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

Abstract—The Graphics Processing Units (GPUs) provide high computation power at a low cost and is an important compute accelerator with a massively multithreaded architecture. In this paper, we present fast implementations of common graph operations like breadth-first search, st-connectivity, single-source shortest path, all-pairs shortest path, minimum spanning tree, and maximum flow for undirected graphs on the GPU using the CUDA programming model. Our implementations exhibit high performance, especially on large graphs. We use two data-parallel programming methodologies for these algorithms. One is an iterative, mask-based approach that processes valid data elements like vertices and edges using independent threads for each. The other is a divide-and-conquer approach that reduces the problem into smaller problems that are handled later using the same approach. Parallel algorithms for such problems have been reported in the literature before, especially on supercomputers. 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. We show the impact of our implementations on random, scale-free, and real-life graphs of up to millions of vertices on high-end and low-end GPUs. The availability and spread of GPUs to desktops and laptops make them ideal candidates to accelerate graph operations over the CPU-only implementations. Practical implementations of basic operations go a long way in realizing their potential.

Index Terms—Graph Algorithms, GPU, CUDA.

I. INTRODUCTION

Modern Graphics Processing Units (GPUs) provide high computation power at low costs and have been described as desktop supercomputers. Several high-performance, general data processing algorithms such as sorting, matrix multiplication, etc., have been developed for the GPUs. We present a set of general graph algorithms on the GPU using the CUDA programming model. 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 [1], [2] 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 [3], [4], [5]. 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.

The GPUs expose a general, data-parallel programming model today in the form of CUDA and CAL. The recently adopted OpenCL standard [6] provides a common computing model to all GPUs and 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. 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 up to 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 supercomputers, which typically have a small number of powerful cores. The parallelizing approach there is 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 on irregular algorithms, using \( O(V) \) or \( O(E) \) simultaneous threads, breaking down and working at the problem at its smallest constituent.

In this paper, we present a set of general graph algorithms on the GPU, using the CUDA programming model. We adopt two data parallel approaches in this paper: the iterative mask based approach and the divide and conquer approach to solve irregular graph algorithms. Specifically, we present implementations of breadth first search (BFS), st-connectivity (STCON), single source shortest path (SSSP) and maximum flow (MF) using the iterative mask based approach. And the implementation of minimum spanning tree (MST) using the...
divide-and-conquer approach. We compare various approaches to solve the all pairs shortest path (APSP) problem including iterative, recursive and a matrix multiplication approach. Our implementations exhibit high performance, especially on large graphs. We show experiments on random, scale-free, and real-life graphs of up to millions of vertices. 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 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 [7]. Compared to the CPU a speed up of nearly $10 - 15$ times over the Boost Graph Library is achieved for all algorithms reported in this paper.

The prevalence of GPUs on desktops and laptops today make them feasible accelerators for a wide variety of applications including common graph algorithms. Comparison of timing with the CPU implementations gives an indication of the accelerated performance one can get using low-end and high-end GPUs. Our BFS and SSSP code is already being used by different users and has been included in the Rodinia benchmark [8]. We will make all code available to whoever is interested in using them.

II. COMPUTE UNIFIED DEVICE ARCHITECTURE

In this section we present a small overview of the CUDA programming and hardware models. Please see [9] for more details about CUDA programming. Figure 1 depicts the CUDA programming model, mapping a software CUDA block to a hardware CUDA multiprocessor. A number of blocks can be assigned to a multiprocessor and they are time-shared internally by the CUDA programming environment. Each multiprocessors consists of a series processors which run the threads present inside a block in a time-shared fashion based on the warp size of the CUDA device. Each multiprocessor further contains a small shared memory, a set of 32-bit registers, texture, and constant memory caches common to all processors inside it. Processors in the multiprocessor executes the same instruction on different data, which makes CUDA an SIMD model. Communication between multiprocessors is through the device global memory which is accessible to all processors within a multiprocessor.

The CUDA API provides 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 code executes as threads running in parallel in batches of warp size, time-shared on the CUDA processors. Each thread can use a number of private registers for its computation. 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. The available shared memory and registers are split equally amongst all blocks that timeshare a multiprocessor. An execution on a device generates a number of blocks, collectively known as a grid (Figure 1).

Each thread executes a single instruction set called the kernel. Threads and blocks are given a unique ID that can be accessed within the thread during its execution. These can be used by a thread to perform the kernel task on its part of the data resulting in an SIMD execution. Algorithms may use multiple kernels, which share data through the global memory and synchronize their execution either at the end of each kernel or forcefully using barriers.

III. REPRESENTATION AND PROGRAMMING METHODOLOGY

We adopt two data parallel programming approaches in our implementations.

- The iterative mask based approach, in which a set of vertices take part in execution at each iteration. We process each vertex in the mask in parallel. Synchronization occurs after execution of all vertices at every iteration. We use this approach in implementing BFS, STCON, SSSP and Maximum Flow.
- The divide-and-conquer approach. Here we divide the problem into its simplest constituent and process each constituent in parallel while merging them recursively as we move up the hierarchy. We give one thread to each constituent and process them in parallel. This approach is used in the implementation of the Minimum Spanning Tree.

In implementing all pairs shortest paths we compare implementations using both approaches, iterative from our group and recursive from Buluc et al. [10], along-with another matrix multiplication approach. In all implementations we map the problem to a data parallel scenario. We assume there can exist a thread for each vertex/edge in the graph. This assumption is in contrast with previous supercomputing approaches, where the problem is mapped onto a fixed set of processes. A bulk synchronous parallel programming model is used in implementing all algorithms.

A. Graph Representation

Efficient data structures for graph representation have been studied in depth. Complex data structures like hash tables [11] have been used for efficiency on the CPU. Creating an efficient data structure on the GPU memory model, however, is a challenging problem [12], [13].

Adjacency matrix representation is not suitable for large sparse graphs because of its $O(V^2)$ space requirements, restricting the size of graphs that can be handled by the GPU. Adjacency list is a more practical representation for large sparse 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). CUDA model treats memory as general arrays and can support such representation efficiently. We assume the GPU can hold entire data into memory using this representation.

Table I states the variables used for representing graph in adjacency list format. 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 $V_a$, $W_a$ holds the edge weight for each edge. We deal with undirected graphs resulting in each edge
The CUDA Hardware Model

The CUDA Device, with a number of Multiprocessors

The device global memory

Grid with multiple blocks resulting from a Kernel call

The CUDA Programming Model

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_a$</td>
<td>Holds starting index of edge list in $E_a$</td>
</tr>
<tr>
<td>$E_a$</td>
<td>Holds vertex id of outgoing vertex</td>
</tr>
<tr>
<td>$W_a$</td>
<td>Holds the weight of every edge</td>
</tr>
<tr>
<td>Terminate</td>
<td>Global variable written over by all threads to achieve consensus using logical OR</td>
</tr>
</tbody>
</table>

Table I

General variables used in graph representation and the CPU_SKELETON code

having 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 similar to the BFS problem itself.

