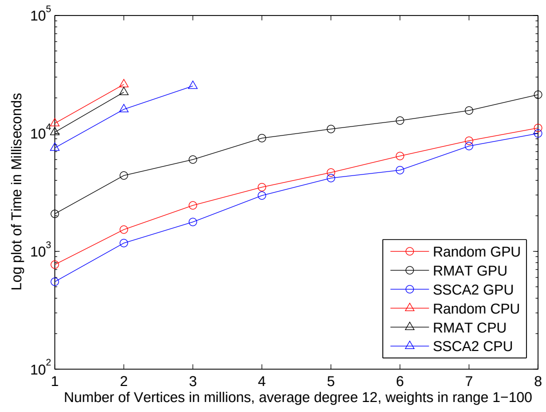


Figure 16. Minimum spanning tree results for varying number of vertices for synthetic graph models

gree, which proves inefficient on a SIMD model.

A speed up of 15 times is achieved for random and SSCA#2 graphs, a speed up of nearly 5 times is achieved over the Boost C++ implementation for R-MAT.

10.9. Max Flow (MF)

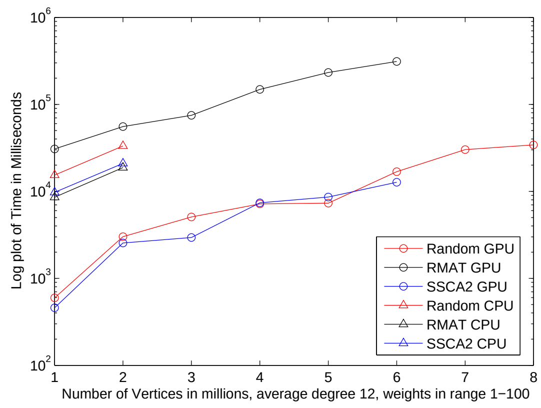


Figure 17. Maxflow results for varying number of vertices for synthetic graph models

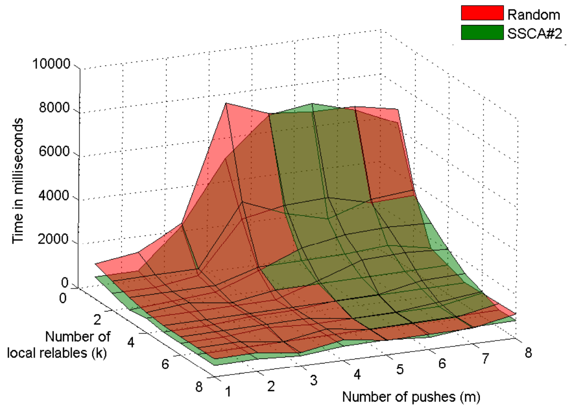
Maxflow timings for various synthetic graphs are shown in Figure 17. We average timings for max flow over 5 iterations for randomly selected source s and sink t vertices. R-MAT timings on the GPU out shoots the CPU times because of linear nature of these graphs.

We also study the behavior of our max flow implementation for varying m and k, to control the periodicity of the local and global relabeling steps as given in Algorithm 20. Figure 18 depicts this behavior for all three generative models on a 1M vertex graph. Random and SSCA#2 graphs show a similar behavior with time increasing with number of pushes for low or no local relabels. Time decreases as we apply more local relabels. We found for $m = 3$ and $k = 7$ the timing were optimal for random and SSCA#2 graphs. R-MAT graphs however exhibit very different behavior for varying m and k. For low local relabels the time increases with increasing number of pushes similar to random and SSCA#2. However as local relabels are increased we see an increase in timings. This can be attributed to the fact that linearity poses slow convergence of local relabels in case of R-MAT graphs.

