Graffix: Efficient Graph Processing with a Tinge of GPU-Specific Approximations

Somesh Singh
Indian Institute of Technology Madras, India
ssomesh@cse.iitm.ac.in

Rupesh Nasre
Indian Institute of Technology Madras, India
rupesh@cse.iitm.ac.in

ABSTRACT

Parallelizing graph algorithms on GPUs is challenging due to the irregular memory accesses involved in graph traversals. In particular, three important GPU-specific aspects affect performance: memory coalescing, memory latency, and thread divergence. In this work, we attempt to tame these challenges using approximate computing. We target graph applications on GPUs that can tolerate some degradation in the quality of the output for obtaining the result in short order. We propose three techniques for boosting the performance of graph processing on the GPU by injecting approximations in a controlled manner. The first one creates a graph isomorph that brings relevant nodes nearby in memory and adds controlled replica of nodes to improve coalescing. The second reduces memory latency by facilitating the processing of subgraphs inside shared memory by adding edges among specific nodes and processing *well-connected* subgraphs iteratively inside shared-memory. The third technique normalizes degrees across nodes assigned to a warp to reduce thread divergence. Each of the techniques offers notable performance benefits, and provides a *knob* to control the amount of inaccuracy added to an execution. We demonstrate the effectiveness of the proposed techniques using a suite of five large graphs with varied characteristics and five popular graph algorithms.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; • **Mathematics of computing** → *Graph algorithms*; **Approximation**.

ACM Reference Format:
Somesh Singh and Rupesh Nasre. 2020. Graffix: Efficient Graph Processing with a Tinge of GPU-Specific Approximations. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3404397.3404406

> *Although this may seem a paradox, all exact science is dominated by the idea of approximation.*
>
> — Bertrand Russell

1 INTRODUCTION

Graph is a fundamental data structure to model a broad spectrum of real-world problems. Graph analytics pertains to various fields, such as bioinformatics, machine learning, social network analysis, and computer security among others. Graph analytic algorithms extract useful information from graphs by analyzing their structural properties and how information propagates through them; for instance, the effect of a drug and identifying communities. For scaling graph algorithms, former research has targeted parallelizing popular graph algorithms on multi-core CPUs [2, 27], many-core GPUs [4, 30, 32], as well as distributed and heterogeneous systems [5, 7, 9]. According to the TAO model, the primary technical challenge posed by graphs is due to inherent *irregularity* in the data-access, control-flow, and communication patterns [24]. This forces compilers to make pessimistic assumptions about them as the graphs are available only at runtime, leading to reduced parallelization benefits. The recent past has witnessed the emergence of very effective techniques to represent graphs compactly [3, 27, 28], to tame irregular computations [2, 23, 29, 31], and to map those to the underlying hardware [23, 30].

Our focus in this work is graph analytics on GPUs. The basic execution unit on GPUs is a *wavefront* or a *warp*, wherein threads execute in single-instruction-multiple-data (SIMD) fashion. For best performance, a GPU implementation must be tailored for efficient warp execution. It needs to be optimized along three important dimensions: memory coalescing [31], memory latency [21], and thread divergence [23]. Graph processing poses challenges for coalesced memory accesses due to unpredictable connectivity between graph vertices. A common strategy for improving coalescing is reordering of vertices. It is effective in improving the spatial locality of vertices by assigning consecutive ids to those that are likely to be accessed in tandem [2, 23]. Thus, the graph vertices could be pre-numbered based on the connectivity, so that neighbors of vertices being processed by warp-threads are nearby in GPU memory (typically, the vertices are numerically indexed). The second crucial dimension for efficient GPU execution is memory latency. Graph algorithms are often memory-bound due to the irregular memory access patterns and the resulting reduced cache benefits, making them more sensitive to memory latency. In the presence of hundreds of thousands of threads running on the GPU, per-thread cache benefits are further diminished. Therefore, literature has proposed various mechanisms such as kernel unrolling and usage of shared memory to reduce memory latency [12, 21]. Using shared memory requires identifying reusable attribute data (at the vertex or the edge) in the graph algorithm and taking advantage of the temporal locality. The third necessary dimension for efficient GPU execution is thread divergence. It occurs when warp-threads need to execute different instructions (or no-op) at the same time, resulting in loss of parallelism. thread divergence is rampant in graph algorithms due to arbitrary degree-distribution, leading to load-imbalance. For skewed degree distributions prevalent in several real-world graphs,

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404406

load-imbalance poses sequentiality bottleneck. Former research has proposed degree-sorting, nested kernels, loop-splitting, and edge-based processing to reduce thread divergence [2, 31].

While the proposed solutions in literature are effective in improving the overall execution, there is an inherent limit to their effectiveness. The proposed techniques are *exact*, that is, those (correctly) compute the same information as the original unoptimized program. Therefore, the optimizations cannot be applied beyond a point, limited by the graph structure. This poses a hindrance to efficient computation of time-consuming operations such as betweenness centrality computation, wherein exact parallel computation may take days for a billion-scale network.

Our goal in this work is to go beyond the traditional solutions and improve graph analytics on GPUs by allowing small approximation in the computation. For example, in network visualization tools such as *Gephi*, when employing a force-directed graph layout algorithm (e.g., ForceAtlas2), there is a trade-off between the quality of the simulation and the time for convergence. Similarly, we may *estimate* a set of k nodes with the largest betweenness centrality (BC) in a network faster without computing the exact BC values of the nodes [26]. With the growing importance of edge-computing and low-energy devices, it is paramount to approximate each of CPU-, memory- and communication-intensive processing. Indeed, there have been approximation techniques employed to speed-up graph algorithms [8, 26]. In a similar setting, we target graph applications that can tolerate some degradation in the result quality in exchange for faster execution. However, our target is *GPU-specific optimizations* which would collectively help general graph algorithms, rather than *a particular algorithm*. Thus, we pose the following questions:

- If we compute an approximate solution, can we improve coalescing, and reduce memory latency and thread divergence beyond what is possible for the exact solution? A logical answer to this question is in the affirmative, but it is a non-trivial task, as we illustrate in this paper.
- What would be the improvement due to approximations in terms of the execution time of the graph algorithm?
- What would be the *knob* that enables control of the amount of approximation added and its effect on the execution time?

In this work, we devise a new graph reordering strategy to enable coalesced accesses, a new method exploiting clustering coefficient to improve the usage of shared memory, and an edge-insertion based method to reduce thread divergence while improving convergence. We have designed all the techniques keeping in mind *propagation-based* graph algorithms, following the *vertex-centric* model of parallelization, wherein the threads operating on the assigned set of nodes propagate information to the nodes' neighbors along their incident edges.

Each of our proposed techniques involves preprocessing the input graph. The overhead incurred is expected to be amortized over multiple executions on the transformed graph. There are application scenarios that require input graphs to be processed multiple times, justifying the preprocessing of the graphs. For instance, computing a 2-approximate solution to the Steiner tree problem [13] (routinely used in network design and wiring layout) involves running SSSP from multiple terminal nodes to find the shortest path between every pair of terminal nodes. In the case of page-rank computation, the cost can be justified when more refined results are desired.

It is challenging to add approximations that are also GPU-friendly, to achieve a good speedup. One needs to balance between the precision-efficiency trade-off while designing an approximate method. We clarify that our proposal embodies approximations, but is not an approximation algorithm in the traditional sense.

