Scalable and Performant Graph Processing on GPUs Using Approximate Computing

Somesh Singh and Rupesh Nasre

Abstract—Graph algorithms are being widely used in several application domains. It has been established that parallelizing graph algorithms is challenging. The parallelization issues get exacerbated when graphics processing units (GPUs) are used to execute graph algorithms. While the prior art has shown effective parallelization of several graph algorithms on GPUs, a few algorithms are still expensive. In this work, we address the scalability issues in graph parallelization. In particular, we aim to improve the execution time by tolerating a little approximation in the computation. We study the effects of four heuristic approximations on six graph algorithms with five graphs and show that if an application allows for small inaccuracy, this can be leveraged to achieve considerable performance benefits. We also study the effects of the approximations on GPU-based processing and provide interesting takeaways.

Index Terms—Graph algorithms, irregular programs, GPU, CUDA, approximate computing

1 Introduction

Graphs are fundamental data structures for modeling several real-world phenomena. Graph algorithms are useful, for instance, to simulate molecular interactions, to perform program analysis, as well as to optimize sensor placement. As data sizes grow, handling graphs at larger scale poses performance challenges. Parallel computation of graph algorithms has been one of the ways to improve the execution time of applications. Several popular graph algorithms have now been successfully parallelized across various platforms such as multi-core CPUs, distributed systems, many-core GPUs, and their combinations [1], [2], [3], [4], [5], [6]. When powerful machines are not at disposal, a practical question is, given an infrastructure, how to make the best use of resources to come up with a reasonable inexact solution.

An exact answer is required in some applications. However, one may not *always* be interested in the accurate answer. In this work, we target adding controlled approximations to graph algorithms to improve their efficiency. Thus, instead of computing the exact answer, algorithms perform less work to calculate only an approximate solution. Towards this goal, we propose several graph-theoretic but algorithm-independent heuristics. These heuristics target various parts of computation and data. While many such heuristics are feasible (Fig. 1), we study two techniques directed towards computation and two techniques directed towards data. Our investigation reveals that each of these techniques is quite beneficial in improving the execution time, at the cost of accuracy. Interestingly, a user can control the exhibited inaccuracy by controlling the injected approximation. We believe that our proposed approximations would be helpful for other graph algorithms as well.

This paper makes the following contributions.

- We devise a theoretical model of approximation and illustrate its generality by instantiating it with different kinds of approximations (Section 2).
- We propose techniques for executing graph algorithms on GPUs in an approximate manner. In particular, our techniques perform reduced execution, process only part of the graph, store graph in an approximate manner, and approximate attribute values to gain in efficiency (Section 3). We also discuss how approximations can be exploited for an efficient GPU-based parallel processing (Section 4).
- We show that our proposed techniques work quite well in practice compared to the exact versions. Parameterized solutions provide tunable knobs to change the degree of approximation. Using five large graphs and six graph algorithms, we illustrate that approximate versions offer considerable performance benefits with a small accuracy-loss (Section 5).

2 Approximation Model

We define a theoretical model to characterize approximations. By instantiating this model with various parameters, we obtain different approximation techniques.

An approximation $\mathcal{A}$ ($\mathcal{D}$, $\mathcal{F}$) is defined over domain $\mathcal{D}$ of entities, where function $\mathcal{F}$: entity $\rightarrow$ entity is used to approximate the entities. For instance, $\mathcal{A}$ (AttrVal, Div1024) where Div1024(attrval) :- (attrval / 1024) approximates attribute values by making consecutive 1,024 values non-distinguishable. In other words, $\mathcal{F}$ *maps* entities in $\mathcal{D}$ to a subset of entities in $\mathcal{D}$. Thus, $\mathcal{F}$, in general, is a many-to-one function, which provides the necessary approximation.

The mapping function $\mathcal{F}$ can be arbitrary, ranging from identity function (indicating no approximation) to constant

• *The authors are with IIT Madras, Chennai, Tamil Nadu 600036, India. E-mail: {ssomesh, rupesh}@cse.iitm.ac.in.*

Manuscript received 31 Mar. 2017; revised 10 Jan. 2018; accepted 12 Jan. 2018. Date of publication 23 Jan. 2018; date of current version 14 Sept. 2018. (Corresponding author: Somesh Singh.)
Recommended for acceptance by A. Kalyanaraman and M. Halappanavar.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TMSCS.2018.2795543

Domain $\mathcal{D}$	Mapping function $\mathcal{F}$	Examples
Iterations	iteration $> K \implies$ *void* truncate iterations based on an error threshold ...	reduced execution (Section 3.2) more approximate computation ...
Graph processing	process partial graph process top K graph elements conditional processing ...	truncate computation to K vertices or edges (Section 3.3) sort based on a criterion such as degree and select top K vertices process only those graph elements that satisfy a condition ...
Graph representation	lossy graph compression maximum degree K clustering reduced storage ...	merge vertices based on neighborhood similarity (Section 3.4) truncate graph beyond degree K per vertex merge nearby vertices store graph in a smaller adjacency matrix / lists ...
Attribute values (both vertex and edge)	integer division by K round-off to the nearest power of 2 modulus by K hashing ...	consecutive K entries similar mapped to powers of 2 (Section 3.5) round-robin mapping from 0 to K-1 similarity based on the hash function ...
...	...	...

Fig. 1. Instantiation of the approximation model with various values of $\mathcal{D}$ and $\mathcal{F}$.

function (indicating maximum approximation). A judicious selection of $\mathcal{F}$ can help improve algorithmic performance (both time and space) without losing much precision.

2.1 Function Application Order

The mapping function $\mathcal{F}$ is type-preserving, that is, it maps an entity in the domain to another entity in the same domain. Therefore, $\mathcal{F}$ application can be cascaded: $\mathcal{F}.\mathcal{F}.\mathcal{F}.x$, leading to a chain of approximations. However, as long as $\mathcal{F}$ is deterministic, the ordering in which various function applications are performed does not affect the outcome. For instance, computing minimum among a set of values does not depend upon the order. Such a property is crucial in a parallel setting where various thread orderings lead to the same approximate entity for the original domain entity. On the other hand, a non-deterministic $\mathcal{F}$ may affect the outcome for a different thread-schedule; e.g., merging nodes based on their neighborhood similarity. We experiment with both kinds of approximation functions in this study.

2.2 Idempotent Approximation

Non-determinism in thread-scheduling may result in a different number of approximate function applications. This non-determinism may, in general, lead to different amounts of approximations added to the processing across different executions of the same program. Such behavior may be unacceptable as it means different outputs across different runs of the same program over the same input. Need for such a determinism necessitates $\mathcal{F}$ to be an idempotent function. Thus, multiple applications of $\mathcal{F}$ should be equivalent to a single application. Although expecting the mapping function to be idempotent may sound restrictive, in practice, most of the standard approximation techniques are indeed idempotent. For instance, mapping an edge weight (say 23) to the nearest power of two (32) is an idempotent approximation, as a remapping (on 32) would maintain the value (as 32). All the approximation techniques we propose are idempotent. This allows us to faithfully assess the effect of approximations compared to the exact processing.

2.3 Approximation Structure

We define a relation $\mathcal{R}$ between a pair of entities induced by the approximation function $\mathcal{F}$. Thus, $x_1 \mathcal{R} x_2$ iff $\mathcal{F}(x_1) = \mathcal{F}(x_2)$. $\mathcal{R}$ is a reflexive ($x \mathcal{R} x$), symmetric ($x \mathcal{R} y \Rightarrow y \mathcal{R} x$) and transitive ($x \mathcal{R} y$ and $y \mathcal{R} z \Rightarrow x \mathcal{R} z$), forming an equivalence relation. Thus, $\mathcal{R}$ partitions the domain $\mathcal{D}$.

At one extreme, when the approximation function is an identity function, each element in the domain is in a separate partition, say with cardinality N. At the other extreme, when the approximation function is a constant function, all the elements are in the same partition with cardinality 1. In general, various approximation functions form K partitions with $1 \leq K \leq N$, leading to different precision values.

3 Approximating Graph Algorithms

We instantiate the approximation model with various values of $\mathcal{D}$, and accordingly, multiple values of $\mathcal{F}$. Fig. 1 presents such approximations. The number of instantiations can be huge; we pick one interesting approximation technique for four domains and explore it in depth.

3.1 Graph Algorithms

We work with a variety of algorithms: Single Source Shortest Paths (SSSP), Minimum Spanning Tree (MST), Betweenness Centrality (BC) PageRank (PR), Strongly Connected Components (SCC), and Vertex Coloring (Color). The input to each graph algorithm is a directed graph. The graph-edges have weights in case of SSSP, MST and BC.

SSSP [7] computation finds the shortest distance of each vertex from a designated source in a weighted graph. We implement a variation of Bellman-Ford's algorithm which is more amenable to parallelization compared to the work-efficient Dijkstra's algorithm. MST [8] computation finds a tree in a given graph having the minimum sum of the tree's edge-weights and which spans all the vertices of a connected graph. We use Boruvka's algorithm which offers better parallelism over Prim's or Kruskal's algorithms. Finding SCC [9] is another fundamental problem which identifies cycles in a given graph. We use Forward-Backward algorithm for SCC which offers better parallelism over a depth-first search based processing. Coloring [10] is a well-known NP-complete problem. We use a greedy approximation algorithm for $\delta + 1$ coloring (where δ is the maximum degree in the graph). PR [11] is a propagation-based algorithm to compute page rank values (related to importance) of vertices in a web-graph. BC [12] computes importance of a vertex in a graph, and deals with identifying the fraction

of the shortest paths passing through each vertex. We use Brandes' algorithm on unweighted graphs which is $\mathcal{O}(mn)$, where m and n are the number of edges and the number of vertices of the graph respectively.