This representation is used for all algorithms reported in this paper except in all pairs shortest paths matrix multiplication method (explained in section VII). A block-divided adjacency matrix representation is used to exploit better cache efficiency there. We do not assume the entire matrix can be held in the GPU memory. We stream parts of the matrix from the CPU to GPU memory. APSP output requires $O(V^2)$ space and thus adjacency matrix proves a more suitable representation.

B. 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 PE waits for all PEs to finish their task. 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 ← 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 Kerneln 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
graphs where parallelism expands slowly, compaction makes thread execution and time taken by scan and compacting. For new (Figure 3) creating the mapping of active mask

We compact all entries in the activity mask to an index and a new index amongst the currently active vertices. Each. This establishes a mapping between the original vertex number of active vertices as well as gives an ordinal number to each. 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 improves if we deploy only as many threads as the active vertices, reducing the number of blocks and thus time sharing on the CUDA device.

We assign threads to an attribute of the graph (vertex, color etc.) in all implementations to exploit maximum data-parallelism. This leads to an execution of maximum O(V) parallel threads, though they are time-shared by the CUDA environment. The number of active vertices, however, varies in each iteration of execution. Active vertices are indicated in an activity mask, which holds a 1 for each active vertex. 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 improves if we deploy only as many threads as the active vertices, reducing the number of blocks and thus time sharing on the CUDA device.

Activity mask (Active Threads)

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

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

A scan operation [14] on the activity mask determines the number of active vertices as well as gives an ordinal number to each. This establishes a mapping between the original vertex index and a new index amongst the currently active vertices. We compact all entries in the activity mask to an active mask (Figure 3) creating the mapping of new thread IDs to old vertex IDs. Each thread can now find its vertex id by looking at its respective indices in the global memory.

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

IV. 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 [15]. Zhang et al. [16] gave a heuristic search for BFS using level synchronization. Bader et al.[3] implement BFS for the CRAY MTA−2 supercomputer and Yoo et al. [5] 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.

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

GPU Implementation

Table II states the variables used in BFS 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, 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 \( C_a \).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F_a )</td>
<td>Holds active vertices in each iteration.</td>
</tr>
<tr>
<td>( X_a )</td>
<td>Holds the visited state for each vertex.</td>
</tr>
<tr>
<td>( C_a )</td>
<td>Holds the BFS cost per vertex.</td>
</tr>
<tr>
<td>( F_{ua} )</td>
<td>Used to resolve read after write inconsistencies.</td>
</tr>
</tbody>
</table>

**Algorithm 2 KERNEL1_BFS**

```
1: tid ← getThreadId
2: if \( F_a[tid] \) then
3: \( F_a[tid] ← false \)
4: \( X_a[tid] ← false \)
5: \( C_a[tid] ← false \)
6: \( F_{ua}[tid] ← false \)
7: \( C_{ua}[tid] ← false \)
```

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 \( X_a \). The vertex thread sets the termination flag to false if the vertex is added to the frontier array, \( F_a \).

**Algorithm 3 KERNEL2_BFS**

```
1: tid ← getThreadId
2: if \( F_{ua}[tid] \) then
3: \( F_a[tid] ← true \)
4: \( X_a[tid] ← true \)
5: \( F_{ua}[tid] ← false \)
6: \( C_{ua}[tid] ← false \)
7: \( C_a[tid] ← false \)
8: \( F_{ua}[tid] ← false \)
```

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.

V. ST-CONNECTIVITY (STCON)

The st-Connectivity problem resembles the BFS problem closely. Given an unweighted graph \( G(V, E) \) and two vertices, \( s \) and \( t \), find a path from \( s \) to \( t \) assuming one exists. Bader et al. [3] 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 undirected graphs.

Our approach starts BFS concurrently from \( s \) and \( t \) with Red and Green colors assigned respectively to them. In each iteration, colors are propagated to neighbors along with the BFS cost. Termination occurs when both colors meet. Evidently, both BFS frontiers hold the smallest distance to the current processing vertex from their respective source vertices. The smallest path from \( s \) to \( t \) is reached when frontiers come in contact with each other. Figure 5 depicts two termination conditions due to merging of frontiers, either at a vertex or an edge. We set the Terminate variable to false in this implementation and each thread writes a true in this variable if termination condition is reached.

![Fig. 5. Parallel st-connectivity with colors expanding from s and t vertices.](image)

**GPU Implementation**

Along with \( V_a \), \( E_a \), \( F_a \), and \( C_a \) we keep two boolean arrays \( R_a \) and \( G_a \), for red and green colors, of size \(|V|\) as the vertices visited by \( s \) and \( t \) frontiers respectively, see Table III. 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 and avoid read after write inconsistencies, alternate updating arrays \( R_{ua} \), \( G_{ua} \) and \( F_{ua} \) of size \(|V|\) are also used in each iteration. Variables \( R_f \) and \( G_f \) keep track of the Red and Green frontier lengths at current execution.
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, 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. We need not update both frontiers for termination at an edge, only the Red frontier is updated in this case as shown in Algorithm 4, line 7. 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.

**Algorithm 4 KERNEL1_STCON**

1. $tid ← \text{getThreadID}$
2. if $F_a[tid]$ then
3. $F_a[tid] ← false$
4. for all neighbors nid of $tid$ do
5. if $(G_a[nid] | R_a[nid])$ then
6. if $(R_a[tid] & G_a[nid])$ then
7. $R_f ← C_a[tid]+1$
8. Terminate $← true$
9. end if
10. if $(G_a[tid] & R_a[nid])$ then
11. Terminate $← true$
12. end if
13. else
14. if $G_a[tid]$ then $G_{ua}[nid] ← true$
15. if $R_a[tid]$ then $R_{ua}[nid] ← true$
16. $F_{ua}[nid] ← true$
17. $C_a[nid] ← C_a[tid]+1$
18. end if
19. end for
20. end if

**Algorithm 5 KERNEL2_STCON**

1. $tid ← \text{getThreadID}$
2. if $F_{ua}[tid]$ then
3. $F_{ua}[tid] ← true$
4. if $R_{ua}[tid]$ then
5. $R_a[tid] ← true$
6. $R_f ← C_a[tid]$
7. end if
8. if $G_{ua}[tid]$ then
9. $G_a[tid] ← true$
10. $G_f ← C_a[tid]$
11. end if
12. $F_{ua}[tid] ← false$
13. $R_{ua}[tid] ← false$
14. $G_{ua}[tid] ← false$
15. if $G_{ua}[tid] & R_{ua}[tid]$ then Terminate $← true$
16. end if

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. Variables $R_f$ and $G_f$ are updated to reflect the current frontier lengths. The length of the path between $s$ and $t$ can be obtained by adding $R_f$ and $G_f$. The algorithm needs a maximum of the radius of the Graph $G$ iterations to terminate.