10.10. Scalability

Scalability of our implementations over varying degree are summarized in Table 1. We show results for a $100K$ vertex graph with varying degree. For APSP matrix based approach results for a $4K$ graph are shown. As expected, the running time increase with increasing degree in all cases. However the scaling factor for GPU is much better than CPU in all implementations. This is because scans of edges increase with increasing degree, which is distributed over threads running in parallel, resulting in less increase in time for the GPU as compared to the CPU and thus better scalability.

Results on the ninth DIMACS challenge [1] dataset are summarized in Table 2. GPU performs worse than CPU in most implementations for these inputs. The behavior can be explained based on the linearity of these graphs. Parallel expansion is minimal for these graphs as their average degree is $2 - 3$. Minimum spanning tree however performs much faster than its CPU counterpart on these graphs owing to its inertness to linearity.



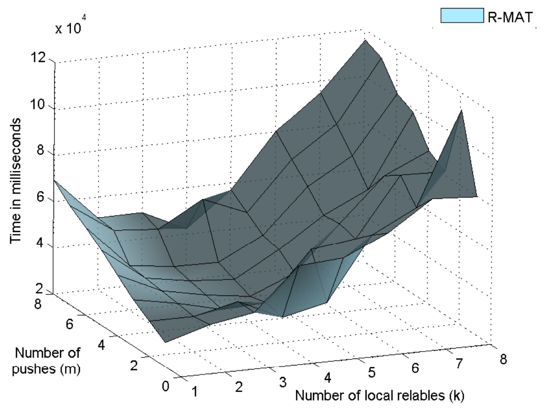


Figure 18. Maxflow behavior on a $1M$ vertex, $12M$ edge graph with varying m and k, Algorithm 20.

11. Conclusions

In this paper, we presented massively multi-threaded algorithms on large graphs for the GPU using the CUDA model. Each operation is typically broken down using a BSP-like model into several kernels executing on the GPU that are synchronized by the CPU. Using vertex list compaction we reduce the number of threads to be executed on the device and hence reducing context switching of multiple blocks for iterative frontier based algorithms. The divide and conquer approach scales well to massively multithreaded architectures for non-frontier based algorithms like MST. Where the problem can be divided to its simplest form at the lowest (vertex) level and conquered recursively further up the hierarchy. We present results on medium and large graphs as well as graphs that are random, scale-free, and inspired by real-life examples. Most of our implementations can process graphs with millions of vertices in $1 - 2$ seconds on a commodity GPU. This makes the GPUs attractive co-processors to the CPU for several scientific and engineering tasks that are modeled as graph algorithms. In addition to the performance, we believe the massively multithreaded approach we present will be applicable to the multicore and manycore architectures that are in the pipeline from different manufacturers.

Acknowledgments:
We would like to thank the Nvidia Corporation for their generous support, especially for providing hardware used in this work. We would also like to acknowledge Georgia Tech Institute for their graph generating software. We would like to thank Yokesh Kumar for his help and discussions for various algorithms given as part of this work.

REFERENCES

1. The Ninth DIMACS implementation challange on shortest paths http://www.dis.uniroma1.it/ challenge9/.
2. F. Alizadeh and A. V. Goldberg. Experiments with the push-relabel method for the maximum flow problem on a connection machine. In *DIMACS Implementation Challenge Workshop: Network Flows and Matching, Tech. Rep.*, volume 92, pages 56–71, 1991.
3. R. J. Anderson and J. C. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 168–177, 1992.
4. D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *J. Parallel Distrib. Com-*

Table 1
Scalability with varying degree on a $100K$ vertex graph, $4K$ for APSP, weights in range $1 - 100$. Times in milliseconds.