This paper makes the following contributions:

- For time-consuming graph applications, we argue for approximate computation to go beyond the current limits of GPU-specific optimizations. Such approximations should be algorithm- and graph-oblivious to apply to a wide variety of graph analytic computations and graph structures.
- We propose `Graffix`, a framework for approximate computing techniques to improve coalescing, memory latency, and thread divergence of graph processing kernels on GPU. Each of the proposed techniques modifies the graph structure to accomplish the goal. Our techniques offer *tunable knobs* to control the amount of approximation injected.
- We illustrate the effectiveness of `Graffix` using a suite of five widely-used algorithms, namely, betweenness centrality computation (BC), minimum spanning tree computation (MST), page rank computation (PR), single-source shortest paths computation (SSSP), and finding strongly connected components (SCC). We observe that `Graffix` leads to improved execution times, trading off some accuracy. Our proposed techniques for improving coalescing, reducing memory latency and reducing thread divergence yield respective geomean speedups of 1.16×, 1.20× and 1.07× while maintaining geomean accuracies in the ballpark of 10%, 12.7% and 8.2% respectively. `Graffix` also improves the execution performance of the state-of-the-art graph frameworks Gunrock [30] and Tigr [23] in exchange for small inaccuracies. Thus, we show that our techniques do not compete with the existing GPU-specific optimizations, but complement those. They can be combined for improved benefits.

2 IMPROVING MEMORY COALESCING

A GPU-parallel algorithm exemplar: *betweenness centrality*. Consider the parallel Brandes' algorithm [25] for computing the vertex betweenness centrality in an unweighted graph, as shown in Algorithm 1. It as an exemplar of a general class of parallel algorithms on GPU. BC ranks the graph vertices according to the number of shortest paths that pass through them.

Brandes' algorithm is a two-pass procedure. The *forward* pass is a breadth-first traversal (BFS) which results into a BFS directed acyclic graph (DAG). This DAG is traversed *backward* in the second pass to accumulate the number of shortest paths passing via each vertex. The accumulation is performed using the notion of *dependency*. The *dependency* of a vertex v w.r.t. a given source vertex s is $\delta_s(v)$. It is computed using the following recurrence:

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

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 (computed using the forward pass at Line 3). The size of the *pred* list of a vertex is bounded by its degree. *pred* lists of all the

vertices together induce a DAG D over the graph G. BC of each vertex is computed as a summation over all the sources, as shown below (computed using the backward pass at Line 6):

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

Algorithm 1 BC Computation over graph $G(V, E)$

```
1:  bc[v] = 0        ∀v ∈ V                          ▷ initialization
2:  for all s ∈ V do
    ▷ Forward Pass: form BFS DAG, D
3:      for all v : Node ∈ G do                      ▷ In parallel
4:          compute σ_sv                             ▷ atomic add
5:          compute pred(s, v)
    ▷ Backward Pass: backward traverse DAG, D
6:      for all v : Node ∈ D do                      ▷ In parallel
7:          compute δ_s(v)
8:          bc[v] += δ_s(v)
9:      for all (u → v) ∈ E do                       ▷ Reset graph attributes
10:         reset(u → v)
```

We pursue the inner parallel strategy of parallelizing Brandes' algorithm, i.e., each of the computation steps (lines 3, 6 in Algorithm 1) is executed in parallel for a single source, and different sources are processed in sequence. In the forward pass, each thread enumerates a vertex's neighbors and updates σ_{sv}. On the GPU, due to multiple threads writing to the same vertex's σ, threads need to synchronize using an atomic instruction (such as `atomicAdd` from CUDA). On the memory access front, the memory access for σ in Algorithm 1 is generally uncoalesced due to the unpredictable node-connectivity. Reading (and writing) a node's neighbors' σ also suffers from low locality which causes significant memory latency and limits overall performance. Further, since warp-threads assigned to different vertices may process different numbers of neighbors, the forward pass incurs thread divergence. In the backward pass, processing δ attribute of a node's predecessors leads to reduced coalescing, low locality, and high thread divergence.

2.1 Coalescing in Graffix

Graffix makes the graph more *structured* to improve coalescing. To this end, we devise a modified graph layout by rearranging graph nodes, edges, and their associated information to make warp-threads access nearby memory locations with higher probability.

We use the Compressed Sparse Row (CSR) format to represent the graph, having an *offset* array, an *edges* array, and auxiliary arrays to store *edge attributes* and *node attributes*. Figure 1 shows an example directed graph and its CSR representation. In a vertex-based processing, a thread is assigned to a vertex. Hence, accesses to the offset array and the source vertex attribute array are coalesced. However, due to the *irregular* memory access pattern of the *node attributes* array resulting from the neighbor traversal of the nodes, the accesses to *node attributes* array are largely uncoalesced.

A key primitive in graph operations is *neighbor enumeration* wherein a warp, assigned to a set of vertices, iterates through their neighbors to propagate information. Such a neighbor enumeration is done in all the algorithms in our experimental setup. Graffix improves coalescing for this primitive. At a high level, our technique uses a careful combination of renumbering and replication to bring together in memory the data of those nodes that are likely to be accessed in tandem. Vertex renumbering helps bring connected

Algorithm 2 Graffix technique for improving memory coalescing

```
Input: Graph G(V, E)
Output: Graph G'(V', E')
1:  function TransformGraph(G)
    ▷ Step 1: Vertex renumbering
2:      v.level = ∞        ∀v ∈ G.V
3:      for Node s : G.V orderedby (decreasing node degree) do
4:          if s.level == ∞ then
5:              s.level = 0
6:              BFS(G, s)                            ▷ Assigns levels to nodes
7:      RenumberVertex(G, k)                         ▷ k is the chunk size
    ▷ Step 2: Node replication
8:      ReplicateVertex(G, k)
    ▷ The transformed graph is G'(V',E')
9:  end function

10: function RenumberVertex(G, k)
11:     gId = 0;
12:     for Node n : L_0 do      ▷ L_0 is list of nodes at the 0^th level in G's BFS forest
13:         n.id = gId++;
14:     for i = 0 .. numLevels-2 do                  ▷ numLevels is number of BFS levels
15:         gId = ⌈gId/k⌉ × k
16:         for j = 0 .. (max node degree in L_i) do  ▷ L_i is the list of nodes at level i
17:             for Node n : L_i do
18:                 if (n.degree > j) && (n.neighbors[j] ∈ L_{i+1}) then
19:                     n.neighbors[j].id = gId++
20: end function

21: function ReplicateVertex(G, k)
    ▷ Nodes array divided into chunks of size k, such that, chunkId[u] = u/k
22:     for Node n : G.V do                          ▷ n is a non-hole node
23:         count_n = [ ] ▷ hash table to count the number of edges from n to a chunk
24:         for Node v : n.neighbors do
25:             if v ∈ L_i && ∃u ∈ L_{i-1}, such that, u is a hole then
26:                 count_n[v.chunkId]++
27:         for curChkId : count_n.ChunkIds do ▷ ids of chunks having edges from n
28:             connectedness^n_curChkId = count_n[curChkId] / (# non-hole nodes with curChkId)
29:             if connectedness^n_curChkId ≥ threshold then
30:                 Duplicate n to get n'
31:                 n'.id = u.id, such that, curChkId in L_i && u ∈ L_{i-1}   ▷ Fill holes
32:                 for Node p : n.neighbors, such that, p.chunkId == curChkId do
33:                     Remove edge n → p
34:                     Add edge n' → p
35:                 Add edges n' → q    ∀q, such that, q.chunkId == curChkId;
                                        q is a 2-hop neighbor of n
36: end function
```

nodes and their data together. However, it has a limitation that a node occurs exactly once, and therefore it cannot be nearby all its neighbors (as their node ids could be far apart). This limitation is overcome with replicating the node, thereby allowing such a node to be nearby its neighbors. Graffix creates copies of a node, subject to a certain condition, and inserts the copies of these nodes, along with their edge-lists, in the vicinity of their neighbors in the CSR representation. Algorithm 2 presents the pseudocode of our technique. TransformGraph() is the driver routine. We explain the scheme in detail below.