**VI. SINGLE SOURCE SHORTEST PATH (SSSP)**

The sequential solution to single source shortest path problem comes from Dijkstra [17]. 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. [19] parallelized Dijkstra’s algorithm for associative parallel processors. Narayanan [20] solves the SSSP problem for processor arrays. Although parallel implementations of the Dijkstra’s SSSP algorithm are reported [21], an efficient PRAM algorithm does not exist [22].

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. In our implementation 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.

**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. Table IV states the variables and their usage. Initially the mask $M_a$ contains the source vertex. Each vertex looks at its entry in the mask $M_a$. If true, it updates the cost of its neighbors if greater than its own cost plus the edge weight to the corresponding neighbor in an alternate updating cost array $C_{ua}$. The alternate cost array $C_{ua}$ is used to resolve read after write inconsistencies in the global memory. Updates in $C_{ua}$ need to lock the memory location before modifying the cost value, as many threads may write different values at the same location concurrently. We use the $\text{atomicMin}$ function supported on CUDA 1.1 hardware (lines 5–9, Algorithm 6) to resolve this.

**Algorithm 6 KERNEL1_SSSP**

```plaintext
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: \hspace{1em} \text{Begin Atomic}
6: \hspace{2em} if $C_{ua[nid]} > C_a[tid] + W_a[nid]$ then
7: \hspace{3em} $C_{ua[nid]} \leftarrow C_a[tid] + W_a[nid]$
8: \hspace{2em} end if
9: \hspace{1em} \text{End Atomic}
10: end for
11: end if
```