Algorithm 1. SSSP Computation Over Graph $G(V, E)$

```
1: v.dist = ∞      ∀v ∈ V                          ▷ initialization
2: source.dist = 0
3: changed = true
4: while changed do                                ▷ outer loop
5:    changed = false;
6:    for all vertices u in worklist do            ▷ GPU parallel
7:       for all outgoing edges u → v do
8:          altdist = u.dist + weight(u → v)
9:          if altdist < v.dist then
10:            v.dist = altdist                    ▷ needs synchronization
11:            changed = true;
12:         end if
13:      end for
14:   end for
15: end while
```

Algorithms 1 and 2 present the pseudo-codes for SSSP and BC respectively. We expose parallelism in BC by processing a single source vertex at a time and performing each of the inner loops (on lines 11, 23, 29) in parallel. When we visit the neighbors of a vertex, edges can be relaxed concurrently. $\delta_s(v)$ on Line 25 is computed as

$$\delta_s(v) = \sum_{w \,|\, v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_s(w)). \tag{1}$$

Equation (1) calculates the *dependency* of a vertex v with respect to a given source vertex s. Here, σ_{sv} is the number of shortest paths from s to v, and $pred(s, w)$ is a list of immediate predecessors of w in the shortest paths from s to w. The size of the $pred$ list of a vertex is bounded by the indegree of the vertex. $pred$ lists of all the vertices together form a DAG D which is a subgraph of G. BC values for each vertex can then be computed as below:

$$bc(v) = \sum_{s \neq v \in V} \delta_s(v). \tag{2}$$

3.2 Technique 1: Reduced Execution

In the reduced execution technique, we cut-short the execution to compute an approximate solution. Graph algorithms are often iterative. We exploit this fact to add approximation to the total amount of work done in terms of the number of iterations. That is, we execute the main processing loop (outermost if there are nested loops) for fewer iterations (compared to the corresponding exact version) with the hope of improving performance. Less overall work forbids the algorithm from reaching the fixed-point or the correct solution. For instance, consider single-source shortest paths computation shown in Algorithm 1. The outer `while` loop at Line 4 is cut-short. Reduced execution approximation is useful for algorithms where a large amount of work gets done in the initial iterations. One way to implement this approximation is by configuring a percentage threshold on the number of loop iterations. This is feasible as long as the loop executes a fixed number of iterations (such as Prim's minimum spanning tree algorithm, Brandes' betweenness centrality computation, and Bellman-Ford shortest paths processing). In general, a more effective way is to provide an inaccuracy-tolerance, and the implementation chooses the maximum possible number of iterations respecting the inaccuracy limit. The inaccuracy-tolerance refers to the amount of inaccuracy permitted by an application, and varies with application. However, we also note that there may exist computations wherein inaccuracy cannot be calculated without computing the exact solution. Reduced execution can be applied in such scenarios too, but without any guarantees on the approximation.

Algorithm 2. BC Computation Over Graph $G(V, E)$

```
1: bc[v] = 0      ∀v ∈ V                           ▷ initialization
2: for each s ∈ V do
3:    marked[v] = false      ∀v ∈ V
4:    σ_s[v] = 0      ∀v ∈ V
5:    σ_s[s] = 1
6:    δ_s[v] = 0      ∀v ∈ V
7:    pred = {}                                    ▷ empty list
8:    Queue Q      ▷ queue for BFS from s; form BFS DAG D
9:    Q.push(s)
10:   marked[s] = true;
11:   while (not Q.empty()) do
12:      u = Q.front()
13:      Q.pop()
14:      for all v ∈ neighbors(u) do
15:         if not (marked[v]) then                 ▷ new node
16:            Q.push(v)
17:            marked[v] = true;
18:            σ_s[v] = σ_s[v] + σ_s[u]            ▷ needs synch.
19:            pred(s, v) = pred(s, v) ∪ {u}
20:         end if
21:      end for
22:   end while
23:   for all v ∈ D do              ▷ Backward traverse DAG D
24:      for each u ∈ pred(s, v) do
25:         δ_s[v] = δ_s[v] + δ_s[u]
26:      end for
27:      bc[v] += δ_s(v)
28:   end for
29:   for all (u → v) ∈ E do                       ▷ Reset graph attributes
30:      reset(u → v)
31:   end for
32: end for
```

Our experimental evaluation shows that a small decrease in the outer loop iterations achieves good benefits in execution time at the cost of small loss in accuracy. For example, we find that for SSSP, reducing the outer loop iterations to 90 percent achieves an average speedup of 1.6×, with an inaccuracy of up to 7 percent. However, the inaccuracy increases rapidly as we further reduce the number of iterations. For instance, reducing the loop iterations to 65 percent of the exact answer results in a performance gain of around 1.7× at the cost of 21 percent accuracy loss. We also found that SSSP is a good candidate for reduced execution as most of the distances get settled within about 50 percent of the iterations (Fig. 7). In contrast, Color exhibits a much higher

inaccuracy (29.18 percent) for a modest (45 percent) performance improvement.

3.3 Technique 2: Partial Graph Processing

Our next proposal is to process only part of the graph, to improve execution time. Not all parts of the graph contribute equally to the final fixed-point information. Ideally, we would like to process only the highest-contributing parts—to reduce execution time to the minimal, incurring minimal inaccuracy. However, such information is often not efficiently computable, and, in fact, changes across iterations. Therefore, we would need to depend upon heuristics to choose the part of the graph to be processed. Thus, based on criteria, for each pass through the graph, or for each iteration of the outermost loop, we choose to selectively process only a subset of the vertices/edges.

Partial graph processing resembles performing a random walk on the graph and is performed as below. For one pass through the graph, for each node we choose to process, we assign special values to its outgoing edges. The values are drawn uniformly at random from the set of non-negative integers $\in [0, m)$, where m is the number of edges. In other words, we generate one of the permutations of the edge identifiers. From among these values, we traverse only the highest few (say top 50 or 60 percent). The other edges are omitted, and the nodes on which the omitted edges are incident may not be processed in that iteration. An iteration is said to be complete when all the threads have completed their share of work. The work of each thread is to process the (selected) edges for the nodes assigned to it, once. We stop when the change in a parameter (dependent on the algorithm) across two successive iterations is *small*.

For instance, in SSSP as shown in Fig. 1, partial graph processing would change the for loop at Line 6 to go over a subset of vertices. We may fix the same edge-permutation for every iteration. But it leads to high deviation from the exact output. Therefore, we propose generating a new permutation of edge identifiers in every iteration. We also experimented with selectively skipping the same outgoing edges for the vertices we process. Such a scheme considerably reduces precision. Note that our method does not even traverse the edges not selected. In some graph algorithms (such as finding the vertex with the maximum-degree) where the processing loop enumerates through vertices or edges, the two approximations, namely, reduced execution and partial graph processing may overlap.

Partial processing allows us to reduce the total number of graph operations compared to the exact version. This also reduces the amount of synchronization required in processing the graph. For instance, in SSSP computation, the number of atomics reduces due to fewer vertices being processed. Since atomics on GPUs are costlier than regular reads and writes, this leads to better execution time.

Our experiments show that if we process only a fraction of the graph without modifying the edge/vertex attributes, the inaccuracy grows fairly quickly as we reduce the fraction of the graph processed. For instance, in SSSP, if we process 75 percent of the edges, it achieves around 1.2× speedup, with 21 percent inaccuracy. However, when we process only 25 percent of the edges, it achieves a speedup of around 2.8× but the inaccuracy shoots-up to 63 percent. On the other hand, if we assign values edge/vertex attributes carefully, then the error grows gradually even with the processing of a small fraction of the original graph. In case of SSSP, when we assign edge weights after preprocessing the graph before applying this approximation, we observe that processing 75 percent of the graph causes the answer to deviate from the exact value by 18 percent on an average. Also, with 25 percent of the graph processed, the inaccuracy is close to 24 percent and does not increase drastically.

3.4 Technique 3: Approximate Graph Representation

Reduced iteration and partial graph processing discussed in the last sections work with the original (exact) graph. In the approximate graph representation technique, the graph itself is stored in an imprecise manner. Thus, instead of working on the exact graph representation, the (exact) algorithm runs on graph's approximation. There are multiple ways to implement this. One way is to assign the same vertex-id to multiple vertices. Alternatively, we can store the graph in a probabilistic data structure (such as bloom filters). In this work, we explore vertex-merging, which involves logically merging the adjacency lists (both incoming and outgoing) of the vertices being merged. Merging leads to a smaller graph containing fewer vertices (and edges), which reduces the execution time. If there is a triangle $\mathsf{a}-\mathsf{b}-\mathsf{c}$ and $\mathsf{a}-\mathsf{b}$ get merged, then we assign the weight to the edge $\mathsf{ab}-\mathsf{c}$ as the mean of the weights of the edges $\mathsf{a}-\mathsf{c}$ and $\mathsf{b}-\mathsf{c}$.