2.2 Renumbering Scheme

Graffix renumbers vertices such that warp-threads are assigned nearby ids. While node renumbering is well-explored in the literature to improve thread divergence and locality [2, 10], it is ineffective when applied directly to improve coalescing. This is because the numbering assigns consecutive ids to a node's neighbors. This improves locality in serial processing, as the same thread would process all the neighbors. However, due to this, threads belonging to the same warp end-up processing vertices numbered far apart –

reducing coalescing. For instance, in Figure 1, assume the warp-size to be 4. The nodes 0–3 are assigned to threads having the same id as the node. With vertex centric processing, the warp-threads would access the attributes of the first neighbor of the respective nodes concurrently, and so on. The first neighbors are indicated by the `offset` array: 0, 7, 13, 16, to be indexed into the `edges` array. The warp threads would access the locations 4, 0, 11, and 19 in the node attributes array together. Further, assuming that the accesses to a chunk of 4 words can be coalesced, the accesses to the destination nodes' (4, 0, 11, 19) data in the *node attributes* array are not coalesced since each of these lies in a separate 4-word chunk. Hence, we propose a new numbering scheme for improving coalescing.

The numbering starts with a vertex having the highest outdegree and performs breadth-first traversal (BFS) on the graph, till all the nodes in the graph are visited, to obtain a BFS tree or a BFS forest if the graph is disconnected. In the graph is disconnected, the source nodes for the subsequent BFS traversals are picked in the decreasing order of outdegree among the unvisited nodes. The levels of the visited nodes are updated to a lower value, if possible, in the case of multiple BFS traversals. The loop at line 3 in Algorithm 2 accomplishes this. For example, in the graph G from Figure 1, vertex 0 has the highest outdegree. Performing BFS from vertex 0 on G visits vertices {0, 4, 5, 6, 7, 8, 13, 14, 15, 17}. The source for the next BFS is vertex 1 since it has the highest outdegree among the unvisited nodes. BFS from 1 covers vertices {1, 10, 12, 18} among the unvisited nodes. Further, among the already visited nodes, the levels of nodes 15 and 17 are reduced to 1. Next, applying BFS from node 2 covers vertices {2, 11, 19}, while BFS from 3, 9 and 16 cover vertices 3, 9 and 16, respectively. Thus, vertices 0, 1, 2, 3, 9 and 16 are at level zero, while all others are at level one.

An important observation is that the nodes at the same level in the BFS forest are going to be accessed by consecutive threads. Therefore, those are assigned ids incrementally in a round-robin fashion. Thus, the first neighbor of each of the parents from the previous level is assigned a new id followed by the renumbering of all the second-neighbors, and so on. For instance, in Figure 2b, which shows the renumbered graph, node 8 is the first unnumbered neighbor

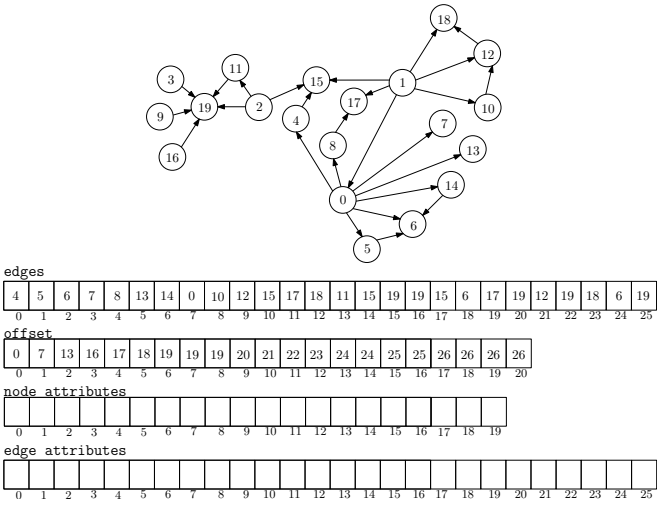

Figure 1: Original graph G and its CSR representation

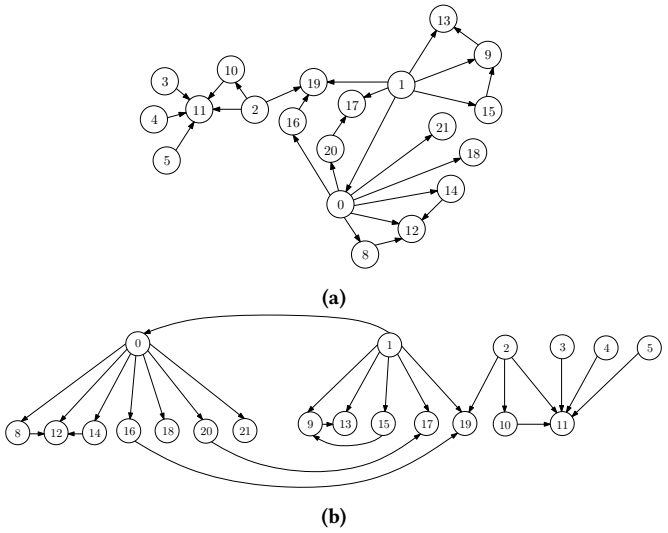

Figure 2: (a) Graph G from Figure 1 with renumbered nodes (b) The same graph reoriented for clarity

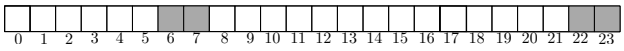

Figure 3: Holes in nodes after renumbering G

of node 1. A crucial aspect of `Graffix`'s numbering scheme is that the new node ids at each level of the BFS forest start from a multiple of k ($1 \leq k \leq$ warp-size), as shown in RenumberVertex() routine at line 10 of Algorithm 2. This is different from the prior numbering schemes and provides an opportunity for accesses to be coalesced *at every level*. For instance, Figure 2a is the renumbered graph with $k = 8$, and Figure 2b is its isomorph. With the new renumbering, vertices 0 through 5 are at BFS level zero. The next level starts with a multiple of k (= 8) greater than the last vertex id 5 (that is, there are no vertices with ids 6 and 7). Hence, the next level is occupied by vertices 8 through 21.

Creation of Holes. An important aftereffect of `Graffix` numbering is that since not all levels have the number of nodes in multiples of k, the renumbering scheme may create *holes* in the CSR representation arrays. For instance, the renumbering gives rise to *holes* in the nodes array at locations 6, 7, 22 and 23, as shown in Figure 3. The choice of k controls the number of *holes* at each level of the BFS forest. The number of holes at a level is $< k$. `Graffix` exploits these *holes* to enhance the degree of coalescing. It uses replication to copy specific nodes to these holes. The controllable node replication modifies the underlying graph and can introduce some approximation. Carefully identifying the nodes to replicate aids the underlying graph computation to reach its fixed-point faster.