**Algorithm 7 KERNEL2_SSSP**

```plaintext
1: $tid \leftarrow \text{getThreadID}$
2: if $C_a[tid] > C_{ua[tid]}$ then
3: $C_a[tid] \leftarrow C_{ua[tid]}$
4: $M_a[tid] \leftarrow \text{true}$
5: $\text{Terminate} \leftarrow \text{false}$
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. The algorithms takes the order of diameter of the graph to converge to equilibrium.

**VII. 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 [23], [24], [25]. Micikevicius [26] 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 [27] 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: \hspace{1em} for all Elements of $A$, in parallel do
4: \hspace{2em} $A[i, j] \leftarrow \min(A[i, j], A[i, k] + A[k, j])$
5: \hspace{1em} end for
6: end for

**A. APSP using SSSP**

Reducing space requirements on the CUDA device directly translates to handle 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 11(d).

**B. APSP as Matrix Multiplication**

Katz and Kider [7] 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 [28]), 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:          $D_{i,j}^{m} \leftarrow \infty$
6:       for $k \leftarrow 1$ to $V$ do
7:          $D_{i,j}^{m} \leftarrow \min(D_{i,j}^{m}, D_{i,j}^{(m-1)} + A_{k,j})$
8:    end for
9: end for
10: end for

We modify the parallel version of matrix multiplication proposed by Volkov and Demmel [29] for our APSP solution. We replace the multiplication and addition operations in Volkov and Demmel kernel to addition and min operations. The kernel is looped over $\log V$ times using an outer loop to solve the APSP problem.

**Fig. 7.** Blocks for matrix multiplication by Volkov and Demmel [29] modified to stream from CPU to GPU.

**Cache Efficient Graph Representation:** For matrix multiplication based APSP, we use an adjacency matrix to represent graph. Figure 7 depicts an extension of the 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 boost in performance.

**Streaming Blocks:** To handle large graphs, the adjacency matrix present in the CPU memory is divided into rectangular row and column sub-matrices. 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 sub-matrices of the original matrix present in the device memory. For every row sub-matrix $R$ we iterate through all column sub-matrices $C$ 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 $C$ sub-matrices of size $(B \times V)$ and $(V \times B)$ respectively such that

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

where $B$ is the block size. A total of

$$\log V \left( V^3 \frac{1}{B} + V^2 \right) \equiv O \left( \log V \left( V^3 \frac{1}{B} \right) \right)$$

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 as will be shown in Section X.

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

**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 remaining entries of the adjacency matrix are infinity. With entries involving infinity, additions and subsequent minimum finding can be skipped altogether without affecting correctness. We, therefore, evaluate the addition and 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, however, making the running time degree-dependent.

**GPU Implementation:** Let $R$ be the row and $C$ be the column sub-matrices 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 using modified Volkov and Demmel kernel, while the rest of the code executes on the CPU. Shared memory is used as a user managed cache to
problem. In a serial setting it takes Borůvka’s algorithm [32] is a popular solution to the MST where α lem, proposed by Bernard Chazelle [31], is O\((\log V)\) time. Numerous parallel variations of this algorithm also exist [33]. Chong et al. [34] report a EREW PRAM algorithm requiring O\((\log V)\) time and O\((V \log V)\) work. Bader et al. [35] design a fast algorithm for symmetric multiprocessors with O\(((V + E)/p)\) lookups and local operations for a p

processor machine. Chung et al. [36] 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 [37].

We implement a modified parallel Borůvka algorithm on CUDA using the divide-and-conquer approach similar to the algorithm reported by Johnson and Metaxas in [38]. We initiate colored trees from all vertices. Grow individual trees by adding the minimum weighted edge to the minimum outgoing vertex and merge colors when trees come in contact with each other. Cycles are removed explicitly in each iteration. Connected components are found via color propagation, an approach similar to our SSSP implementation (section VI).

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.

C. Gaussian Elimination Based APSP

In a parallel work, Buluc et al. [10] formulate a fast recursive APSP algorithm based on Gaussian elimination. They cleverly extend the R-Kleene [30] 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 [29]. 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. Our APSP specific optimizations can improve performance over their native implementation, for example, we incorporated the lazy minimum evaluation into the Volkov and Demmel kernel used their approach and obtained a speed up of more than 2 over their native code. Their approach is memory heavy and is best suited when the adjacency matrix can fit completely in the GPU device memory. The approach involves several matrix multiplications and additions. Extending which 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 and 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 11(d). Comparison of matrix multiplication approach with Katz and Kider is given in Figure 11(e). Behavior of the matrix approach with varying degree is reported in Table VII.

VIII. MINIMUM SPANNING TREE (MST)

Best time complexity for a serial solution to the MST problem, proposed by Bernard Chazelle [31], is O\((E\alpha(E, V))\), where α is the functional inverse of Ackermann’s function. Borůvka’s algorithm [32] is a popular solution to the MST problem. In a serial setting it takes O\((E \log V)\) time. Numerous parallel variations of this algorithm also exist [33]. Chong et al. [34] design a fast algorithm for symmetric multiprocessors with O\(((V + E)/p)\) lookups and local operations for a p
Algorithm 10 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 color index to the new color of its new supervertex.
18:   end while
19: Copy $Mst_a$ to CPU memory as output.

Algorithm 11 KERNEL1_MST

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

Algorithm 12 KERNEL2_MST

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

B. Finding and Removing Cycles

As $C$ edges are added for $C$ colors, at least 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 at most 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 whose edges form cycles.

Each added edge increments the degree of both its supervertices using color of the supervertex as its index in $Deg_a$ (Algorithm 13). After color propagation, i.e., merger of supervertices (Section VIII-C), all 1-degree supervertices nullify their added edge in $NMst_a$. They also decrement their own degree and the degree of their added edge’s outgoing supervertex in $Deg_a$.

### TABLE V

<table>
<thead>
<tr>
<th>Variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_a$</td>
<td>Holds color values</td>
</tr>
<tr>
<td>$Ci_a$</td>
<td>Holds index of color for every vertex</td>
</tr>
<tr>
<td>$Ac_a$</td>
<td>Holds active colors out of $C_a$</td>
</tr>
<tr>
<td>$NMst_a$</td>
<td>Holds newly selected edges per iteration</td>
</tr>
<tr>