Degree	BFS GPU/CPU			STCON GPU/CPU			SSSP GPU/CPU		
	Random	R-MAT	SSCA#2	Random	R-MAT	SSCA#2	Random	R-MAT	SSCA#2
100	15/420	91/280	7/160	0.8/1.1	1.47/14.3	1.04/5.8	169/260	305/190	120/170
200	48/800	122/460	13/290	1.5/1.1	2.36/18.9	1.11/8.7	375/380	400/250	237/220
400	125/1520	163/770	24/510	2.8/1.2	3.93/27.9	1.53/8.9	898/710	504/360	474/320
600	177/2300	182/1050	38/730	4.1/1.3	5.26/41.9	2.81/9.1	1449/-	587/430	683/410
800	253/3060	210/1280	67/980	5.5/1.5	6.85/55.4	2.589/9.8	1909/-	691/540	1042/520
1000	364/-	-/-	-/-	6.8/-	-/-	-/-	2157/-	-/-	-/-

Degree	MST GPU/CPU			Max Flow GPU/CPU			APSP Matrix GPU/CPU		
	Random	R-MAT	SSCA#2	Random[§]	R-MAT	SSCA#2[§]	Random	R-MAT	SSCA#2
100	302/12150	461/10290	122/7470	808/6751	3637/4950	345/3750	6111/30880	5450/19470	3400/16390
200	369/25960	638/22180	218/16700	2976/13430	6308/9230	615/7670	8100/53480	5875/27370	5253/25860
400	1149/-	849/-	347/-	10842/32900	8502/16570	1267/17360	9034/92700	6202/38070	7078/41580
600	1908/-	1103/-	499/-	14722/-	11238/-	6018/-	9123/102383	6317/41273	7483/48729
800	2484/-	1178/-	883/-	22489/-	14598/-	8033/-	9231/126830	6391/45950	7888/57460
1000	3338/-	-/-	-/-	32748/-	-/-	-/-	9613/167630	6608/54540	8309/68580

Table 2
Results on the ninth DIMACS challenge [1] graphs, weights in range $1 - 300K$. Times in milliseconds.

Graphs with distances as weights	Vertices	Edges	Time GPU/CPU				
			BFS	STCON	SSSP	MST	Max Flow[§]
New York	264346	733846	147/20	1.25/8.8	448/190	76/780	657/420
San Fransisco Bay	321270	800172	199/20	2.2/11.3	623/230	85/870	1941/740
Colorado	435666	1057066	414/30	2.36/15.9	1738/340	116/1280	3021/2770
Florida	1070376	2712798	1241/80	5.02/37.7	4805/810	261/3840	6415/2810
Northwest USA	1207945	2840208	1588/100	7.8/48.3	8071/1030	299/4290	11018/3720
Northeast USA	1524453	3897636	2077/140	8.8/66.5	8563/1560	383/6050	18722/4100
California and Nevada	1890815	4657742	2762/180	9.4/100	11664/1770	435/7750	19327/4270
Great Lakes	2758119	6885658	5704/240	19.8/114.7	32905/2730	671/12300	21915/6360
Eastern USA	3598623	8778114	7666/400	24.4/183.8	41315/4140	1222/16280	70147/16920
Western USA	6262104	15248146	14065/800	58/379.8	82247/8500	1178/32050	184477/25360
Central USA	14081816	34292496	37936/3580	200/1691	215087/34560	3768/-	238151/-
Full USA[‡]	23947347	58333344	102302/-	860/-	672542/-	8348/-	-/-

[‡]Results taken on Tesla
[§]Max Flow results at $m = 3$ and $k = 7$

put., 65(9):994–1006, 2005.
5. D. A. Bader and K. Madduri. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In *HiPC*, volume 3769 of *Lecture Notes in Computer Science*, pages 465–476, 2005.
6. D. A. Bader and K. Madduri. Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. In *ICPP*, pages 523–530, 2006.
7. D. A. Bader and K. Madduri. GTgraph: A Synthetic Graph Generator Suite. Technical report, 2006.
8. D. A. Bader and K. 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, 2006.
9. D. A. Bader and V. Sachdeva. A Cache-Aware Parallel Implementation of the Push-Relabel Network Flow Algorithm and Experimental Evaluation of the Gap Relabeling Heuristic. In *ISCA PDCS*, pages 41–48, 2005.
10. O. Boruvka. O Jistém Problému Minimálním (About a Certain Minimal Problem) (in Czech, German summary). *Práce Mor. Prírodoved. Spol. v Brne III*, 3, 1926.
11. A. Buluc, J. R. Gilbert, and C. Budak. Gaus-

sian Elimination Based Algorithms on the GPU. Technical report, November 2008.