2.3 Node Replication

The node replication to fill the holes needs to ensure that (i) it improves coalescing, leading to improved execution time, and (ii) the error is small. This is done as follows. Following the renumbering, the `nodes` array (which now also includes holes) is divided into chunks of size k, the same as that used for vertex renumbering. A

warp processes the nodes of a chunk. If a node is *well-connected* to a chunk, our goal is to replicate the node in the chunk containing the parents of the chunk's nodes (as obtained from the BFS forest). For instance, in Figure 2b, hole number 6 can become a replica of node 0, because the chunk containing nodes 8, 12, 14 as well as another chunk containing nodes 16, 18, 20, 21 are well-connected to node 0. This would improve coalescing when neighbors of hole 6 are enumerated (see Figure 4). Graffix achieves it as follows. From each of the non-hole nodes, we maintain a count of the outgoing edges to the chunks whose parent chunks have *holes*. Further, we define $\text{connectedness}^{\text{node}}_{\text{chunk}} \triangleq \left(\frac{\#\text{ edges to chunk from a node}}{\#\text{ non-hole nodes in chunk}}\right)$ for each such node–chunk pair. If the *connectedness* of a node to a chunk is higher than a threshold, the node is deemed to be well-connected to the chunk and thus we replicate the node (ReplicateVertex() routine at line 21 in Algorithm 2). Variable *threshold* is a tunable parameter and controls the amount of inaccuracy. When there are more candidate nodes eligible for replication to a chunk, than holes in that chunk, the nodes with higher *edge-count* are prioritized.

Since holes in the CSR representation arrays do not contribute to any 'useful' work done, it is instructive to reduce the unoccupied holes in the modified graph. A judicious choice of k and *threshold* is instrumental in increasing the occupancy of the holes. In our experiments, we use $k = 16$ and set the *threshold* to 0.6 and 0.4 for the scale-free graphs and the road networks, respectively. We introduce new edges from a node replica to the non-hole nodes of a chunk. If the node being replicated has an edge to a node in the chunk, we add edges from the replica to its 2-hop neighbors inside the chunk to which there is not already an edge.

Controlling the approximation. Adding edges in this manner expedites the propagation of information among nodes while ensuring that the node attributes read or written to in a coalesced fashion also contribute to some meaningful computation. The amount of inexactness is proportional to the number of new edges added in the graph. Thus, by controlling the number of newly added edges, Graffix can keep the inaccuracy in check. The addition of edges as above results in only a few additional edges per replica since we restrict the view to a contiguous chunk of size k in the nodes array at a time, while looking for the 2-hop neighbors of the node being replicated. Graffix ensures that the node to be replicated has a high degree. So, adding few extra edges adds only small inaccuracy.

For our example, we divide the *nodes* array (Figure 3) in the renumbered graph into chunks of size k (= 8). Assume that the threshold on *connectedness* for replication of a node is set to 0.6. In the renumbered graph in Figure 2, node 0 has 4 edges to the chunk 16..23 and the chunk has two holes. Hence, the $\text{connectedness}^{0}_{16..23} = \frac{4}{6} = 0.667$. Since the *connectedness* of 0 to the chunk is greater than the *threshold*, we replicate 0 in chunk 0..7. We assign the id 6 to the replica of 0 and distribute the existing edges of 0 between 0 and 6. We also add new edges from 6 to nodes 17 and 19, as these are the 2-hop neighbors of 0 in the chunk 16..23. This leads to the modified graph G' shown in Figure 4.

2.4 Confluence due to Replication

Due to controlled node replication, the underlying graph structure undergoes some changes. As aftermath, different node-copies in the modified graph may exhibit different attribute values at the end of a GPU kernel iteration. Since logically these copies represent the same node, these attribute values need to be *merged*. The merging or the confluence may be done after a certain number of iterations or at the end of the overall computation. To reduce inaccuracies, Graffix merges attribute values from the copies of the same node after every iteration. The merge operator itself could be defined in two ways: (i) algorithm-aware, and (ii) algorithm-agnostic. The former is likely to result in better accuracy but needs additional knowledge. Graffix uses the latter approach and applies a generic confluence operator which computes arithmetic mean of different values. One can easily redefine the merging. For instance, in Figure 4, at the beginning of the algorithm, the attributes of nodes 0 and 6 will be the same. After each iteration, we merge the attribute values of nodes 0 and 6 using the arithmetic mean.

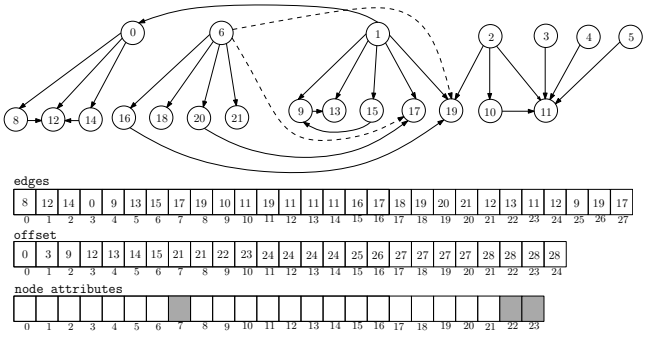

Figure 4: Modified graph G' with its CSR representation. Note that more edges got added compared to the original graph as well as there are holes in the CSR representation.

3 REDUCING MEMORY LATENCY

We seek to exploit the GPU memory hierarchy to reduce the time spent in fetching/updating data from/to global memory to curtail the execution time. Shared memory available per thread-block has been exploited in various ways in literature, and demands reuse of data items. For instance, in an unrolled kernel, the updated attribute values can be kept temporarily in shared memory. Alternatively, when a connected subgraph is processed by a thread, the stack or the queue can be stored in shared memory depending upon whether the subgraph traversal is depth-first or breadth-first [20].

Graffix proposes a new way of exploiting shared memory to process more-frequently-accessed nodes. Identifying such nodes at runtime adds inefficiency. On the other hand, identifying such nodes based on crude approximations such as degree is not very fruitful. Graffix exploits the graph property of *clustering coefficient* (CC) to identify such nodes. For the purpose of computing CC, we consider the graph to be undirected. The nodes having CC higher than a *threshold* are moved to shared memory, along with their neighbors. For instance, in Figure 5a, node N_1 has a high CC, so it can be moved to shared memory along with its neighbors. As nodes with a high CC are part of well-connected clusters, such clusters will be accessed frequently in iterative processing of the graph. Such high-CC nodes can be processed inside shared memory. Due to the power-law distribution, very few nodes have very

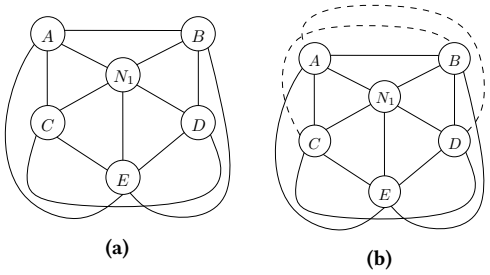

Figure 5: Reducing memory latency using shared memory

high CC leading to underutilization of shared memory. Adding approximation improves the applicability of the technique. `Graffix` selectively adds edges between nodes to effect the following:

(1) Increase the CC of the nodes having CC lower than, but close to, the threshold. This allows moving such nodes, along with the neighbors, from global memory to shared memory.
(2) Further boosting the CC of the nodes whose CC are already higher than the threshold.