The merging can, in general, be performed on an arbitrary pair of vertices. But it reduces inaccuracy if performed carefully. We enable vertex-merging for a pair of vertices if their neighborhoods are *similar*. Two vertices have similar neighborhoods if their Jaccard's coefficient is above a threshold. Jaccard's coefficient J_{ij}, for vertices v_i and v_j with sets of neighbors $N(v_i)$ and $N(v_j)$ respectively, is

$$J_{ij} = \frac{|N(v_i) \cap N(v_j)|}{|N(v_i) \cup N(v_j)|}. \tag{3}$$

As vertices get merged, they form a meta-vertex, which, in turn, may get merged with another vertex or meta-vertex, and so on. v_i or v_j in Equation (3) may represent an original vertex or a meta-vertex. The merging order is important to the quality of the approximate representation (see Section 2.1). We merge the vertices using a greedy heuristic, prioritizing merging vertices with higher degrees.

Node-merging necessitates logical merging of the incoming and the outgoing edges of the vertices being merged. That is, neighbors of the vertices become neighbors of the merged vertex (removing self-loops if there was an edge between the vertices being merged).

The minimized graph thus obtained is fed as an input to the exact version of the algorithm. In our experiments, we find that decreasing the merging threshold of the Jaccard coefficient increases the inaccuracy. Decreasing the threshold also increases the speedups we obtain, in most cases. This is because decreasing the threshold decreases the number of vertices in the minimized graph, though the number of outgoing edges for a vertex may increase. We need to choose the Jaccard's threshold judiciously so as to achieve

good performance benefits while keeping within the acceptable limits of inaccuracy compared to the exact version.

3.5 Technique 4: Approximate Attribute Values

Our fourth proposal is to reduce the computation cost of large graphs by approximating attribute values of graph elements. Numeric attribute values (such as vertex distances or edge weights) can be rounded-off to discrete values, say powers of 2. Non-numeric values can be changed to be chosen from a smaller domain (e.g., vertex colors). We discuss below applying such a discretization for SSSP.

In SSSP, discretization is a two-step process. In the first step, we perform a traversal through the edges to find the maximum and the minimum weight edges. Let the maximum and the minimum edge-weights be w_{max} and w_{min} respectively. In the second step, we perform another traversal on the edges to modify their edge-weights. w_{min} is rounded-up to its nearest power of 2 and w_{max} is rounded-down to the nearest power of 2. We call these new limits as w'_{min} and w'_{max} respectively. All the edge weights are rounded to their nearest power of 2 in the range $[w'_{min}, w'_{max}]$. With the above modification, we are guaranteed that any edge in the graph can take only one of the $k \triangleq \{log_2(w'_{max}) - log_2(w'_{min})\}$ values. Assuming each edge takes any of these values with equal likelihood, an edge has an expected weight calculated as below.

Let X be a random variable which is defined as the weight assigned to an edge $e \in E$. X can take values from the set $S = \{w'_{min}, \ldots, w'_{max}\}$. An edge can be assigned any of these values with probability $\frac{1}{k}$. So

$$\begin{aligned} E[X] &= \sum_{x \in S} x.Pr(X = x) \\ &= \sum_{x \in S} x.\frac{1}{k} \\ &= \frac{2 \times w_{max} - w_{min}}{k}. \end{aligned}$$

So in expectation, the maximum distance between any two vertices can be $(graph\ diameter) \times \frac{2 \times w_{max} - w_{min}}{k}$.

We can make an informed choice about initial attribute values which would aid in reducing the portion of the graph we process. For instance, in case of SSSP computation, we can initialize the distance from the source to every vertex to some value other than the customary ∞. Such a value is computed as a preprocessing step as follows. We run a single pass of the Breadth-First-Search (BFS) on the graph starting at the source vertex s. This gives us the hop distance of every vertex from the source vertex. During the traversal, we also find the largest edge-weight value. Now, we set the initial distance of every vertex as: $v \in V$, $dist(s, v) = (\#\ of\ hops\ from\ s\ to\ v) \times (max\ edge\ weight)$.

Approximation of edge-weights, with a careful extra preprocessing, can enable us to transform the SSSP computation into an easily parallelizable BFS. As a preprocessing step, we run BFS on the given graph, from the source vertex s to get the level information of all the vertices with respect to s, in the form of the BFS tree rooted at s. In this BFS tree, we compute a weighted mean of the weights of the edges from level i to $i+1$, where $i \in \{0, 1, 2, \ldots\}$. The weights are drawn from a uniform distribution with values in the range (0, 1). We assign this weighted mean to all the edges from level i to $i+1$ and do the same for all the levels. After assigning the new edge-weights to all the edges in the BFS tree, a traversal of the tree gives the approximate shortest path distance of every vertex from the source node. For computing the weighted mean, the weight assigned to an edge-value is inversely proportional to it, i.e., higher edge-value is multiplied by a lower weight. This is done so that the weighted mean is not skewed towards the higher edge-values and is only slightly away from the exact shortest path length. This approximate technique makes it feasible to achieve good speedups of the iterative SSSP computation with plausible error bounds.

Similarly, for the PageRank algorithm, we initialize the pageranks of all the vertices to $\frac{1}{n}$ and not to some arbitrary discrete value. This serves two purposes. First, $\frac{1}{n}$ implies that the surfer lands on each page with equal probability. Second, since the PageRank computation essentially gives an estimate of the likelihood that the surfer lands on each page, the value does not vary drastically from the initial value. Hence, even here, we can afford to process the graph only partially and still have a reasonable solution. It has the effect that in large graphs every unprocessed vertex can be reached with a fairly low probability.

In case of MST, we round-up or -down the edge-weight to the closest power of 2. We stop when the weight of the MST overshoots a threshold set to $(n-1) \times (mean\ weight)$ rounded to the nearest power of 2, where n is the number of nodes in the graph. We find that this scheme leads to better execution time compared to the exact version.

In BC, the betweenness centrality value of each vertex is in the range [0..1]. We sub-divide this range into 10 equal-sized buckets, and the centrality value of each vertex is rounded to the nearest tenth.

4 Benefits to GPU-Based Processing

Though the proposed approximation techniques can be applied independent of the architecture, employing them on a parallel GPU code is particularly useful since they address important performance bottlenecks. Such bottlenecks are artifacts of issues such as synchronization, workload-imbalance, CPU-GPU data transfer, etc. We discuss how our approximation techniques help in diminishing their effect.

4.1 Technique 1: Reduced Execution

Typically, a graph algorithm gets modeled in a bulk-synchronous fashion where the host code repeatedly calls the processing kernels. Such processing involves an implicit barrier at the end of each kernel invocation (e.g., Line 14 in Algorithm 1). The approximation technique of reducing the number of iterations of the algorithm reduces the number of barriers invoked. For algorithms that require a large number of iterations, the cumulative effect of reducing the number of barriers helps improve performance.

We also observe that in typical algorithms, the amount of work done in later iterations is relatively much lesser (see, for example, Figs. 7 and 10). Especially in the context of massive-multithreading, such a behavior reduces the parallel work-efficiency. Reduced execution mitigates such an effect, and improves average work efficiency.

Graph	$\|V\|$ $\times 10^6$	$\|E\|$ $\times 10^6$	Graph type
rmat28	268.4	1073.7	R-MAT using GTgraph [16]
random	134.2	1073.7	Random graph using GTgraph [16]
LiveJournal	4.8	68.9	Social network
USA-road	23.9	57.7	Road network, large diameter
twitter	41.6	1468.3	Twitter graph 2010 snapshot

Fig. 2. Input graphs.

4.2 Technique 2: Processing Part of the Graph

In case of partial processing, the algorithm operates on only a subset of the graph. Processing fewer edges translates to lesser synchronization in each iteration. For instance, in SSSP computation of Algorithm 1, which uses an atomic operation at Line 10, reducing the number of processed edges implies that the number of incoming edges to a node reduces, thereby reducing the number of atomic operations.

Partial processing also helps partially address the problem of unbalanced work distribution among threads in a vertex-centric GPU implementation. Load imbalance happens due to a few vertices having large outdegrees (as in social networks). Due to the approximation of partial processing, however, the number of edges processed by each thread is lesser, mitigating the effect of load imbalance.

4.3 Technique 3: Approximate Representation

One of the primary bottlenecks in CPU–GPU systems is the inter-device data transfer over a relatively slow PCIe interconnect, especially for large graphs. Thus, it is desirable that the number of CPU–GPU transfers be reduced and the amount of data being sent be also small. With approximate graph representation, the device-to-device data transfer reduces, leading to performance benefits.

4.4 Technique 4: Approximate Attributes

Approximating attribute values helps in achieving the fixed-point in fewer iterations, leading to reduced synchronization in terms of implicit barriers. In case of algorithms such as SSSP, we approximate the vertex attributes to get rid of the atomic operations. We initialize vertex distances using a BFS-based approximation, which enables us to transform the SSSP computation to an easily parallelizable level-by-level BFS processing. Since level-synchronous BFS can be implemented without explicit atomic instructions [13], avoiding the synchronization improves performance.

In case of MST computation, which often requires several iterations to converge, we devise a policy for it to converge faster. MST's parallelism profile suggests that it has a good amount of parallelism initially, which reduces as the algorithm progresses. Online approximation of attributes, across iterations, helps us terminate the algorithm early. By rounding the edge attributes to powers of two and setting a suitable threshold (Section 3.5), the algorithm makes rapid strides and converges faster.

5 Experimental Evaluation

In this section we evaluate the performance of the various approximation techniques, and compare it with the exact versions of the respective algorithms. We study six graph algorithms: single-source shortest paths computation (SSSP), minimum spanning tree computation (MST), finding strongly connected components, vertex coloring (Color), page rank and node betweenness centrality computation (BC). We compare our approximate SSSP and approximate MST with the respective exact versions from LonestarGPU [7], approximate SCC with the exact SCC by Devshatwar et al. [9], Color with our parallel implementation of exact largest-degree-first (LDF) coloring algorithm, approximate PR with Totem [14] and BC with our parallel implementation of exact Brandes' algorithm.