<td>$Mst_a$</td>
<td>Edges selected in MST upto current iteration</td>
</tr>
<tr>
<td>$Deg_a$</td>
<td>Degree of every supervertex in current iteration</td>
</tr>
<tr>
<td>$Cy_a$</td>
<td>Used to eliminate cycle making edges</td>
</tr>
<tr>
<td>$PrevC_a$</td>
<td>Stores previous state of $C_a$ in each iteration</td>
</tr>
</tbody>
</table>

A. Finding Minimum Weighted Edge

Each vertex finds its minimum weighted outgoing edge using edge weights $W_a$. The index of this edge is written atomically to the color index of the supervertex in global memory. Multiple edges in a supervertex can have minimum weights, the one with minimum outgoing color is selected. Algorithm 11 finds the minimum weighted edge for each supervertex. Please note lines 10 – 14 in the pseudo code (Algorithm 11) are implemented as multiple atomic operations in practice.

Algorithm 12 adds the minimum weighted edge from each supervertex to the final MST output array $Mst_a$. This kernel is important as we cannot add an edge to $Mst_a$ until all vertices belonging to the supervertex have voted for their lowest weighted edge. This Kernel executes for all supervertices (or active colors) after KERNEL1_MST executes for every vertex of the graph.
we use an alternate color array to store intermediate values
the vertices to the lower one. As in the SSSP implementation,
executes for each added edge and updates the colors of both
anism is similar to our SSSP step. Kernel
4

C. Merging Supervertices

Fig. 9. For C colors, C edges are added, resulting in multiple cycles. One
edge per cycle must be removed.
(Algorithm 15). 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\textsubscript{a}
for each supervertex. Old colors are also needed after color
propagation to identify supervertices while decrementing the
degrees. We preserve old colors before propagating new colors
in an alternate color array PrevC\textsubscript{a} (Algorithm 13).

After removing 1-degree supervertex edges, resulting super-
vertices write their edge from NMst\textsubscript{a} to their new color
location in Cy\textsubscript{a} (Algorithm 16), after new colors have been
assigned to supervertices of each disjoint component using
Algorithm 14. One edge of the two, per disjoint component
cycle, survives this step. Since both edges have equal weights,
no preference is given over edges. Edges in Cy\textsubscript{a} are then
removed from the output MST array Mst\textsubscript{a} (Algorithm 17)
resulting in cycle free set of MST edges.

Algorithm 13 KERNEL\textsubscript{3}_MST
1: col ← getThreadID
2: if Ac\textsubscript{a}[col] then
3: col2 ← C\textsubscript{a}[Ci\textsubscript{a}[E\textsubscript{a}[NMst\textsubscript{a}[col]]]]
4: Begin Atomic
5: Deg\textsubscript{a}[col] ← Deg\textsubscript{a}[col]+1
6: Deg\textsubscript{a}[col2] ← Deg\textsubscript{a}[col2]+1
7: End Atomic
8: end if
9: PrevC\textsubscript{a}[col] ← C\textsubscript{a}[col]

D. 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 established
during the previous step. This link must be traversed by each
vertex to find the lowest color it should point to. The colors are
set same as the index initially, leading to same color and index
and to resolve read after write inconsistencies (not shown in
Algorithm 14).

Algorithm 14 KERNEL\textsubscript{4}_MST
1: cid ← getThreadID
2: col ← C\textsubscript{a}[cid]
3: if Ac\textsubscript{a}[col] then
4: cid2 ← Ci\textsubscript{a}[E\textsubscript{a}[NMst\textsubscript{a}[col]]]
5: Begin Atomic
6: if C\textsubscript{a}[cid] > C\textsubscript{a}[cid2] then
7: C\textsubscript{a}[cid] ← C\textsubscript{a}[cid2]
8: end if
9: if C\textsubscript{a}[cid2] > C\textsubscript{a}[cid] then
10: C\textsubscript{a}[cid2] ← C\textsubscript{a}[cid]
11: end if
12: End Atomic
13: end if

Algorithm 15 KERNEL\textsubscript{5}_MST
1: cid ← getThreadID
2: col ← PrevC\textsubscript{a}[cid]
3: if Ac\textsubscript{a}[col] & Deg\textsubscript{a}[col] = 1 then
4: col2 ← PrevC\textsubscript{a}[Ci\textsubscript{a}[E\textsubscript{a}[NMst\textsubscript{a}[cid]]]]
5: Begin Atomic
6: Deg\textsubscript{a}[col] ← Deg\textsubscript{a}[col]−1
7: Deg\textsubscript{a}[col2] ← Deg\textsubscript{a}[col2]−1
8: End Atomic
9: NMst\textsubscript{a}[col] ← φ
10: end if

Algorithm 16 KERNEL\textsubscript{6}_MST
1: cid ← getThreadID
2: col ← PrevC\textsubscript{a}[cid]
3: if Ac\textsubscript{a}[col] & NMst\textsubscript{a}[col] ≠ φ then
4: newcol ← C\textsubscript{a}[Ci\textsubscript{a}[E\textsubscript{a}[NMst\textsubscript{a}[col]]]]
5: Cy\textsubscript{a}[newcol] ← NMst\textsubscript{a}[col]
6: end if

Algorithm 17 KERNEL\textsubscript{7}_MST
1: col ← getThreadID
2: if Cy\textsubscript{a}[col] ≠ φ then
3: Mst\textsubscript{a}[Cy\textsubscript{a}[col]] ← false
4: end if

C. 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 mecha-
nism is similar to our SSSP step. Kernel4 (Algorithm 14)
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
for all active colors. This property is exploited while updating colors for each vertex. Each vertex in Kernel8 (Algorithm 18) finds its colorindex \( cid \) and traverses the colors array \( C_a \) until colorindex is not equal to color, converging at the lowest active color of its supervertex. The entire process is repeated until a single supervertex remains. A total of \( \sqrt{|V|} \) iterations are needed for the algorithm to terminate [38].

**E. Primitive based MST**

Another variation of the MST algorithm in a recursive framework using primitives such as scan, segmented-scan and split is developed from our group [39]. Though the algorithm reported in [39] is \( 2-3 \) times faster than the implementation stated above, it is heavy on memory requirements and cannot handle graphs larger than \( 6M \) and weights larger than \( 1K \) because of the \( 32-\)bit restriction of the segmented-scan operation and \( O(E) \) sized scan and split operations used in the implementation. Please see [39] for comparison of the above mentioned and recursive MST implementations on smaller sized graphs than reported here.

**IX. MAXIMUM FLOW (MF)/MIN CUT**

Maxflow tries to find the minimum weighed cut that separates a graph into two disjoint sets of vertices, containing the source \( s \) and the target \( t \) vertices. The fastest serial solution due to Goldberg and Rao takes \( O(E\min(V^{2/3}, \sqrt{E}) \log(V^2/E) \log(U)) \) time [40], where \( U \) is the maximum capacity of the graph.

Popular serial solutions to the max flow problem include Ford-Fulkerson’s algorithm [41], later improved by Edmond and Karp [42], and the Push-Relabel algorithm [43] 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, however, 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.


GPU implementations of the push-relabel algorithm are also reported [47]. A CUDA implementation for grid graphs specific to vision applications is reported in [48]. 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, \( E_f \). 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 \( E_f \) or saturation of vertex, i.e., empty reservoir.
- **Relabel**: Relabel operation is applied to change the heights. Any vertex having excess flow, which cannot flow due to height mismatch undergoes relabeling. The relabel operation ensures the height of the vertex to be one more than the minimum height of its neighbor.

Better estimates of height values can be obtained using global or gap relabeling [45]. 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.

**Algorithm 18 KERNEL8_MST**