In the first case, we add new edges preferentially between those neighbors of a high-CC node that have common neighbors. The purpose is to increase the CC of the node, and its neighbors, to make them candidates for being processed inside shared memory. In the second scenario, we add edges between those neighbors of a high-CC node that have the fewest edges with the other neighbors of that high-CC node. The rationale is to increase the connectivity among the neighbors of the high-CC node. `Graffix` looks at the connectivity only among the siblings of the high-CC nodes since these nodes will be in shared memory. We move the high-CC nodes to shared memory, along with their immediate neighbors alone. For instance, in Figure 5a, we add edges between the neighbors of N_1 having the fewest edges incident on them, that is, nodes `A, B, C, D`. For faster convergence, in both the scenarios above, the edges are added between the 2-hop neighbors. The resulting subgraph is shown in Figure 5b. Only a few edges are added in this manner. Additionally, we maintain a global limit for the number of edges added to the graph to contain the approximation. limits the inaccuracy. To enable reuse, the sub-graphs in shared memory are processed for a few iterations (say, t). We found that setting $t \sim (2 \times$ diameter of the subgraph) gave good performance benefits because of sufficient reuse. Thereafter, the attribute values of the nodes are pushed back to global memory.

Discussion. An alternative scheme for increasing the number and the size of the subgraph processed inside shared memory is to set a lower threshold on the clustering coefficient. However, this results in diminished benefits due to low reuse and impaired accuracy. Therefore, we recommended keeping the CC cut-off relatively high.

4 REDUCING THREAD DIVERGENCE

While degree-sorting [2] is an effective way to address thread divergence, it is often an overkill, since having *nearly-uniform* degrees *only within each warp* often suffices. `Graffix` combines bucket-sort and approximate computing to reduce thread divergence, as we explain below. As a preprocessing step, `Graffix` performs bucket sort on the `nodes array` using the node-degree as the key. This groups the nodes having *similar* degrees together. In each bucket,

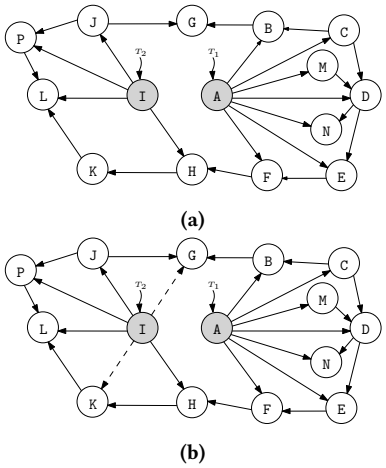

Figure 6: Handling divergence by graph transformation. Edges `IG` and `IK` are added for uniform node-degrees.

we assign nodes to warps in the order of their bucket positions. When node degrees are different, we carefully add a few edges to reduce thread divergence. Judicious addition of edges reduces the effect of approximations.

Adding edges. Additional edges are the source of approximation. Hence, among the warp-nodes, we add extra edges to only those that are deficient in their connectivity. If the difference of a node's degree to the warp's max-degree is lower than a *threshold*, we add edges to it to get its degree *close* to the warp-nodes' max-degree. This causes the warp-node degrees to become more uniform. The threshold dictates the number of edges added. As an extreme, it is possible to remove thread divergence fully with this technique.

Noting that most graph algorithms are propagation-based, we choose the destination nodes to be the 2-hop neighbors, leading to faster convergence. While the structural changes take care of node degrees, the choice of the edge-weights for the new edges (for weighted algorithms such as SSSP) is often fuzzy. In the case of weighted graphs, we set the weight of a new edge as the sum of the weights of the edge between the node and the 1-hop neighbor, and the edge between the 1-hop and the 2-hop neighbors. One can choose an alternative method to set up the edge-weights.

By adding edges in this manner, the warp threads which would otherwise be waiting for the longest-running thread to complete, perform some useful work in the meantime. The information propagated to their 2-hop neighbors is useful for the next iterations of the algorithm. Thus, the extra work done by the few warp threads per iteration contributes to the overall performance improvement.

Example. Consider the graph in Figure 6a. Suppose threads T_1 and T_2 belong to the same warp and are operating on nodes `A` and `I` respectively. Since the outdegree of node `A` (7) is more than that of `I` (4), T_1 has to process more edges than T_2. Assume that the threshold on the difference in node degrees for the purpose of adding edges is $\frac{maxdegree}{2}$. Also, assume that vertex `A` is the max-degree node in the warp. As the difference in the degrees of `I` and `A` is 3, which is less than $\frac{7}{2}$, our method adds new edges `IG` and `IK` to make the outdegree of node `I` close to the max-degree. The new outdegree

Graph	$\|V\|$ $\times 10^6$	$\|E\|$ $\times 10^6$	Graph type
rmat26	67.1	1073.7	R-MAT using GTgraph [16]
random26	67.1	1073.7	Random graph using GTgraph [16]
LiveJournal	4.8	68.9	Social network, small diameter
USA-road	23.9	57.7	Road network, large diameter
twitter	41.6	1468.3	Twitter graph 2010 snapshot

Table 1: Input graphs

Graph	Exact Time (sec)				
	SSSP	MST	SCC	PR	BC
rmat26	37	8996	21	12	15223
random26	29	10087	23	16	13127
LiveJournal	2	3424	7	1	1711
USA-road	152	82	12	1	2043
twitter	231	10943	37	18	21462

Table 2: Baseline-I: Execution time for exact versions

Graph	Exact Time (sec)		
	SSSP	PR	BC
rmat26	6	0.914	587
random26	4	1.180	498
LiveJournal	0.046	0.452	66
USA-road	12	0.130	38
twitter	17	3.000	827

Table 3: Baseline-II: Execution time for Tigr

Graph	Exact Time (sec)		
	SSSP	PR	BC
rmat26	19	1.070	872
random26	8	1.500	740
LiveJournal	0.142	0.530	98
USA-road	25.139	0.181	56
twitter	53	4.000	1227

Table 4: Baseline-III: Execution time for Gunrock

Technique	Graph	Preprocessing overhead	
		Time (sec)	Additional space
Improving coalescing	rmat26	76	9%
	random26	59	11%
	LiveJournal	8	6%
	USA-road	304	8%
	twitter	463	7%
Reducing latency	rmat26	155	5%
	random26	107	8%
	LiveJournal	21	5%
	USA-road	348	4%
	twitter	532	7%
Reducing thread divergence	rmat26	42	2%
	random26	46	3%
	LiveJournal	5	2%
	USA-road	38	1.5%
	twitter	157	4%

Table 5: Preprocessing overhead

of I is 6 (~85% of max-degree). Nodes G and K are 2-hop neighbors of I. Figure 6b shows the modified graph.

Effect of* Graffix *techniques on parallel BC. In Algorithm 1, Graffix's technique for memory coalescing brings closer nodes that are accessed in tandem by the warp-threads during the graph traversal on lines 3 and 6. The technique for memory latency ensures that the *well-connected* subgraphs are processed iteratively in shared memory. Further, the degrees of the warp-nodes during the traversal are normalized to curtail workload-imbalance.