Algo.	Graph	Exact Time(sec)	Algo.	Graph	Exact Time(sec)
SSSP	rmat28	43	Color	rmat28	16
	random	29		random	18
	LiveJournal	2		LiveJournal	5
	USA-road	207		USA-road	10
	twitter	231		twitter	32
MST	rmat28	8996	PR	rmat28	12
	random	10087		random	16
	LiveJournal	3424		LiveJournal	1
	USA-road	82		USA-road	1
	twitter	10943		twitter	18
SCC	rmat28	21	BC	rmat28	15223
	random	23		random	13127
	LiveJournal	7		LiveJournal	1711
	USA-road	12		USA-road	2043
	twitter	37		twitter	21462

Fig. 3. Execution time for the exact versions.

We perform experiments on a machine with an Intel Xeon 32-core E5-2650 v2 @ 2.6 GHz CPU having 100 GB RAM and Nvidia Kepler (Tesla K40C) GPU having 2,880 cores spread across 15 SMXs with 12 GB memory. The machine runs CentOS 6.5. We use CUDA 6.5 to compile and execute our methods on the GPU. We evaluate our approach on a range of input graphs from SNAP [15] shown in Fig. 2. The base execution times (in second) for the exact versions are listed in Fig. 3. While other techniques do not require any preprocessing, approximate graph representation needs to compute neighborhood similarities to calculate Jaccard's coefficient. Since this is a one-time cost, we do not account for this preprocessing in the execution time.

An important aspect of measuring the effectiveness of approximations is to compare the accuracy of the computed values. This can be achieved by computing an absolute difference between the attribute values of the vertices for the exact and the approximate versions, and taking an average across vertices for a run. For multiple runs (say, across graphs), we compute the geomean difference over the averages for each run. For SSSP, the attribute is the distance value; for PR, it is the page rank value; and for BC, it is the betweenness centrality value. For SCC, we calculate the difference in the number of SCCs computed by the exact and the approximate methods. For MST, we calculate the difference in the weight of the minimum spanning tree computed by the two methods. Such a mechanism does not work for discrete values such as colors in vertex coloring. A straightforward solution is to use number of colors as a measure to compare. However, due to the non-determinism in thread-scheduling, multiple runs of our coloring algorithm may result in different colorings; leading to a difference in the number of colors used for the same graph. In our implementation of LDF algorithm, this happens when one or more neighbors of a vertex have the same degree. Therefore, the baseline accuracy of such an exact version cannot be

Algo.	Technique	Mean Speedup	Mean Inaccuracy
SSSP	Outer-loop iterations	1.49	6.34%
	Partial processing of graph	1.47	17.82%
	Approx. graph representation	1.27	14.37%
	Approx. attributes	1.92	17.64%
MST	Outer-loop iterations	1.22	14.08%
	Partial processing of graph	1.74	17.23%
	Approx. graph representation	1.56	15.5%
	Approx. attributes	1.48	19.07%
SCC	Outer-loop iterations	1.26	16.48%
	Partial processing of graph	1.32	19.50%
	Approx. graph representation	1.45	21.5%
	Approx. attributes	–	–
Color	Outer-loop iterations	1.45	**29.18%**
	Partial processing of graph	1.28	16.43%
	Approx. graph representation	1.36	18.39%
	Approx. attributes	–	–
PR	Outer-loop iterations	2.03	**2.45%**
	Partial processing of graph	1.82	12.76%
	Approx. graph representation	1.54	13.63%
	Approx. attributes	–	–
BC	Outer-loop iterations	1.74	18.07%
	Partial processing of graph	1.42	16.73%
	Approx. graph representation	1.33	14.35%
	Approx. attributes	1.41	23.16%

Fig. 4. Overall results.

faithfully captured in the parallel setting. To address this, we measure the accuracy as the percentage of *pairs* of adjacent nodes having different colors. Such a quantity indicates the degree of closeness with the exact coloring (which would have this value as 100 percent), and importantly, it will be independent of the thread-scheduling.

5.1 Overall Results

Fig. 4 summarizes the effects of the four approximation techniques for the six graph algorithms. We list the geomean speedup and the inaccuracy values across all the graphs in our setup. The approximation of the attribute values is inapplicable for PR, SCC and Color; hence the entries are marked as –. Overall, we observe that the approximations have the capability to achieve high speedups by trading off more and more accuracy. However, some algorithms seem to be more amenable to effective approximation than others. For instance, approximate SSSP and PR achieve relatively higher speedups for lower inaccuracy compared to MST, SCC, Color and BC. This indicates that continuous-value-based algorithms (such as SSSP and PR) provide better approximation opportunities than discrete-value-based (such as Color). Second, algorithms that depend heavily on the graph structure (such as SCC and Color) have a relatively higher inaccuracy. This is an artifact of values getting refined in each iteration, but structures are often binary (either an SCC or not, either a neighbor or not), which affects approximation opportunities. High inaccuracy for outerloop iterations in Color is because reduced execution for Color translates to using fewer colors.

We now look at the overall effect of individual approximations; detailed discussion follows in the subsequent sections. First, the effect of reduced execution follows value-based approximations—more effective for SSSP and PR (with lower inaccuracy and higher speedup) and less so for MST, SCC and Color. BC gets benefitted in execution time (1.74×), but at the cost of high inaccuracy (18.07 percent), due to BFS from fewer source vertices. However, PR appears to be exceptionally benefitted by this approximation, as the page rank values converge rapidly to their final values in a few iterations. Thus, for algorithms where fixed-points are approached quickly and then refined slowly, reducing execution turns out to be very useful. Therefore, it is a useful approximation for gradient-descent kind of algorithms.

Second, partial graph processing is uniformly useful across algorithms, with high speedups, but the usefulness is offset by a higher inaccuracy. This is an indication of algorithms working on graph properties that are *global*, and get affected by most of the graph elements. For instance, removing some edges of the graph would affect shortest path or page rank value propagations. Partial graph processing is more beneficial for algorithms that compute *local* properties, such as computing the maximum clique or minimum spanning tree. In such problems, removal of a few edges or nodes would have a reduced probability of affecting the max-clique or MST, leading to the approximation being more effective. We observe such behavior for MST wherein the performance of the approximate version with partial graph processing is particularly higher (speedup of 1.74×) compared to other techniques (speedup $\leq$ 1.5×). For BC, partial processing achieves moderate benefits.

Third, approximate graph representation (using Jaccard's similarity) is consistently beneficial for performance, with relatively better accuracy (compared to other techniques). This is understandable for structure-based algorithms such as SCC. Even for Color, since two vertices with almost common neighborhood can be given the same color, merging them is likely to maintain accuracy. A similar propagation effect happens in case of PR—common neighbors propagate common values across vertices—hence merging the nodes does not adversely affect accuracy. However, such a merging approximation is unlikely to be useful for SSSP where edge-weights play a major role despite the common neighborhood. Therefore, we observe reduced benefits due to this approximation for SSSP (speedup of 1.27×), compared to other techniques (speedup $>$ 1.3×). For BC, setting the Jaccard's similarity threshold for merger moderately affects the speedup. A lower threshold aggressively merges the vertices but, in turn, increases the number of neighbors of the meta-vertex. On the other hand, setting a high Jaccard's similarity threshold merges fewer vertices.

Finally, approximating values is applicable for weighted algorithms such as SSSP and MST. We have also applied it to unweighted BC, approximating the vertex attribute, that is, the BC value. While this approximation achieves good performance for both the weighted algorithms, the inaccuracy is higher for MST (19.07 percent) compared to SSSP (17.64 percent). This happens because of the algorithm's behavior—MST is implemented using Boruvka's algorithm which merges components based on the lightest inter-component edge. Thus, the number of choices for inter-component edge increases after the power-of-2 approximation—which can lead to a different edge getting selected. An accumulation of errors across multiple iterations leads to increased inaccuracy for MST. A similar effect happens in SSSP too, but since the effect is restricted to choosing the minimum distance across neighbors (rather than across a collection of vertices), the effect is small, leading to better accuracy. For BC, we observe that the decent speedup of

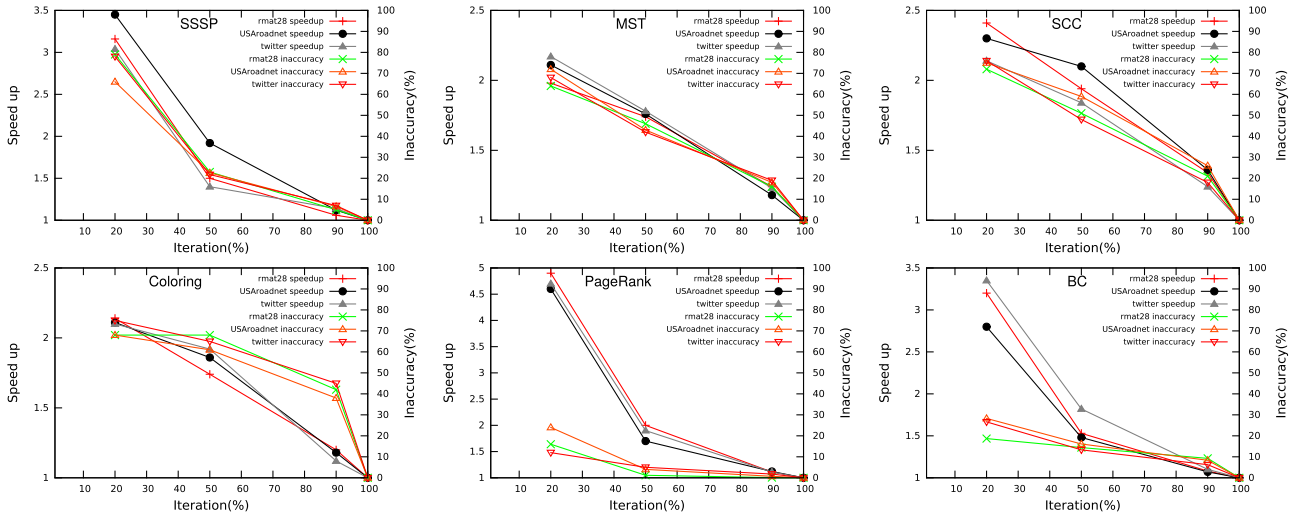

Fig. 5. Algorithm-wise effect of varying the percentage of outer loop iterations.