12. D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *In SIAM International Conference on Data Mining*, 2004.
13. B. Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000.
14. J. D. Cho, S. Raje, and M. Sarrafzadeh. Fast Approximation Algorithms on Maxcut, k-Coloring, and k-Color Ordering for VLSI Applications. *IEEE Transactions on Computers*, 47(11):1253–1266, 1998.
15. K. W. Chong, Y. Han, and T. W. Lam. Concurrent threads and optimal parallel minimum spanning trees algorithm. *J. ACM*, 48(2):297–323, 2001.
16. S. Chung and A. Condon. Parallel Implementation of Borvka's Minimum Spanning Tree Algorithm. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 302–308, 1996.
17. A. Crauser, K. Mehlhorn, and U. Meyer. A parallelization of Dijkstra's shortest path algorithm. In *In Proc. 23rd MFCS'98, Lecture Notes in Computer Science*, pages 722–731. Springer, 1998.
18. A. Crauser, K. Mehlhorn, and U. Meyer. A parallelization of dijkstra's shortest path algorithm. In *In Proc. 23rd MFCS'98, Lecture Notes in Computer Science*, pages 722–731, 1998.
19. J. R. Crobak, J. W. Berry, K. Madduri, and D. A. Bader. Advanced shortest paths algorithms on a massively-multithreaded architecture. *Parallel and Distributed Processing Symposium, International*, 0:497, 2007.
20. P. D'Alberto and A. Nicolau. R-kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. *Algorithmica*, 47(2):203–213, 2007.
21. F. Dehne and S. Götz. Practical Parallel Algorithms for Minimum Spanning Trees. In *SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, page 366, 1998.
22. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
23. J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM*, 19(2):248–264, 1972.
24. N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. Single-Source Shortest Paths with the Parallel Boost Graph Library. In *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, November 2006.
25. L. R. Ford and D. R. Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.
26. A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, 1998.
27. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 136–146, 1986.
28. N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. Memory—A memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 89, 2006.
29. Y. Han, V. Y. Pan, and J. H. Reif. Efficient Parallel Algorithms for Computing all Pair Shortest Paths in Directed Graphs. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 353–362, 1992.
30. P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *HiPC*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208, 2007.
31. M. Hussein, A. Varshney, and L. Davis. On Implementing Graph Cuts on CUDA. In *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.
32. J. Hyvonen, J. Saramaki, and K. Kaski. Efficient data structures for sparse network representation. *Int. J. Comput. Math.*, 85(8):1219–1233, 2008.