5 EXPERIMENTAL EVALUATION

We evaluate the performance of our approximate techniques and compare it with the exact versions of the respective algorithms. We study five graph problems: SSSP, MST, SCC, PR and BC (Section 1). All these problems are popular in the community and, along with various graphs, their parallel algorithms stress-test our techniques.

Input Graphs. We select graphs with varying characteristics, shown in Table 1, to demonstrate the robustness of our approach. The graphs include an R-MAT graph and an Erdős-Rényi graph both having a billion edges, generated by GTgraph [16]; small-diameter LiveJournal and Twitter social networks; and a large-diameter real-world USA-road road network, all three from SNAP [14]. These graphs exhibit different behaviors for different techniques.

Baselines. We use three baselines to evaluate our techniques. First, we compare our approximate techniques with the exact implementation of SSSP, PR and BC available in Gunrock [30]. Second, we compare our approximate techniques with the exact implementation of SSSP, PR and BC available in Tigr [23]. Third, we compare our approximate SSSP and approximate MST with the respective exact versions from LonestarGPU [4], approximate SCC with the exact SCC by Devshatwar *et al.* [6], and approximate PR and BC with the parallel implementations of the exact PR computation and exact Brandes' algorithm respectively [28]. Singh and Nasre's work [28] is the closest to our work. They propose algorithm-agnostic approximate techniques for graph algorithms on GPUs. In contrast, Graffix exploits approximation for GPU-specific optimizations. We note that the average inaccuracy using their method is close to 20%. In contrast, Graffix incurs only half of its precision loss.

Machine Configuration. We perform experiments on a machine with an Intel Xeon 32-core E5-2650 v2 CPU having 100 GB RAM and Nvidia K40C GPU having 2880 cores spread across 15 SMXs with 12 GB memory. It runs CentOS 6.5. We use CUDA 8.0.

The execution times of the exact methods on the five graphs for the algorithms from the three baseline implementations are presented in Tables 2, 3 and 4. We report on the effect of approximations on the actual execution times of the algorithm implementation, which preclude file I/O and preprocessing steps, but include graph attribute initialization (such as vertex distances), initial CPU-GPU data transfer, and the main fixed-point loop repeatedly calling the primary kernel. We measure the inaccuracy incurred for each of the techniques by averaging the absolute difference between the attribute values of the vertices for the exact and the approximate versions. For SSSP, the attribute is the distance; for PR, it is the page rank; and for BC, it is the betweenness centrality. For SCC, we calculate the difference in the number of connected components, while for MST, we calculate the difference in the minimum spanning tree weights computed by exact and approximate methods. The code for Graffix is available at *https://github.com/ssomesh/Graffix*.

5.1 Preprocessing Overhead

The preprocessing overheads for the approximate techniques targeting memory coalescing, memory latency, and thread divergence are presented in Table 5. The mean times for transforming the graphs in our test-suite for improving coalescing, reducing memory latency and reducing thread divergence are 182s, 233s, and 58s respectively. This is a one-time offline cost. The execution of complex algorithms

	Graph	Speedup	Inaccuracy
SSSP	rmat26	1.22 ×	12%
	random26	1.13 ×	10%
	LiveJournal	1.18 ×	11%
	USA-road	1.15 ×	9%
	twitter	1.17 ×	12%
MST	rmat26	1.18 ×	13%
	random26	1.13 ×	15%
	LiveJournal	1.14 ×	12%
	USA-road	1.23 ×	11%
	twitter	1.17 ×	13%
SCC	rmat26	1.14 ×	8%
	random26	1.08 ×	14%
	LiveJournal	1.13 ×	7%
	USA-road	1.16 ×	11%
	twitter	1.15 ×	12%
PR	rmat26	1.20 ×	5%
	random26	1.15 ×	7%
	LiveJournal	1.21 ×	7%
	USA-road	1.19 ×	6%
	twitter	1.22 ×	7%
BC	rmat26	1.17 ×	9%
	random26	1.12 ×	13%
	livejournal	1.15 ×	10%
	USA-road	1.19 ×	12%
	twitter	1.14 ×	11%
Geomean		**1.16 ×**	**10%**

Table 6: Effect of memory coalescing

	Graph	Speedup	Inaccuracy
SSSP	rmat26	1.26 ×	12%
	random26	1.08 ×	17%
	LiveJournal	1.22 ×	13%
	USA-road	1.30 ×	13%
	twitter	1.18 ×	12%
MST	rmat26	1.22 ×	16%
	random26	1.10 ×	18%
	LiveJournal	1.18 ×	16%
	USA-road	1.20 ×	19%
	twitter	1.16 ×	15%
SCC	rmat26	1.20 ×	12%
	random26	1.10 ×	16%
	LiveJournal	1.22 ×	13%
	USA-road	1.20 ×	12%
	twitter	1.18 ×	13%
PR	rmat26	1.32 ×	7%
	random26	1.16 ×	11%
	LiveJournal	1.26 ×	7%
	USA-road	1.30 ×	5%
	twitter	1.22 ×	9%
BC	rmat26	1.24 ×	14%
	random26	1.13 ×	18%
	LiveJournal	1.21 ×	16%
	USA-road	1.26 ×	15%
	twitter	1.17 ×	13%
Geomean		**1.20 ×**	**13%**

Table 7: Effect of shared memory

	Graph	Speedup	Inaccuracy
SSSP	rmat26	1.06 ×	8%
	random26	1.03 ×	9%
	LiveJournal	1.07 ×	8%
	USA-road	1.12 ×	7%
	twitter	1.09 ×	6%
MST	rmat26	1.05 ×	10%
	random26	1.02 ×	11%
	LiveJournal	1.07 ×	8%
	USA-road	1.09 ×	10%
	twitter	1.05 ×	9%
SCC	rmat26	1.04 ×	9%
	random26	1.00 ×	7%
	LiveJournal	1.04 ×	6%
	USA-road	1.05 ×	9%
	twitter	1.06 ×	8%
PR	rmat26	1.10 ×	4%
	random26	1.04 ×	9%
	LiveJournal	1.08 ×	5%
	USA-road	1.06 ×	8%
	twitter	1.09 ×	8%
BC	rmat26	1.11 ×	11%
	random26	1.05 ×	14%
	livejournal	1.09 ×	9%
	USA-road	1.12 ×	7%
	twitter	1.06 ×	12%
Geomean		**1.07 ×**	**8%**

Table 8: Effect of thread divergence

Approximate `Graffix` versus exact Baseline-I

	Graph	Speedup	Inaccuracy
SSSP	rmat26	1.16 ×	12%
	random26	1.06 ×	10%
	LiveJournal	1.13 ×	11%
	USA-road	1.08 ×	9%
	twitter	1.12 ×	12%
PR	rmat26	1.14 ×	5%
	random26	1.08 ×	7%
	LiveJournal	1.15 ×	7%
	USA-road	1.12 ×	6%
	twitter	1.15 ×	7%
BC	rmat26	1.09 ×	9%
	random26	1.05 ×	13%
	livejournal	1.07 ×	10%
	USA-road	1.11 ×	12%
	twitter	1.06 ×	11%
Geomean		**1.10 ×**	**9%**