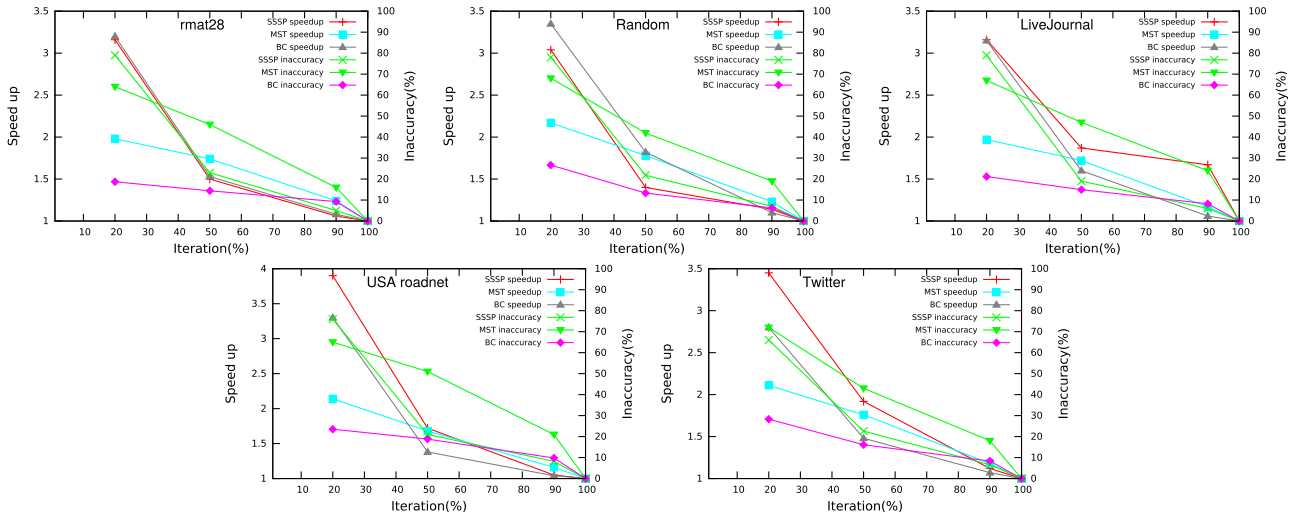

Fig. 6. Graph-wise effect of varying the percentage of outer loop iterations.

1.41× is accompanied by a high inaccuracy of 23.16 percent. This is primarily due to the per-iteration approximation of BC values. To be specific, we discretize the BC values after each iteration to the nearest tenth. The algorithm halts when the difference in the BC values across iterations is less than a threshold.

We discuss the effect of each technique in more detail.

5.2 Effect of Reduced Execution

Fig. 5 presents the effect the reduced execution technique (Section 3.2) on the six algorithms. To avoid clutter, we show results for three largest graphs. We observe that by trading off some accuracy, one may enjoy considerable performance benefits. From the shape of the plots, we see that algorithms which compute global properties such as MST and SCC, the inaccuracy almost linearly follows the added approximation. However, SSSP and BC depend upon a source vertex and perform the computation based on it. Such algorithms are more sensitive to reduced execution—they can achieve high speed-up with high inaccuracy. USA-road is a notable exception—the performance benefits due to reduced execution approximation is relatively low. This is an artifact of structural properties of the road networks. In particular, road networks have large diameters and uniform degree distribution. This is in contrast to other networks that follow *small-world* property and have power-law degree distribution. PR benefits substantially by the approximation on outer loop iterations with little drop in accuracy. This happens due to fast convergence of PR. On the other hand, Color has the highest overall inaccuracy with low-performance improvement. This happens in power-law graphs as there is a long tail of small degree nodes, only some of which get processed. Note that other algorithms do not process these scale-free graphs in degree order.

Takeaway 1. *Reduced execution approximation is beneficial for algorithms that converge quickly to the final solution.*

Takeaway 2. *Reduced execution approximation provides reduced benefits for algorithms whose precision gets affected by the long tail of vertices processed in scale-free graphs.*

Fig. 6 shows the effect of varying the percentage of outer loop iterations for various graphs (20, 50 and 90 percent iterations). We observe a good similarity in the plots across the graphs: not only the trend, but also the values are similar—which hints at the robustness of this

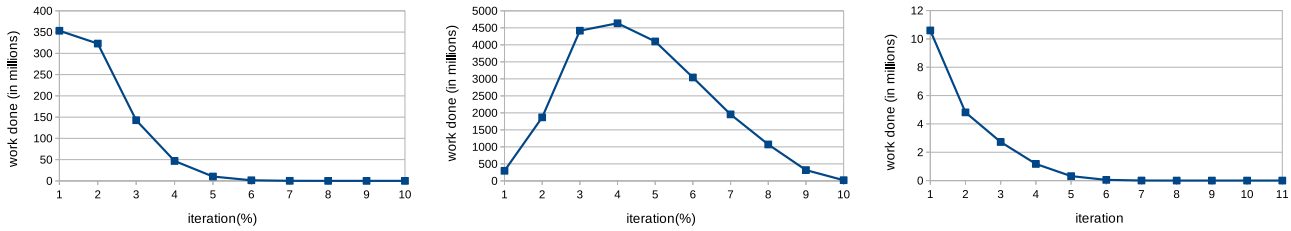

Fig. 7. Work done per iteration in SSSP for rmat28, USA-road and LiveJournal.

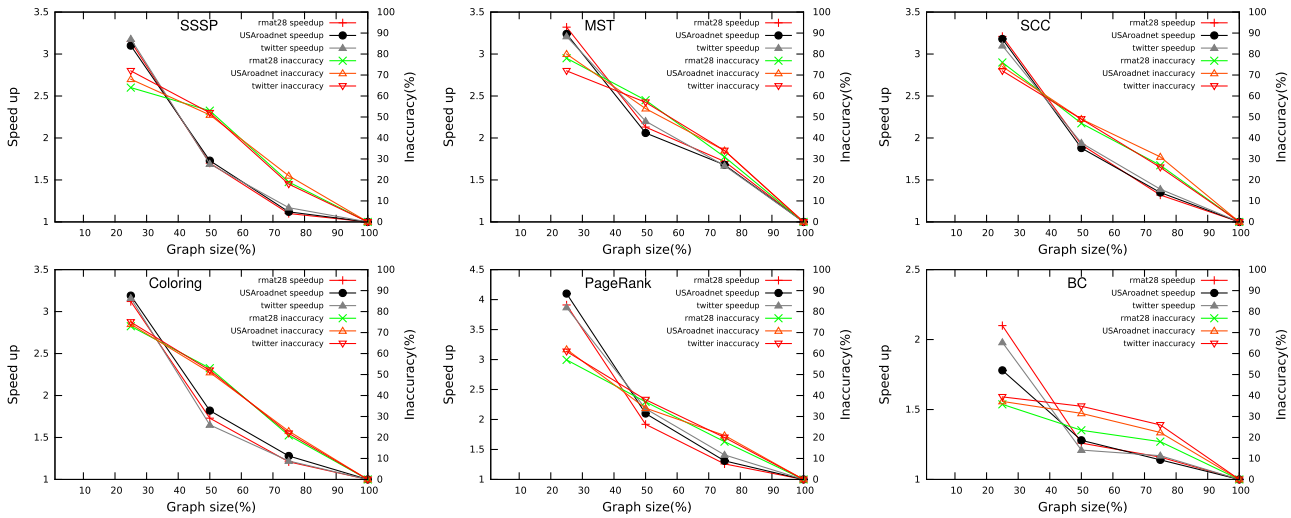

Fig. 8. Algorithm-wise effect of varying the percentage of graph processed.

technique (as we will see, not all techniques exhibit this robustness).

Fig. 7 shows the work done (number of vertex distances settled) per iteration in exact SSSP for various graphs. We can choose to reduce the execution based on how much work is sufficient for the algorithm. The amount of work done is directly proportional to the accuracy and inversely proportional to its execution time.

5.3 Effect of Partial Graph Processing

Fig. 8 presents the effect of processing part of the graph (Section 3.3). A striking difference with the reduced execution (Fig. 5) is that the accuracy of the results is largely uniform across graphs as well as across algorithms. It is interesting to observe that the speedup effect differs considerably across algorithms (MST being more amenable to this approximation over SSSP), but the inaccuracy values do not. For BC, we observe that the effect of partial graph processing on the scale-free graphs is relatively small in terms of the impact on inaccuracy than that on the road network. This is due to the difference in diameters. For low-diameter graphs, vertices can be reached from one another by traversing only a few edges. Therefore, processing only a fraction of the edges still has a high probability of traversing from one vertex to another, reducing the overall BC error. This is an indication that the inaccuracy of partial graph processing depends primarily on the amount of graph processed (more detailed results follow). This is expected, but not always true with other approximations.