```
1: \( tid \leftarrow \text{getThreadID} \)
2: \( cid \leftarrow C_i[tid] \)
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: \( C_i[tid] \leftarrow cid \)
9: if \( col \neq 0 \) then
10: \( \text{Terminate} \leftarrow \text{false} \)
11: end if
```

Fig. 10. Parallel maxflow, showing push and relabel operations based on \( e_a \) and \( h_a \)
**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 states per vertex, 0 corresponding to the relabeling state ($e_a(u) > 0, h_a(v) \geq h_a(u) \forall$ neighbors $v \in G_f$). $M_a$ is set to 1 for the push state ($e_a(u) > 0$ and $h_a(u) = h_a(v) + 1$ for any neighbor $v \in G_f$) else the mask is set as 2 for saturation (Table VI). Based on these values the push and relabel operations are undertaken. Initially activity mask is set to 0 for all vertices. A backward BFS from the sink node is used for global relabeling. Global relabeling is used heuristically in our implementation. We apply multiple pushes before applying the relabel operation. Multiple local relabels are applied before applying a single global relabel step (Algorithm 19).

**Algorithm 21 KERNEL2_MAXFLOW**

1. $tid \leftarrow$ getThreadID
2. if $M_a[tid] = 1$ then
3. for all neighbors nid of $tid$ do
4. if nid $\in G_f$ and $h_a[nid] = h_a[u] + 1$ 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 22 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.

**Algorithm 22 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

The operations terminates when there is no change in the activity mask. This does not necessarily occur when all nodes have saturated. Due to saturation of edges, unsaturated nodes may get cutoff from sink. Such nodes are not actively taking part in the process, consequently their state does not change. Termination hence cannot be based on saturation of nodes. Results of this implementation are given in Figure 11(g). Figure 11(h) and Figure 11(i) shows the behavior of our implementation with varying $m$ and $k$ in accordance to Algorithm 19.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_a$</td>
<td>Holds activity state per vertex</td>
</tr>
<tr>
<td>$e_a$</td>
<td>Holds excess flow per vertex</td>
</tr>
<tr>
<td>$h_a$</td>
<td>Holds height at every vertex</td>
</tr>
<tr>
<td>$W_a$</td>
<td>Holds residual capacity for every edge</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE VI</th>
<th>VARIABLES AND THEIR USE IN MAX FLOW IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_a$</td>
<td>Holds activity state per vertex</td>
</tr>
<tr>
<td>$e_a$</td>
<td>Holds excess flow per vertex</td>
</tr>
<tr>
<td>$h_a$</td>
<td>Holds height at every vertex</td>
</tr>
<tr>
<td>$W_a$</td>
<td>Holds residual capacity for every edge</td>
</tr>
</tbody>
</table>

**Relabel:** Relabels are applied as given in Algorithm 19. 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. Kernel 1 (Algorithm 20) explains this operation. Global relabeling uses backward BFS from sink, which propagates the height values to each vertex in the residual graph based on its actual distance from sink.

**Algorithm 20 KERNEL1_MAXFLOW**

1. $tid \leftarrow$ getThreadID
2. if $M_a[tid] = 0$ then
3. for all neighbors nid of $tid$ do
4. if nid $\in G_f$ and $minh > h_a[nid]$ 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:** 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 $(v, u)$. Kernel 2 (Algorithm 21) performs the push operation.
X. Performance Analysis

We choose graphs representatives of real world problems. Our graph sizes vary from $1M$ to $10M$ vertices for all algorithms except APSP. Scarpazza et al. [15] focus on improving the throughput of the Cell/B.E. for BFS. Bader and Madduri [3], [4], [35] use CRAY MTA−2 for BFS, STCON and MST implementations. Dehne and Götz [37] use CC−48 to perform MST. Edmonds et al. [49] use Parallel Boost graph library and Crobak et al. [50] use CRAY MTA−2 for their SSSP implementations. Yoo et al. [5] 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 [7] and Buluc et al. [10] on similar graph sizes as the implementations are directly comparable.

The focus of the performance analysis is on what GPUs can deliver on the seemingly irregular problems involving graphs. The low costs of the GPUs make them highly available to a wide audience and are good candidates for accelerating different types of tasks. We compare the GPU performance with other GPU implementations when available. We also compare the performance on a standard CPU using standard implementations as an indication of the practical acceleration that the GPU can provide. To this end, we show performance of all our implementations on a high-end GPU. We also show the performance on low-end and medium-end GPUs on feasible graph sizes. Comparison with the CPU is not otherwise meaningful as the two devices are radically different.

A. Types of Graphs

We tested our algorithms on various types of synthetic and real world large graphs including graphs from the ninth DIMACS challenge [51]. Primarily, three generative models were used for performance analysis, using the Georgia Tech. graph generators [52].

- 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 [53]/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, Erdős-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 [54]: These graphs are made up of random sized cliques of vertices with a hierarchical distribution of edges between cliques based on a distance metric.

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.

B. Experimental Setup

Our testbed consisted of a single Nvidia GTX 280 graphics adapter with 1024MB memory controlled by a Quad Core Intel processor (Q6600 @ 2.4GHz) with 4GB RAM running Fedora Core 9. For CPU comparison we use the C++ Boost graph library (BGL), with the exception of BFS, compiled using gcc at optimization setting −O4 on the Intel Quad Core Q6600, 2.4GHz processor. We use our own BFS implementation on GPU 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 a similar GPU as GTX 280 with 4096MB of memory clocked at a slightly lower frequency. We also show scalability of our algorithms on low end GPUs including 8600GT (32 stream processors and 256MB RAM) and 8800GT (112 stream processors and 512MB RAM).

We are aware there may exist more optimized implementations of algorithms reported than BGL. Our aim to to show data parallel approaches presented to be applicable on readily available hardware, with better scalability and performance than CPU. Detailed timings of the plots given are listed in Table X and Table XI in the Appendix.

C. Iterative Mask Based Approach

We implement BFS, STCON, SSSP and Maxflow using the iterative mask based approach. A speedup of nearly 15 − 20 times over BGL is observed in these implementations on Random and SSCA#2 graphs as shown in Figures 11(a), 11(b), 11(c) and 11(g). Data parallelism is exploited in 80 − 90% of the total time taken for execution. For Random and SSCA#2 graphs 7 − 8% time is taken to reach full parallelism (number of threads ≥ number of processors), while R-MAT graphs take nearly 20% of the total time. Low degree and linear graphs exhibit lower performance for the iterative mask based implementations empirically, as seen in Table IX. This behavior is not surprising, since this approach cannot exploit parallelism under such a scenario. A maximum of two vertices can be processed in parallel in each iteration for a linear graph using the iterative mask based approach. Vertex list compaction, however, helps reduce running time by 25 − 40% in such cases. Large variation in degree also 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 (Figures 11(a), 11(b), 11(c) and 11(g)).