Table 9: Effect of memory coalescing

	Graph	Speedup	Inaccuracy
SSSP	rmat26	1.24 ×	12%
	random26	1.07 ×	17%
	LiveJournal	1.20 ×	13%
	USA-road	1.26 ×	13%
	twitter	1.15 ×	12%
PR	rmat26	1.30 ×	7%
	random26	1.14 ×	11%
	LiveJournal	1.26 ×	7%
	USA-road	1.28 ×	5%
	twitter	1.22 ×	9%
BC	rmat26	1.19 ×	14%
	random26	1.11 ×	18%
	LiveJournal	1.17 ×	16%
	USA-road	1.23 ×	15%
	twitter	1.16 ×	13%
Geomean		**1.19 ×**	**12%**

Table 10: Effect of shared memory

	Graph	Speedup	Inaccuracy
SSSP	rmat26	1.02 ×	8%
	random26	1.01 ×	9%
	LiveJournal	1.02 ×	8%
	USA-road	1.04 ×	7%
	twitter	1.03 ×	6%
PR	rmat26	1.06 ×	4%
	random26	1.02 ×	9%
	LiveJournal	1.04 ×	5%
	USA-road	1.03 ×	8%
	twitter	1.05 ×	8%
BC	rmat26	1.04 ×	11%
	random26	1.01 ×	14%
	livejournal	1.02 ×	9%
	USA-road	1.05 ×	7%
	twitter	1.03 ×	12%
Geomean		**1.03 ×**	**8%**

Table 11: Effect of thread divergence

Approximate `Graffix` versus exact Baseline-II

such as those for BC and MST consume more time than preprocessing. For the simpler algorithms such as those for SSSP, SCC, and PR, the preprocessing time is significantly higher. This extra preprocessing cost may be amortized over several runs of multiple algorithms. The corresponding mean extra space consumed by the transformed graphs (w.r.t. the original graph) is 8%, 5.6%, and 2.3% respectively for the three techniques, which is practically not high.

5.2 Effect of Coalescing

Table 6 shows the effect of `Graffix`'s technique for coalescing for five graphs on the five algorithms from Baseline-I. We report the results with threshold on *connectedness* set to the value which provides best results (which is different for different graphs). In particular, threshold of 0.6 performs well for power-law graphs and

	Graph	Speedup	Inaccuracy
SSSP	rmat26	1.20 ×	12%
	random26	1.1 ×	10%
	LiveJournal	1.17 ×	11%
	USA-road	1.12 ×	9%
	twitter	1.16 ×	12%
PR	rmat26	1.17 ×	5%
	random26	1.13 ×	7%
	LiveJournal	1.19 ×	7%
	USA-road	1.18 ×	6%
	twitter	1.20 ×	7%
BC	rmat26	1.11 ×	9%
	random26	1.07 ×	13%
	livejournal	1.09 ×	10%
	USA-road	1.16 ×	12%
	twitter	1.09 ×	11%
Geomean		**1.14 ×**	**9%**

Table 12: Effect of memory coalescing

	Graph	Speedup	Inaccuracy
SSSP	rmat26	1.22 ×	12%
	random26	1.06 ×	17%
	LiveJournal	1.23 ×	13%
	USA-road	1.28 ×	13%
	twitter	1.16 ×	12%
PR	rmat26	1.27 ×	7%
	random26	1.12 ×	11%
	LiveJournal	1.19 ×	7%
	USA-road	1.25 ×	5%
	twitter	1.17 ×	9%
BC	rmat26	1.21 ×	14%
	random26	1.13 ×	18%
	LiveJournal	1.19 ×	16%
	USA-road	1.24 ×	15%
	twitter	1.14 ×	13%
Geomean		**1.19 ×**	**12%**

Table 13: Effect of shared memory

	Graph	Speedup	Inaccuracy
SSSP	rmat26	1.07 ×	7%
	random26	1.03 ×	8%
	LiveJournal	1.06 ×	7%
	USA-road	1.08 ×	7%
	twitter	1.05 ×	6%
PR	rmat26	1.09 ×	5%
	random26	1.03 ×	6%
	LiveJournal	1.10 ×	5%
	USA-road	1.07 ×	8%
	twitter	1.08 ×	8%
BC	rmat26	1.06 ×	11%
	random26	1.04 ×	13%
	livejournal	1.08 ×	10%
	USA-road	1.10 ×	6%
	twitter	1.07 ×	12%
Geomean		**1.07 ×**	**8%**

Table 14: Effect of thread divergence

Approximate Graffix versus exact Baseline-III

of 0.4 for the road-network. We observe significant performance gains (mean 1.16×) for several algorithm-technique pairs, with some accuracy loss (mean 10%). Note that the accuracy loss is nearly half of that provided by the approximate versions of the baseline (as stated in the paper [28]). Tables 9 and 12 show the effect of our approximate techniques for coalescing for five graphs on the algorithms in Tigr and Gunrock, respectively. We observe that the speedups achieved over Gunrock are similar to Baseline-I. The speedups achieved over Tigr are lower since Tigr implements a memory access optimization, *edge-array coalescing*, to alleviate the irregularity in memory accesses. The inaccuracies for graph–algorithm pairs are similar across all baselines, because inaccuracy is tied to the modifications in the graph's structure.

Effect of Connectedness. *Connectedness* forms the tunable knob between speedup and inaccuracy. Figure 7 compares Graffix against Baseline-I, for various values of the threshold on the *connectedness* of a node to a chunk, for a fixed chunk-size of 16. For a small threshold, the speedup is low and the inaccuracy is high due to more replications. However, we observe a steady increase in the speedup with increase in the threshold up to a point (0.6 in the plot), followed by a gradual decline in the performance gains. This is because the number of nodes getting replicated is enough for the combined benefits of coalesced accesses to show effect. Also, the occupancy of the *holes* is high. However, upon further increasing the threshold (beyond 0.6), only a few nodes get replicated and the number of unoccupied *holes* is large. This results in reduced performance benefits for larger thresholds. The inaccuracy, on the other hand, gets benefited by increasing the threshold. This is due to fewer edges getting added due to a larger value of the threshold.

Guidelines for the Threshold. The threshold on *connectedness*, for a fixed chunk size, is based on the degree distribution. The power-law graphs have some high degree nodes. Majority of such nodes may be replicated if the threshold is low. To ensure that only the nodes with a high *connectedness* are replicated, in the interest of accuracy and the graph size, the threshold is set to a fairly large value. Setting a high threshold (above 0.6) would prohibit enough replication, which would hurt performance. In contrast, the node degrees in a road-network are small and largely uniform. For good hole occupancy, the threshold is chosen to be small (below 0.5).

5.3 Effect of Memory Latency

Table 7 shows the effect of using shared memory on the five algorithms for five graphs from Baseline-I. We observe performance gains for various algorithm-technique pairs. The threshold for clustering coefficient (CC) is set to a different value for each of the graphs for obtaining decent accuracy and speedup. Tables 10 and 13 show the effect of our approximate techniques for reducing memory latency for five graphs on the algorithms in Tigr and Gunrock, respectively. The speedups achieved over Gunrock and Tigr are similar (1.19×) to those achieved over Baseline-I. The inaccuracies for graph–algorithm pairs are similar (11%) across all baselines.