Fig. 9 shows the effect of varying the percentage of the graph processed for each graph in our testbed. We observe a considerable similarity in the shapes of the plots indicating a near-uniform effect of this approximation across graphs. There is some variation in the behavior for the road network USA-road in MST computation, but otherwise, the speedups and the inaccuracy values follow the trend.

Takeaway 3. *Partial graph processing affects computation in a uniform manner across various graphs in our testbed.*

Fig. 10 shows the effect of processing part of the graph in SSSP for rmat28, USA-road and LiveJournal. It plots the amount of work done in each iteration for the number of edges processed as 100 percent (exact), 50 and 25 percent. The three plots show different shapes of these curves: for low-diameter graphs such as rmat28 and LiveJournal, the amount of work done is initially high and reduces gradually; whereas for road networks, the work done is high in the middle (due to uniform degree distribution). In all the cases, we observe that the approximate versions clearly perform much lesser work, leading to better performance.

Fig. 11 shows the variation in work done per iteration for partial graph processing in case of Color. In this algorithm, we consider work done to be the number of nodes colored. An interesting observation is that, unlike in SSSP, the shapes of the plots remain the same and are not guided by the diameter. This occurs because coloring follows the largest-degree-first processing, and thus performs more work initially. Based on this observation, one may wish to reduce the fraction of the edges processed with increasing number of iterations.

5.4 Approximate Graph Representation

Approximate graph representation offers relatively higher benefits compared to the other approximation techniques.

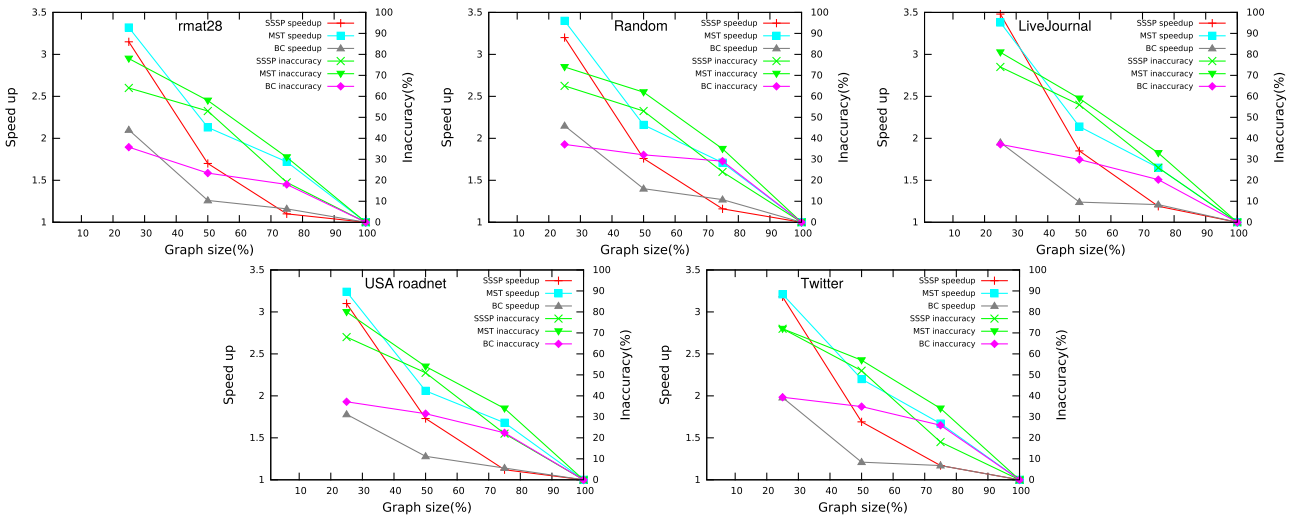

Fig. 9. Graph-wise effect of varying the percentage of graph processed.

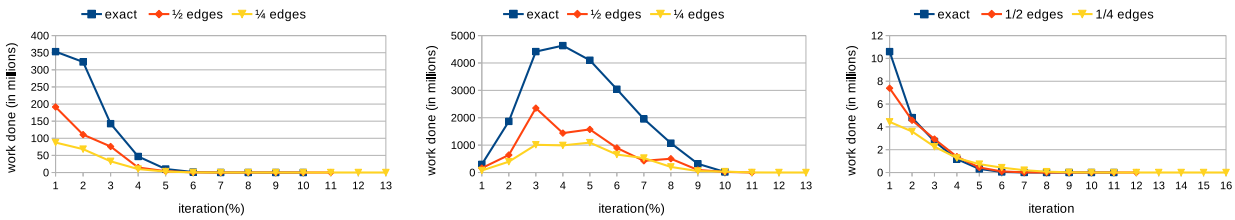

Fig. 10. Work done across iterations in SSSP for rmat28, roadNet-USA, and LiveJournal due to partial graph processing.

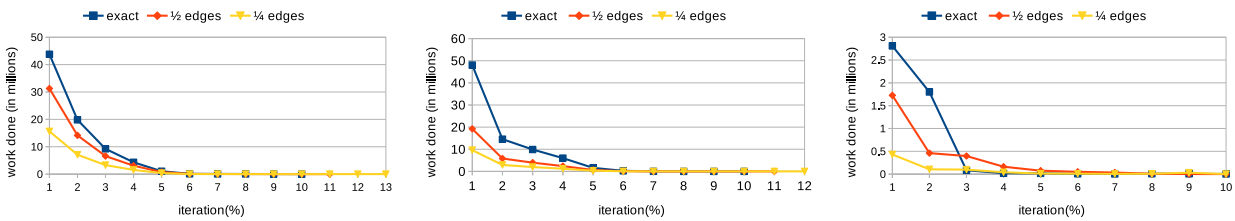

Fig. 11. Work done across iterations in Color for rmat28, roadNet-USA, and LiveJournal due to partial graph processing.

Fig. 12 shows the algorithm-wise effect of approximate graph representation using Jaccard's coefficient. We observe a uniformity in the behavior both in terms of trend and magnitude. An interesting aspect is that the speedups and the inaccuracy values get clustered for this approximation for each graph. This occurs because similarity measure makes sure that the nodes being merged are indeed similar.

Fig. 13 shows the graph-wise effect of the approximation. We observe a similar trend across various algorithms, and there is also a high uniformity in the magnitudes.

Takeaway 4. *Partial graph processing with Jaccard's similarity affect speedup and inaccuracy uniformly in our testbed.*

Fig. 14 plots the work done per iteration for different thresholds of Jaccard's index for SCC and SSSP computations. For SCC, work done is defined as the number of vertices changing their component. For both the algorithms, the overall work done per iteration reduces as we lower the Jaccard's index threshold for merging of vertices. This is due to power-law degree distribution for LiveJournal. When the threshold for merging is high (J-index = 0.8), the merger causes the higher degree vertices to merge while the smaller degree nodes are largely left unmerged. So the number of nodes reduces but the degree of the merged nodes increases. As we reduce the threshold for merging (J-Index = 0.6), the merger causes even the smaller degree nodes to merge.

5.5 Approximate Attribute Values

Fig. 15 presents the effect of approximating the attribute values in SSSP, MST and BC, which work on weighted graphs (see Section 3.5). We observe relatively higher benefits with moderate inaccuracies for SSSP and MST, but consistently high inaccuracies for BC. On an average, we observe a 17 percent inaccuracy with a harmonic mean speedup of 1.9× in SSSP. For MST, an average inaccuracy of 19 percent fetched a speedup of around 1.4×. For BC, we observe a speedup of 1.4× and a high inaccuracy (∼23 percent). While the speedup is encouraging, the high inaccuracy is due to discretization of the BC values obtained after *every* iteration.

Takeaway 5. *Approximating attribute values achieves better speedup at the cost of accuracy in our testbed.*

Fig. 16 plots the work done per iteration in MST for rmat28 and USA-road. Work done per iteration is measured as the number of edges contracted. We observe that the work done per iteration with approximate attribute values

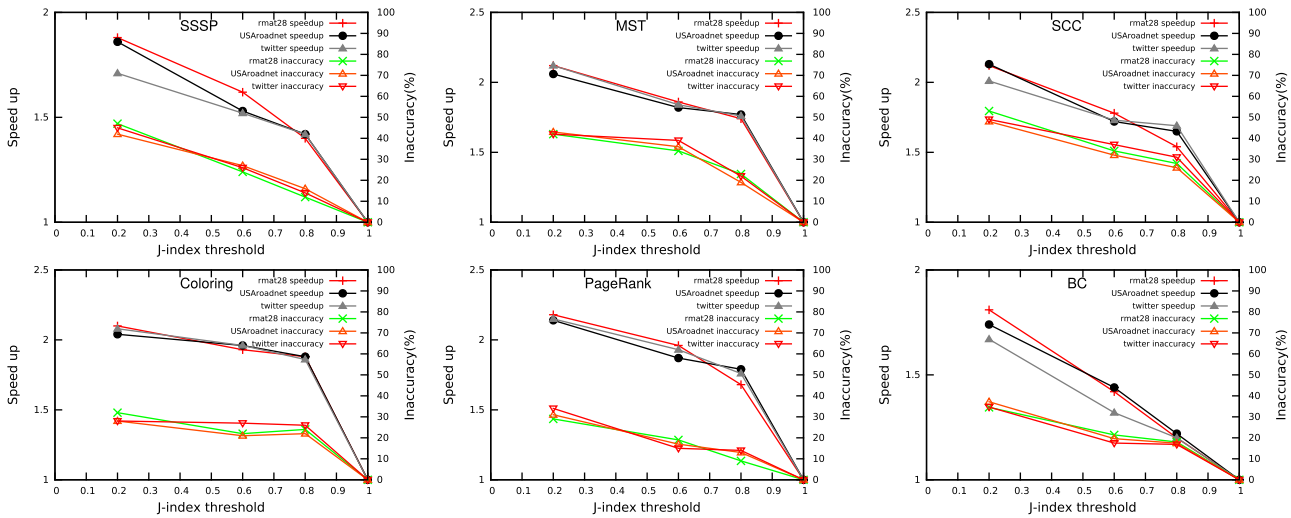

Fig. 12. Algorithm-wise effect of varying the Jaccard's coefficient.