33. G. J. Katz and J. T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 47–55, 2008.
34. V. King, C. K. Poon, V. Ramachandran, and S. Sinha. An optimal EREW PRAM algorithm for minimum spanning tree verification. *Inf. Process. Lett.*, 62(3):153–159, 1997.
35. A. Lefohn, J. M. Kniss, R. Strzodka, S. Sengupta, and O. J. D. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, Jan. 2006.
36. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.
37. P. Micikevicius. General Parallel Computation on Commodity Graphics Hardware: Case Study with the All-Pairs Shortest Paths Problem. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1359–1365, 2004.
38. A. Munshi. OpenCL: Parallel computing o the GPU and CPU. In *SIGGRAPH, Tutorial*, 2008.
39. P. J. Narayanan. Single Source Shortest Path Problem on Processor Arrays. In *Proceedings of the Fourth IEEE Symposium on the Frontiers of Massively Parallel Computing*, pages 553–556, 1992.
40. P. J. Narayanan. Processor Autonomy on SIMD Architectures. In *International Conference on Supercomputing*, pages 127–136, 1993.
41. A. S. Nepomniaschaya and M. A. Dvoskina. A simple implementation of Dijkstra's shortest path algorithm on associative parallel processors. *Fundam. Inf.*, 43(1-4):227–243, 2000.
42. NVIDIA. NVIDIA CUDA Programming Guide 2.0 http://www.nvidia.com/object/cuda_develop.html/.
43. M. Pal and G. P. Bhattacharjee. An optimal parallel algorithm for all-pairs shortest paths on unweighted interval graphs. *Nordic J. of Computing*, 4(4):342–356, 1997.
44. O. Reingold. Undirected ST-connectivity in log-space. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 376–385, 2005.
45. D. P. Scarpazza, O. Villa, and F. Petrini. Efficient Breadth-First Search on the Cell/BE Processor. *IEEE Trans. Parallel Distrib. Syst.*, 19(10):1381–1395, 2008.
46. S. Sengupta, M. Harris, Y. Zhang, and O. J. D. Scan primitives for GPU computing. In *GH 07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, 2007.
47. T. Takaoka. An Efficient Parallel Algorithm for the All Pairs Shortest Path Problem. In *WG '88: Proceedings of the 14th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 276–287, 1989.
48. V. Trifonov. An O(log n log log n) space algorithm for undirected st-connectivity. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 626–633, 2005.
49. G. Venkataraman, S. Sahni, and S. Mukhopadhyaya. A Blocked All-Pairs Shortest-Paths Algorithm. *Journal of Experimental Algorithmics*, 8:2003, 2003.
50. V. Vineet and P. Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *CVGPU08*, pages 1–8, 2008.
51. V. Volkov and J. Demmel. LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs. Technical report, EECS Department, University of California, Berkeley, May 2008.
52. A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 25, 2005.
53. Y. Zhang and E. Hansen. Parallel Breadth-First Heuristic Search on a Shared-Memory Architecture. In *AAAI-06 Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications*, 2006.

APPENDIX

Table 3
Summary of results for synthetic graphs. Times in milliseconds

Algo	Graph	Number of Vertices, average degree 12, weights varying from 1 to 100									
	Type	1M	2M	3M	4M	5M	6M	7M	8M	9M	10M
BFS*	Random GPU†	38	82	132	184	251	338	416	541	635‡	678‡
	Random CPU	530	1230	2000	2710	3480	4290	5040	5800	-	-
	R-MAT GPU†	244	433	778	944	1429	1526	1969	2194	2339‡	3349‡
	R-MAT CPU	340	760	1230	1680	2270	2760	3220	3620	-	-
	SSCA#2 GPU†	30	62	95	142	178	233	294	360	433‡	564‡
	SSCA#2 CPU	420	930	1460	2010	2550	3150	3710	4310	-	-
STCON	Random GPU	1.42	3.06	4.28	5.34	6.62	7.37	9.96	10.8	11.15	-
	Random CPU	68	164	286	310	416	536	692	-	-	-
	R-MAT GPU	19.2	32.37	172.1	347.4	408.3	579.1	626	1029	-	-
	R-MAT CPU	160	358	501	638	926	1055	1288	-	-	-
	SSCA#2 GPU	1.96	3.76	5.33	5.44	7.23	8.08	9.1	12.33	-	-
	SSCA#2 CPU	78	176	286	422	552	595	665	-	-	-
SSSP	Random GPU	116	247	393	547	698	920	947	1140	1247	1535
	Random CPU	2330	5430	10420	18130	-	-	-	-	-	-
	R-MAT GPU†	576	1025	1584	1842	2561	3575	11334	-	-	-
	R-MAT CPU	1950	4200	6700	11680	-	-	-	-	-	-
	SSCA#2 GPU	145	295	488	632	701	980	1187	1282	1583	2198‡
	SSCA#2 CPU	2110	4490	6970	9550	-	-	-	-	-	-
MST	Random GPU†	770	1526	2452	3498	4654	6424‡	8670‡	11125‡	-	-
	Random CPU	12160	26040	-	-	-	-	-	-	-	-
	R-MAT GPU†	2076	4391	5995	9102	10875	12852	15619‡	21278‡	-	-
	R-MAT CPU	10230	22340	-	-	-	-	-	-	-	-
	SSCA#2 GPU†	551	1174	1772	2970	4173	4879	7806‡	9993‡	-	-
	SSCA#2 CPU	7540	15980	25230	-	-	-	-	-	-	-
MF	Random GPU§	598	3013	5083	7179	7323‡	16871‡	30201‡	34253‡	-	-
	Random CPU	15390	33290	-	-	-	-	-	-	-	-
	R-MAT GPU	30743	55514	74767	148627	232789‡	311267‡	-	-	-	-
	R-MAT CPU	8560	18770	-	-	-	-	-	-	-	-
	SSCA#2 GPU§	459	2548	2943	7388	8606‡	12742‡	-	-	-	-
	SSCA#2 CPU	9760	20960	-	-	-	-	-	-	-	-