BFS (Figure 11(a)), STCON (Figure 11(b)) and SSSP (Figure 11(c)) show similar behavior on all graph models. We use vertex list compaction in BFS and SSSP leading to 40% reduction in time in case of R-MAT graphs. Running times for 100 iterations of randomly selected $s$ and $t$ are reported for STCON. The implementations are highly scalable and exhibit an almost linear respone in timings as the size of the graph is increased. R-MAT graphs, even after compaction,
Fig. 11. Experiments on varying sizes and varying degree for three types of graph models.

(a) Breadth first search on varying number of vertices for synthetic graphs.
(b) st-Connectivity for varying number of vertices for synthetic graph models.
(c) Single source shortest path on varying number of vertices for synthetic graph models.


Maxflow timings for various graphs are shown in Figure 11(g). We average timings over 5 iterations for randomly selected source s and sink t vertices. R-MAT GPU times out shoots the CPU times because of their low degree nature and slow convergence of local relabel thereof. The behavior of our max flow implementation for varying m and k, controlling the periodicity of the local and global relabeling steps (Algorithm 19), is given in Figure 11(h) and Figure 11(i).
for 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 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 further reinforces the fact that low degree poses slow convergence of local relabels.

A speed up of nearly \( 5 – 7 \) times in case of R-MAT graphs and \( 15 – 20 \) times in case of Random and SSCA#2 graphs is observed over BGL for these implementations. Larger degree graphs benefit more using iterative mask based implementations as the expansion per iteration is more, resulting in better expansion of data and thus better performance.

### D. The Divide-and-Conquer Approach

The MST algorithm is implemented using a divide-and-conquer approach. Timings for minimum spanning tree implementation are summarized in Figure 11(f) for synthetic graphs. The divide-and-conquer approach is not affected by linearity of the graph, as each supervertex is processed in parallel independent to other supervertices and there is no frontier expansion. However, for R-MAT graphs we see a slowdown due to uneven loops over vertices with high degree, which prove inefficient on an SIMD model.

### E. All Pairs Shortest Paths (APSP)

The SSSP, matrix multiplication and Gaussian elimination APSP implementations are compared in Figure 11(d) on a GTX 280 and Tesla. The SSSP based solution uses iterative mask based approach where as Gaussian Elimination based approach due to Buluc et al. [10] is recursive divide-and-conquer. The matrix multiplication based APSP uses graph representation outlined in Section VII. 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 has similar compute power to that of GTX280, but with 4 times the memory. The Gaussian Elimination based APSP by Buluc et al. [10] 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 11(d) and Table XI. For direct comparison with Katz and Kider [7], we also show results on Quadro FX 5600. Figure 11(e) 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.

### F. Scalability

Behavior of our implementations with varying degrees are summarized in Table VII. We show results for a \( 100K \) vertex graph with varying degree. For APSP matrix based approach results for a \( 4K \) graph are shown. The running time increase with increasing degree in all cases, however, GPU implementations scale better than their CPU counterparts for all implementations. A sub-linear reduction in time is observed for GPU in contrast to a linear slow down on the CPU. The behavior can be explained based on the work done in these implementations, that is distributed over parallel threads resulting in a lower increase in time as compared to CPU. Table VIII shows results for the BFS, SSSP and MST implementations on low end graphics processors, the 8600GT with 32 stream processors and 256MB RAM and the 8800GT with 112 stream processors and 512MB of RAM. Inferring from experiments we can see both iterative mask and divide-and-conquer approaches scale linearly with the number of stream processors on the CUDA device. We see a speedup of \( 2 – 3 \) times on 8600GT and \( 7 – 8 \) times for 8800GT over CPU for these approaches, which are entry level CUDA devices. Nvidia has integrated GPUs on motherboards which support CUDA processing, the 9400 series motherboards come with a low end GPU. Performance over CPU can be gained by porting algorithms to CUDA on such devices.

Results on the ninth DIMACS challenge [51] dataset are summarized in Table IX. GPU performs worse than CPU in most implementations for these inputs expect in case of minimum spanning tree. The divide and conquer approach is inert to linearity of the graph and thus performs better than the iterative mask based implementations. Average degree is \( 2 – 3 \) makes these graphs almost linear and expand data minimally in each iteration the frontier based implementations.

### XI. Conclusions

In this paper, we presented massively multithreaded algorithms on large graphs for the GPU using the CUDA model. Each operation was typically broken down using a BSP-like model into several kernels executing on the GPU, synchronized by the CPU. We showed results on random graphs and scale-free graphs that are inspired by real-life problems. The high performance demonstrated makes the GPUs attractive as co-processors to the CPU for several scientific and engineering tasks that are modeled as graph algorithms. The wide availability of the GPUs puts it within the reach of every user who needs high performance computing. Practical implementations that deliver superior performance will ease the adoption of GPUs by a wide range of users. The source code for our implementations will be made available on the web. This is likely to facilitate their adoption by other users, going by our experience on BFS and SSSP. In addition to the higher performance, we believe the approaches presented will be applicable to the multicore and manycore architectures that are in the pipeline from different manufacturers for graph algorithms.
### TABLE VII
Scalability with varying degree on a 100K vertex graph, 4K for APSP, weights in range 1 – 100. Times in milliseconds.

<table>
<thead>
<tr>
<th>Degree</th>
<th>BFS GPU/CPU</th>
<th>STCON GPU/CPU</th>
<th>SSSP GPU/CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Random</td>
<td>R-MAT</td>
<td>SSCA#2</td>
</tr>
<tr>
<td>100</td>
<td>15/420</td>
<td>917/280</td>
<td>7/160</td>
</tr>
<tr>
<td>200</td>
<td>48/800</td>
<td>122/460</td>
<td>13/290</td>
</tr>
<tr>
<td>400</td>
<td>125/1520</td>
<td>163/770</td>
<td>24/510</td>
</tr>
<tr>
<td>600</td>
<td>177/2300</td>
<td>182/1050</td>
<td>38/730</td>
</tr>
<tr>
<td>800</td>
<td>253/3060</td>
<td>210/1280</td>
<td>67/980</td>
</tr>
<tr>
<td>1000</td>
<td>364/-</td>
<td>6.8/-</td>
<td>-/-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Degree</th>
<th>MST GPU/CPU</th>
<th>Max Flow GPU/CPU</th>
<th>APSP Matrix GPU/CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Random</td>
<td>R-MAT</td>
<td>SSCA#</td>
</tr>
<tr>
<td>100</td>
<td>302/12150</td>
<td>461/10290</td>
<td>122/7470</td>
</tr>
<tr>
<td>200</td>
<td>369/25960</td>
<td>638/22180</td>
<td>1218/16700</td>
</tr>
<tr>
<td>400</td>
<td>1149/-</td>
<td>849/-</td>
<td>347/-</td>
</tr>
<tr>
<td>600</td>
<td>1908/-</td>
<td>1103/-</td>
<td>499/-</td>
</tr>
<tr>
<td>800</td>
<td>2484/-</td>
<td>1178/-</td>
<td>883/-</td>
</tr>
<tr>
<td>1000</td>
<td>3338/-</td>
<td>-/-</td>
<td>-/-</td>
</tr>
</tbody>
</table>