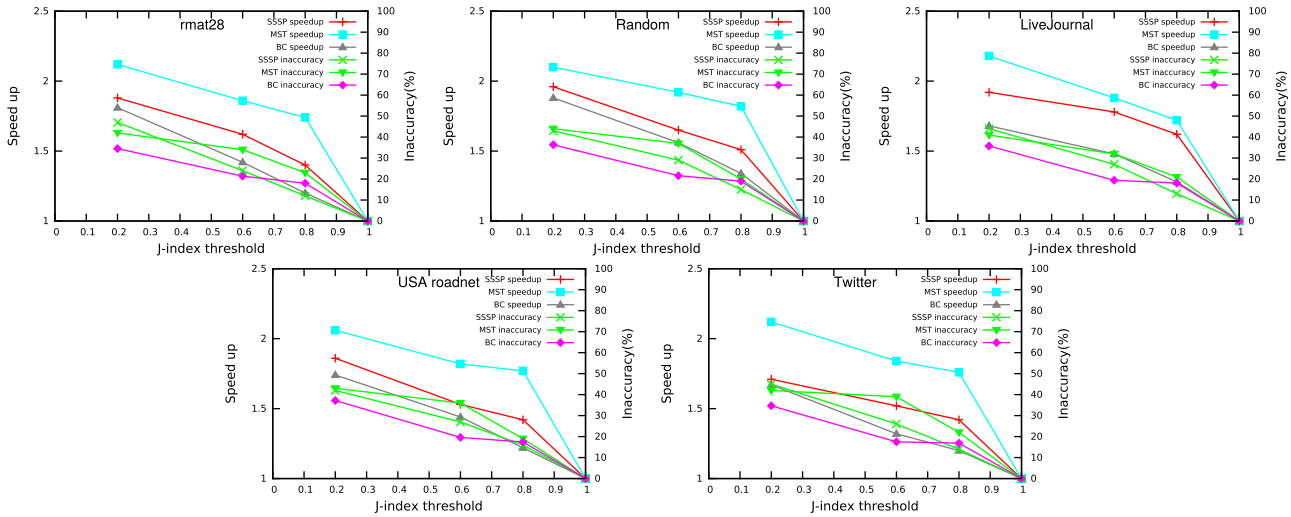

Fig. 13. Graph-wise effect of varying the Jaccard's coefficient.

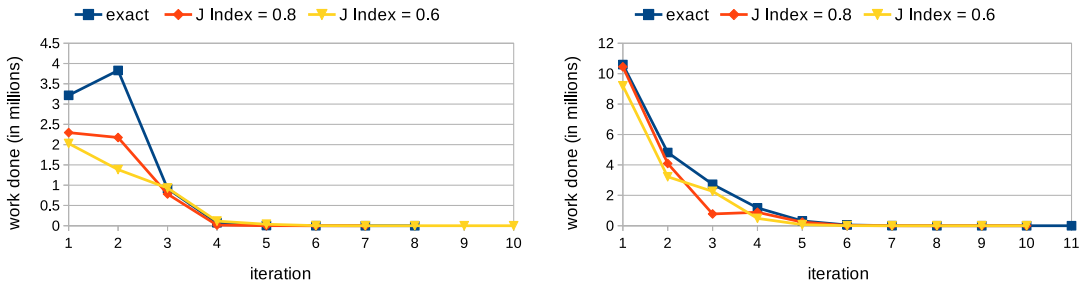

Fig. 14. Work done per iteration for LiveJournal in SCC and SSSP for varying Jaccard's coefficient.

closely follows the exact version for USA-road. Note that it does not mean that the two versions contract the *same edges*; it simply indicates that they contract almost the *same number of edges*, but the approximate version converges faster. This reduces the overall work done and, correspondingly, the execution time. In case of rmat28, the work done per iteration reduces considerably due to approximating attribute values.

5.6 Effect on Graph Type

Our testbed consists of scale-free low-diameter graphs (rmat28, LiveJournal and Twitter), Erdös-Renyí style random graph (random), and large diameter road network (USA-*road*). Work-done per iteration for scale-free graphs is initially high and quickly reduces, and remains low for several iterations (long tail). In contrast, for large diameter graphs, it increases in the initial iterations, remains high for some iterations, and then gradually reduces. For random graphs, it remains almost uniform throughout. This behavior dictates how an approximation affects processing on these graphs. Reduced execution is more useful for large diameter graphs. Partial graph processing changes only the magnitude of work done per iteration. Hence, it affects graphs uniformly. A similar behavior is observed with approximate

	Graphs	Speedup	Inaccuracy
SSSP	rmat28	2.65	14%
	random	1.42	19%
	LiveJournal	2.18	21%
	USA-road	2.06	17%
	twitter	1.58	18%
MST	rmat28	1.58	16%
	random	1.43	22%
	LiveJournal	1.64	19%
	USA-road	1.29	21%
	twitter	1.53	18%
BC	rmat28	1.40	21%
	random	1.53	23%
	LiveJournal	1.34	22%
	USA-road	1.41	27%
	twitter	1.42	25%

Fig. 15. Effect of approximating attribute values.

graph representation. However, we see that the performance and the inaccuracy values are more sensitive to partial graph processing compared to the approximate graph representation. Finally, the effect of approximating attribute values is not conclusively dependent upon the graph type, as graph type is a structural property whereas attribute value is a numeric property of the graph elements.

Summary. Overall, we note that our proposed approximations bear the potential to trade-off performance and accuracy very effectively. While, in general, algorithmic processing and input graphs affect the magnitude of benefit, we see that approximations consistently offer considerable improvement. We believe that practical large-scale algorithms on GPUs would find our approximation proposals suitable.

6 Related Work

A survey of approximations is presented by Mittal [17].

Gubichev et al. [18] presented a preprocessing-based technique for finding the approximate single source shortest path from a designated source node for a given graph. The precomputation step involves computing for every node in the graph a shortest path to and from a small number of *landmark* nodes. The obtained set of paths are stored in external memory. The precomputed information is used to provide a fast approximation of the node distance at query time. It works by combining the distance of the query nodes, s and d to or from a selected landmark node l into the approximate distance $\tilde{d}$(s,d). They implement a few other optimizations—Cycle Elimination, Shortcutting, on top of the basic scheme to further improve the query time. The proposed scheme is implemented on an *RDF-3X* system. Our method, on the contrary, does not perform any preprocessing to store the shortest path distances.

A* search [19] heuristic was proposed to compute approximate shortest distances with a reduced latency compared to the traditional algorithms. They also establish plausible error bounds on the computed solution. This heuristic is complementary to our techniques. Agarwal et al. [20] propose a look-up based iterative approach to efficiently compute point-to-point shortest paths for a large fraction of source-destination pairs. They maintain a data structure to store the exact distance to the immediate neighbor of a node and also its two hop neighbors. This enables them to trace the shortest path along with reporting the shortest distance.

Grosset et al. [21] propose a parallel graph coloring algorithm for GPUs. The algorithm proceeds in three phases. First, the graph is partitioned so that there is load-balance across threads. In the second phase, each thread assigns colors to its subgraph, using a heuristic—first fit, highest saturation then highest degree, maximum degree out of the partition, minimum degree out of the partition. In the third phase, conflicts are resolved sequentially on the CPU. The proposed GPU implementation provides better results in terms of the number of colors used, and the running time is comparable to the sequential First Fit algorithm.

Deveci et al. [10] propose an edge-based graph coloring approach. It is shown to be equivalent to $\Delta + 1$ coloring. It is empirically shown that edge-based coloring usually outperforms other algorithms on GPUs. They implement the scheme using the Kokkos library. In contrast, our coloring algorithm employs the largest-degree first heuristic.

There is a large body of work on graph compaction techniques. *Spanners and sparsifiers* have been studied extensively with respect to approximate graph representation. These are sparse subgraphs of the original graph from which properties of the original graph can be approximated. Spielman and Teng [22] construct spanners by performing a natural random rounding of the graph to achieve a good approximation of the original graph. Another popular method for obtaining approximate graph representation is sampling. Benczúr and Karger [23] propose sampling graph edges with varying probabilities. The compressed graph is built by including an edge e with probability p_e and assigning it a weight of $\frac{1}{p_e}$ if it is included. In contrast, our technique for graph compaction uses Jaccard's coefficient for similarity.

There has been some work on parallelization of BC computation. Sariyüce et al. [12] discuss various techniques for speeding up the exact betweenness centrality computation on GPUs and on heterogeneous CPU/GPU architectures by exploiting the coarse-grained and fine-grained parallelism available in Brandes' algorithm.