Table 4
Summary of results for synthetic graphs APSP approaches. Times in milliseconds

APSP	Graph	Number of Vertices, average degree 12, weights in range $1-100$										
	Type	256	512	1024	2048	4096	9216	10240	11264	18K	25K	30K
Using SSSP GTX 280	Random	499	1277	3239	7851	18420	56713	65375	77265	160608	316078	556313
	R-MAT	489	1531	4145	12442	38812	143991	170121	211277	465037	1028275	1362119
	SSCA#2	469	1300	3893	7677	17450	50498	58980	67794	163081	353166	461901
Matrix GTX 280	Random	2.77	11.3	55.7	330.4	2240.8	41150	58889	72881‡	244264‡	1724970‡ ¶	3072443‡ ¶
	R-MAT	2.54	10.9	66.2	478	3756	32263	56906	71035‡	339188‡	1152989‡ ¶	4032675‡ ¶
	SSCA#2	2.55	8.6	42.9	263.6	2063.7	43045	62517	76399‡	220868‡	1360469‡ ¶	1872394‡ ¶
	Fully Conn.	2.9	15.2	112	959	8363	110658	151820	202118‡	799035‡	4467455‡ ¶	-
Matrix FX 5600	Random	3.25	21.3	136.3	827.9	5548	65552	87043	113993	1048598¶	-	-
	R-MAT	2.99	22	174.8	1307.6	10534	94373	115294	137854	1487025¶	-	-
	SSCA#2	2.91	15.78	103.2	635.3	4751	62555	82708	109744	1001212¶	-	-
	Fully Conn.	2.9	25.1	221.5	1941	16904	268757	368397	490157	4300447¶	-	-
GE Based Lazy Min GTX 280	Random	1.8	4.4	12.1	44.9	230	1505	-	-	-	-	-
	R-MAT	1.8	4.5	12.1	45	230	1505	-	-	-	-	-
	SSCA#2	1.8	4.5	12.1	44	230	1497	-	-	-	-	-
GE Based Buluc [11] GTX 280	Random	2.18	6	19.64	96.5	639	3959	-	-	-	-	-
	R-MAT	1.86	5.9	19.1	96	638	3959	-	-	-	-	-
	SSCA#2	2.18	5.9	19.67	96	638	3965	-	-	-	-	-
Katz [33]	-	7.7	34.9	230.1	1735.6	13720	158690	216400	1015700	-	-	-

*CPU implementation is ours
†Using Compaction process
‡Results taken on Tesla
§Max Flow results at $m = 3$ and $k = 7$
¶Results using streaming from CPU to GPU memory