### TABLE VIII
Scalability of BFS, SSSP and MST on lower end GPUs. Times in milliseconds. Average Degree 12.

<table>
<thead>
<tr>
<th>Number of vertices</th>
<th>Time 8600GT/8800GT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BFS</td>
</tr>
<tr>
<td></td>
<td>Random</td>
</tr>
<tr>
<td></td>
<td>161/68</td>
</tr>
<tr>
<td>1M</td>
<td>1733/919</td>
</tr>
<tr>
<td>2M</td>
<td>-1/2016</td>
</tr>
<tr>
<td>3M</td>
<td>-2/3880</td>
</tr>
<tr>
<td>4M</td>
<td>-3/3900</td>
</tr>
<tr>
<td>5M</td>
<td>-/4029</td>
</tr>
</tbody>
</table>

### TABLE IX
Results on the Ninth DIMACS Challenge [51] graphs, weights in range 1 – 300K. Times in milliseconds.

<table>
<thead>
<tr>
<th>Graphs with distances as weights</th>
<th>Vertices</th>
<th>Edges</th>
<th>BFS</th>
<th>STCON</th>
<th>Time GPU/CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York</td>
<td>264346</td>
<td>733846</td>
<td>147/20</td>
<td>1.25/8.8</td>
<td>448/190</td>
</tr>
<tr>
<td>San Francisco Bay</td>
<td>321270</td>
<td>800172</td>
<td>199/20</td>
<td>2.2/11.3</td>
<td>623/230</td>
</tr>
<tr>
<td>Colorado</td>
<td>435666</td>
<td>1057066</td>
<td>414/30</td>
<td>2.36/15.9</td>
<td>1738/340</td>
</tr>
<tr>
<td>Florida</td>
<td>1070376</td>
<td>2712798</td>
<td>1241/80</td>
<td>5.02/37.7</td>
<td>4805/810</td>
</tr>
<tr>
<td>Northwest USA</td>
<td>1207945</td>
<td>2840208</td>
<td>1588/100</td>
<td>7.8/48.3</td>
<td>8071/1030</td>
</tr>
<tr>
<td>Northeast USA</td>
<td>1524453</td>
<td>3897636</td>
<td>2077/140</td>
<td>8.6/66.5</td>
<td>8563/1560</td>
</tr>
<tr>
<td>California and Nevada</td>
<td>1890815</td>
<td>4657742</td>
<td>2762/180</td>
<td>9.4/100</td>
<td>11664/1770</td>
</tr>
<tr>
<td>Great Lakes</td>
<td>2758119</td>
<td>6885658</td>
<td>5704/240</td>
<td>19.8/114.7</td>
<td>32905/2730</td>
</tr>
<tr>
<td>Eastern USA</td>
<td>3598623</td>
<td>8778114</td>
<td>7666/400</td>
<td>24.4/183.8</td>
<td>41315/4140</td>
</tr>
<tr>
<td>Western USA</td>
<td>6262104</td>
<td>15248146</td>
<td>14065/800</td>
<td>58/379.8</td>
<td>82247/850</td>
</tr>
<tr>
<td>Central USA</td>
<td>14081816</td>
<td>34292496</td>
<td>37936/3580</td>
<td>200/1691</td>
<td>215087/34560</td>
</tr>
<tr>
<td>Full USA</td>
<td>23947347</td>
<td>58333344</td>
<td>102302/</td>
<td>860/-</td>
<td>672542/-</td>
</tr>
</tbody>
</table>

1Results taken on Tesla | 5Max Flow results at $m = 3$ and $k = 7$

### ACKNOWLEDGMENT

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.
### APPENDIX

### TABLE X
**SUMMARY OF RESULTS FOR SYNTHETIC GRAPHS. TIMES IN MILLISECONDS**

<table>
<thead>
<tr>
<th>Algo</th>
<th>Graph Type</th>
<th>Number of Vertices, average degree 12, weights varying from 1 to 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS*</td>
<td>Random GPU†</td>
<td>38 82 132 184 251 338 416 541 635† 678†</td>
</tr>
<tr>
<td></td>
<td>Random CPU</td>
<td>530 1230 2000 2710 3480 4290 5040 5800 - -</td>
</tr>
<tr>
<td></td>
<td>R-MAT GPU†</td>
<td>244 433 778 944 1429 1526 1969 2194 2339† 3349†</td>
</tr>
<tr>
<td></td>
<td>R-MAT CPU</td>
<td>340 760 1230 1680 2270 2760 3220 3620 - -</td>
</tr>
<tr>
<td></td>
<td>SSCA# GPU†</td>
<td>30 62 95 142 178 233 294 360 433† 564†</td>
</tr>
<tr>
<td></td>
<td>SSCA# CPU</td>
<td>420 930 1460 2010 2550 3150 3710 4310 - -</td>
</tr>
<tr>
<td>STCON</td>
<td>Random GPU†</td>
<td>1.42 3.06 4.28 5.34 6.62 7.37 9.36 10.8 11.15 -</td>
</tr>
<tr>
<td></td>
<td>Random CPU</td>
<td>68 164 286 310 416 536 692 - - -</td>
</tr>
<tr>
<td></td>
<td>R-MAT GPU†</td>
<td>19.2 32.37 172.1 347.4 408.3 579.1 626 1029 - -</td>
</tr>
<tr>
<td></td>
<td>R-MAT CPU</td>
<td>160 358 501 638 926 1055 1288 - - -</td>
</tr>
<tr>
<td></td>
<td>SSCA# GPU†</td>
<td>1.96 3.76 5.33 5.44 7.23 8.08 9.1 12.33 - -</td>
</tr>
<tr>
<td></td>
<td>SSCA# CPU</td>
<td>78 176 286 422 552 595 665 - - -</td>
</tr>
<tr>
<td>SSSP</td>
<td>Random GPU†</td>
<td>116 247 393 547 698 920 947 1140 1247 1355</td>
</tr>
<tr>
<td></td>
<td>Random CPU</td>
<td>2330 5430 10420 18130 - - - - - -</td>
</tr>
<tr>
<td></td>
<td>R-MAT GPU†</td>
<td>576 1025 1584 1842 2561 3575 11334 - - -</td>
</tr>
<tr>
<td></td>
<td>R-MAT CPU</td>
<td>1950 4200 6700 11680 - - - - - -</td>
</tr>
<tr>
<td></td>
<td>SSCA# GPU†</td>
<td>145 295 488 632 701 980 1187 1282 1583 2198†</td>
</tr>
<tr>
<td></td>
<td>SSCA# CPU</td>
<td>2110 4490 6970 9550 - - - - - -</td>
</tr>
<tr>
<td>MST</td>
<td>Random GPU†</td>
<td>770 1526 2452 3498 4654 6424† 8670† 11125† - -</td>
</tr>
<tr>
<td></td>
<td>Random CPU</td>
<td>12160 26040 - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>R-MAT GPU†</td>
<td>2076 4391 5995 9102 10875 12852 15619† 21278† - -</td>
</tr>
<tr>
<td></td>
<td>R-MAT CPU</td>
<td>551 1174 1772 2970 4173 4879 7806 † 9993† - -</td>
</tr>
<tr>
<td></td>
<td>SSCA# GPU†</td>
<td>7540 15980 25230 - - - - - -</td>
</tr>
<tr>
<td></td>
<td>SSCA# CPU</td>
<td>598 3013 5083 7197 7323† 16871† 30201† 34253† - -</td>
</tr>
<tr>
<td>MF</td>
<td>Random GPU†</td>
<td>15390 33290 - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>Random CPU</td>
<td>30743 55514 74767 148627 232789† 311267† - - -</td>
</tr>
<tr>
<td></td>
<td>R-MAT GPU†</td>
<td>8560 18770 - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>R-MAT CPU</td>
<td>459 2548 2943 7388 8606† 12742† - - -</td>
</tr>
<tr>
<td></td>
<td>SSCA# GPU†</td>
<td>9760 20960 - - - - - - -</td>
</tr>
<tr>
<td></td>
<td>SSCA# CPU</td>
<td>2.18 6 19.64 96.5 639 3959 - - - -</td>
</tr>
</tbody>
</table>

### TABLE XI
**SUMMARY OF RESULTS FOR SYNTHETIC GRAPHS APSP APPROACHES. TIMES IN MILLISECONDS**

<table>
<thead>
<tr>
<th>APSP</th>
<th>Graph Type</th>
<th>Number of Vertices, average degree 12, weights in range 1 – 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using</td>
<td>Random</td>
<td>499 1277 3239 7851 18420 56713 65375 72625 166080 316078 556313</td>
</tr>
<tr>
<td>SSSP</td>
<td>R-MAT</td>
<td>489 1531 4145 12442 38812 143991 170121 211277 465037 1028275 1362119</td>
</tr>
<tr>
<td>GTX 280</td>
<td>SSCA#2</td>
<td>469 1300 3893 7677 17450 50498 58980 67794 163081 353166 461901</td>
</tr>
<tr>
<td>Matrix</td>
<td>Random</td>
<td>2.77 11.3 55.7 330.4 2240.8 41150 58889 72881† 242264† 1724970† 3072443†</td>
</tr>
<tr>
<td>GTX 280</td>
<td>R-MAT</td>
<td>2.54 10.9 66.2 478 3756 32263 56906 71025† 339188† 1152989† 4032675†</td>
</tr>
<tr>
<td></td>
<td>SSCA#2</td>
<td>2.55 8.6 42.9 263.6 2063.7 43045 62517 76399† 220868† 1360469† 1872394†</td>
</tr>
<tr>
<td></td>
<td>Fully Conn.</td>
<td>2.9 15.2 112 959 8363 110658 151820 202118† 799035† 4467455† -</td>
</tr>
<tr>
<td>Matrix</td>
<td>Random</td>
<td>3.25 21.3 136 827.9 3548 65552 87043 113993 1048598 - -</td>
</tr>
<tr>
<td></td>
<td>R-MAT</td>
<td>2.99 22 174.8 1307.6 10534 94373 115294 137854 1487025 - -</td>
</tr>
<tr>
<td></td>
<td>SSCA#2</td>
<td>2.91 15.78 103.2 635.3 4751 62555 82708 109744 1001212 - -</td>
</tr>
<tr>
<td></td>
<td>Fully Conn.</td>
<td>2.9 25.1 221.5 1941 16904 268757 368397 490157 4300447 - -</td>
</tr>
<tr>
<td>FX 5600</td>
<td>Random</td>
<td>1.8 4.4 12.1 44.9 230 1505 - - - -</td>
</tr>
<tr>
<td></td>
<td>R-MAT</td>
<td>1.8 4.5 12.1 45 230 1505 - - - -</td>
</tr>
<tr>
<td></td>
<td>SSCA#2</td>
<td>1.8 4.5 12.1 44 230 1497 - - - -</td>
</tr>
<tr>
<td>GE Based</td>
<td>Random</td>
<td>2.18 6 19.64 96.5 639 3959 - - - -</td>
</tr>
<tr>
<td></td>
<td>R-MAT</td>
<td>1.86 5.9 19.1 96 638 3959 - - - -</td>
</tr>
<tr>
<td></td>
<td>SSCA#2</td>
<td>2.18 5.9 19.67 96 638 3965 - - - -</td>
</tr>
<tr>
<td>Katz†</td>
<td>-</td>
<td>7.7 34.9 230.1 1735.6 13720 158690 216400 1015700 - -</td>
</tr>
</tbody>
</table>

*CPU implementation is ours
†Using Compaction process
‡Results taken on a Tesla S1070
§Max Flow results at m = 3 and k = 7
¶Results using streaming from CPU to GPU memory