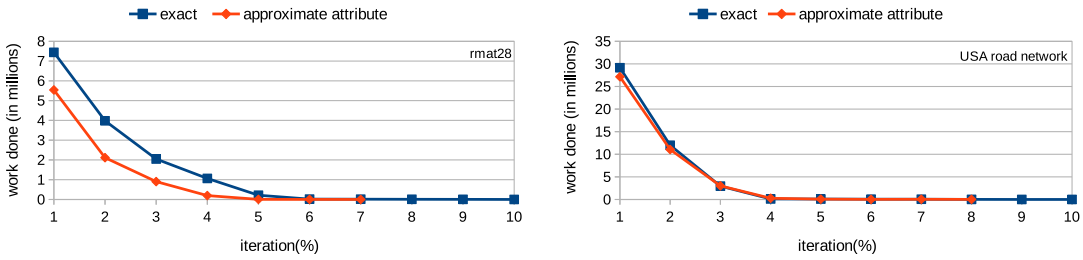

Fig. 16. Work done per iteration in MST due to approximating attribute values.

There have also been attempts at approximating the betweenness centrality of nodes. Riondato and Upfal [24] propose a progressive sampling based family of algorithms to approximate the BC values in a graph. They also establish probabilistic guarantees on the solutions. The method outperforms the exact versions and other approximation algorithms with the same guarantees on quality. In comparison, the approximation techniques we apply to Brandes' algorithm result in inexact but mostly deterministic solutions.

Seo et al. [25] propose *GStream*, a GPU based exact method for processing large-scale graphs that do not fit entirely in the GPU device memory. Nasre et al. [26] implement Boruvka's algorithm in CUDA for finding the MST on a weighted undirected graph through successive *edge contractions*. Nobari et al. [27] propose a GPU based parallel minimum spanning forest (MSF) algorithm. They propose a parallelized version of Prim's algorithm where they concurrently expand several subsets of the computed MSF. Our baseline for MSF is Boruvka's algorithm.

Hong et al. [28] present a scalable implementation for finding strongly connected components, which performs well on diverse, small-world graphs. They improve over the conventional FW-BW-Trim algorithm by exploiting the data-level parallelism, letting every thread work on the same partition of the graph. All the threads are used to find the reachable sets. Subsequently, they return to the conventional implementation, which exploits task-level parallelism. In the current work, we apply approximations on top of the exact FW-BW-Trim algorithm.

Graph algorithms [29], [30] have been shown to bear enough parallelism especially in the context of distributed [31], [32], [33] and heterogeneous systems [14]. Merrill et al. [1] propose work-efficient graph traversal with several optimizations based on prefix-sum. Luo et al. [2] propose a BFS for multi-core CPUs and a single GPU. Their method uses a well-optimized hierarchical queue to keep track of frontiers. Hong et al. [3] propose multiple methods for BFS on a heterogeneous system. Their hybrid method chooses the best execution among sequential, multi-core CPUs and single GPU. Harish and Narayanan [34] propose a two-phase method for performing BFS. Fu et al. [35] propose BFS for GPU clusters. Being a distributed setup, their focus is communication across devices. Contrary to our work, these methods deal with exact graph algorithms.

7 Conclusion

We studied the effect of various algorithmic approximations on graph algorithms on GPUs. It is believed that for irregular computations such as graph algorithms, the effectiveness of a technique depends primarily upon the input. There exist algorithms, for instance, that target specially power-law graphs and which do not work well with large diameter graphs. On the contrary, our study reveals that while the amounts of performance improvement and inaccuracy vary, approximations are consistently helpful in achieving the trade-off well. In other words, approximate computation of graph algorithms is a robust way of dealing with irregularities. Our techniques are general and applicable to other graph algorithms as well. We believe that our proposals and their study would pave the way for many more algorithm-specific and algorithm-independent approximations.

References

[1] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proc. 17th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2012, pp. 117–128.

[2] L. Luo, M. Wong, and W.-M. Hwu, "An effective GPU implementation of breadth-first search," in *Proc. 47th Des. Autom. Conf.*, 2010, pp. 52–55.

[3] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proc. ACM Symp. Principles Practice Parallel Program.*, 2011, pp. 267–276.

[4] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proc. 22nd Annu. ACM Symp. Parallelism Algorithms Archit.*, 2010, pp. 303–314.

[5] E. Mastrostefano and M. Bernaschi, "Efficient breadth first search on multi-GPU systems," *J. Parallel Distrib. Comput.*, vol. 73, no. 9, pp. 1292–1305, Sep. 2013.

[6] T. Mitsuishi, J. Suzuki, Y. Hayashi, M. Kan, and H. Amano, "Breadth first search on cost-efficient multi-GPU systems," *SIGARCH Comput. Archit. News*, vol. 43, no. 4, pp. 58–63, 2016.

[7] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Proc. IEEE Int. Symp. Workload Characterization*, 2012, pp. 141–151.

[8] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan, "Fast minimum spanning tree for large graphs on the GPU," in *Proc. Conf. High Perform. Graph.*, 2009, pp. 167–171.

[9] S. Devshatwar, M. Amilkanthwar, and R. Nasre, "GPU centric extensions for parallel strongly connected components computation," in *Proc. 9th Annu. Workshop Gen. Purpose Process. Using Graph. Process. Unit*, 2016, pp. 2–11.

[10] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, "Parallel graph coloring for manycore architectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2016, pp. 892–901.

[11] S. Beamer, K. Asanovi, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2017, pp. 820–831.

[12] A. E. Sariyüce, K. Kaya, E. Saule, and U. V. Çatalyürek, "Betweenness centrality on GPUs and heterogeneous architectures," in *Proc. 6th Workshop Gen. Purpose Processor Using Graph. Process. Units*, 2013, pp. 76–85.

[13] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on GPUs," in *Proc. 27th IEEE Int. Parallel Distrib. Process. Symp.*, 2013, pp. 463–474.

[14] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu, "A yoke of oxen and a thousand chickens for heavy lifting graph processing," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2012, pp. 345–354.

[15] J. Leskovec and R. Sosič, "SNAP: A general purpose network analysis and graph mining library in C++," Jun. 2014. [Online]. Available: http://snap.stanford.edu/snap

[16] K. Madduri and D. A. Bader, "GTgraph: A suite of synthetic random graph generators," 2006. [Online]. Available: http://www.cse.psu.edu/madduri/software/GTgraph/, Accessed on: May 28, 2013.

[17] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surveys*, vol. 48, no. 4, pp. 62:1–62:33, Mar. 2016.

[18] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum, "Fast and accurate estimation of shortest paths in large graphs," in *Proc. 19th ACM Int. Conf. Inf. Knowl. Manage.*, 2010, pp. 499–508.

[19] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in *Proc. 16th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2005, pp. 156–165.

[20] R. Agarwal, M. Caesar, P. B. Godfrey, and B. Y. Zhao, "Shortest paths in less than a millisecond," in *Proc. ACM Workshop Online Social Netw.*, 2012, pp. 37–42.

[21] A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall, "Evaluating graph coloring on GPUs," in *Proc. 16th ACM Symp. Principles Practice Parallel Program.*, 2011, pp. 297–298.

[22] D. A. Spielman and S.-H. Teng, "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems," in *Proc. 36th Annu. ACM Symp. Theory Comput.*, 2004, pp. 81–90.

[23] A. A. Benczúr and D. R. Karger, "Approximating s-t minimum cuts in O(n2) time," in *Proc. 28th Annu. ACM Symp. Theory Comput.*, 1996, pp. 47–55.

[24] M. Riondato and E. Upfal, "ABRA: Approximating betweenness centrality in static and dynamic graphs with rademacher averages," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2016, pp. 1145–1154.
[25] H. Seo, J. Kim, and M.-S. Kim, "GStream: A graph streaming processing method for large-scale graphs on GPUs," in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2015, pp. 253–254.
[26] R. Nasre, M. Burtscher, and K. Pingali, "Morph algorithms on GPUs," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 147–156.
[27] S. Nobari, T.-T. Cao, P. Karras, and S. Bressan, "Scalable parallel minimum spanning forest computation," in *Proc. 17th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2012, pp. 205–214.
[28] S. Hong, N. C. Rodia, and K. Olukotun, "On fast parallel detection of strongly connected components (SCC) in small-world graphs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, pp. 92:1–92:11.
[29] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 211–222, 2007.
[30] K. Pingali, et al., "The tao of parallelism in algorithms," in *Proc. 32nd ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2011, pp. 12–25.
[31] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 65:1–65:12.
[32] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, "Breaking the speed and scalability barriers for graph exploration on distributed-memory machines," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 13:1–13:12.
[33] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2010, pp. 1–11.
[34] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. Int. Conf. High-Perform. Comput.*, 2007, pp. 197–208.
[35] Z. Fu, H. K. Dasari, B. Bebee, M. Berzins, and B. Thompson, "Parallel breadth first search on GPU clusters," in *Proc. IEEE Int. Conf. Big Data*, Oct. 2014, pp. 110–118.

Somesh Singh is working toward the PhD degree at the Department of Computer Science and Engineering, IIT Madras. His research interests include high-performance computing and parallel programming. His current focus is on designing approximation techniques for parallel graph algorithms on graphics processing units.

Rupesh Nasre received the PhD degree from IISc Bangalore and post-doctoral fellowship from the University of Texas at Austin. He is an assistant professor in the Department of Computer Science and Engineering, IIT Madras. His research focus is in parallelization.