Effect of CC Threshold. Figure 8 plots the speedup and inaccuracy, w.r.t. Baseline-I, with varying thresholds for clustering coefficient. There is a consistent increase in speedup with increase in the threshold, since an increased threshold implies well-connected subgraphs occupying the shared memory, thereby benefiting from the its low memory latency. However, for threshold $\sim$ 1, fewer nodes are moved to shared memory, resulting in diminished gains.

As the threshold is increased, the inaccuracy first rises and later reduces. The rise in inaccuracy is because it exposes more nodes whose CC can be increased by addition of edges using the scheme presented in Section 3. However, after a point (threshold = 0.8), the inaccuracy reduces as the candidate nodes for processing inside shared memory have better connectivity; so we add fewer edges.

Guidelines for the Threshold. The choice of the threshold for CC is based on the graph's average CC and degree distribution. Since the focus is on finding nodes that are part of a well-connected cluster, the threshold must be set to a high value for all graphs.

5.4 Effect of Thread Divergence

Table 8 shows the effect of reducing thread divergence for five graphs on the five algorithms from Baseline-I. We obtain minor performance improvements for various algorithm-technique pairs,

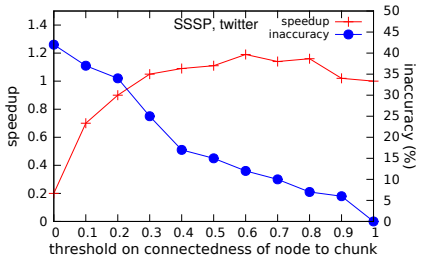

Figure 7: Effect of varying the threshold for node replication.

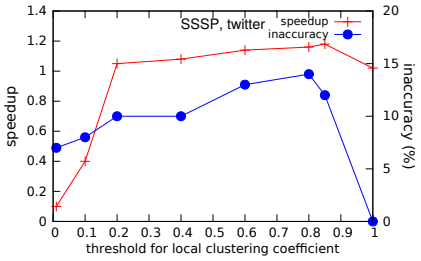

Figure 8: Effect of varying the threshold for clustering-coefficient

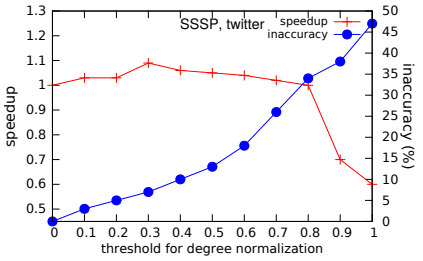

Figure 9: Effect of varying the threshold for degree normalization.

with reasonably high accuracy in most cases. Tables 11 and 14 show the effect of our approximate techniques for reducing thread divergence for five graphs on the algorithms in Tigr and Gunrock, respectively. We observe that the speedups achieved over Gunrock are similar to Baseline-I. Tigr already implements node splitting transformations for reducing thread divergence. Therefore, speedups achieved over Tigr are lower. The inaccuracies for graph–algorithm pairs are similar across all baselines.

Effect of Degree Similarity. To measure the variation in node degrees, we define:

$$\text{degreeSim}_{\text{node}} \triangleq \left(1 - \frac{\text{node degree}}{\text{maximum degree of warp nodes}}\right)$$

degreeSim identifies the deficit in the degree of a node compared to other nodes assigned to the same warp. Figure 9 plots the speedup and inaccuracy, w.r.t. Baseline-I, with varying thresholds for $\text{degreeSim}_{\text{node}}$. The node degree is made 85% of the warp's max-degree. As we increase the threshold, we allow more edges. We observe that the speedup increases with increase in threshold up to a point (0.3 in Figure 9) after which it begins to drop. This is because when limited edges are added, the performance improves due to the combined effect of reduced thread divergence and faster propagation. Performance gains drop with further increase in threshold. This is because the size of the graph increases due to addition of considerable number of edges, which begins to dominate. Inaccuracy increases monotonically with increase in threshold since a higher threshold allows for addition of more edges.

Guidelines for the Threshold. For obtaining reasonable accuracy and speedup, the threshold on *degreeSim* is set based on the degree distribution. If on an average, the mean node degree in a bucket is quite low, or if it is closer to the maximum node degree than to the minimum node degree in the bucket, then the threshold should be set to a low value (below 0.4). Picking the threshold this way ensures addition of limited extra edges as we normalize the degree of only relatively-large-degree nodes in a warp.

6 RELATED WORK

To the best of our knowledge, `Graffix` is the first technique that marries approximate computing with GPU-specific optimizations.

Parallel Graph Processing. Graph algorithms [24] have been shown to bear enough parallelism especially in the context of distributed [5, 9] and heterogeneous systems [5, 7]. Merrill et al. [18] propose work-efficient graph traversal on GPUs with several optimizations based on prefix-sum. Hong et al. [10] propose multiple methods for BFS on a heterogeneous system. Their hybrid method chooses the best execution among sequential, multi-core CPUs and single GPU. Nobari et al. [22] propose a GPU-based parallel Prim's algorithm for minimum spanning forest (MSF) computation. They concurrently expand several subsets of the computed MSF. Hong et al. [11] present a scalable algorithm for finding strongly connected components, tailored for small-world graphs. They exploit the data- and task-level parallelism in the FW-BW-Trim algorithm. McLaughlin and Bader [17] present an efficient parallel implementation for betweenness centrality computation on heterogeneous architectures. Their schemes provide performance improvements on diverse graphs. RADAR [2] combines data duplication and graph reordering to accelerate graph processing on multi-core systems. It uses *degree-sorting* to assign highly-connected *hub* vertices consecutive id's. It creates per-thread copy for the hub vertices to reduce false sharing and the cost of atomic updates. Reverse Cuthill-McKee (RCM) [15] is a reordering algorithm that uses a refinement of BFS. In contrast to `Graffix`, RCM performs level order traversal such that nodes at a level are visited in order of their BFS parent's placement in the previous level and in descending degree order for nodes with the same earliest BFS parent.

GPU-specific Optimizations. Zhang et al. [31] present techniques for removal of *dynamic irregularities* in GPU computation, to effect better memory coalescing. They use data reordering and job swapping, and runtime adaptation techniques for effective reduction in dynamic irregularities. Nasre et al. [21] discuss using shared memory for maintaining part of the worklist, and local worklist needed for kernel unrolling in the case of data-driven and topology-driven approaches respectively. Ashari et al. [1] present a representation of sparse matrices based on the compressed sparse row format to reduce thread divergence by combining rows into groups having a similar number of non-zero elements, which is well suited for graph processing applications. Gunrock [30] operates on frontiers of nodes or edges. A filtering operation removes inactive items from this frontier followed by application of user-defined functors to frontier in parallel. It also employs parallel graph traversal throughput optimization strategies. Nodehi Sabet et al. [23] propose to address the graph irregularity issues by transforming the graph to make it more structured. *Tigr* uses virtual split transformations and memory access optimization, called *edge-array coalescing* to the reduce thread divergence and to improve the data locality.

Approximation Techniques. Mittal [19] presents a survey of approximation techniques. Gubichev et al. [8] present a preprocessing-based technique for approximate SSSP computation. As precomputation, the shortest paths w.r.t few *landmark* nodes are computed for every node. The distance values of the query nodes w.r.t. a selected landmark node are combined to find the approximate distances. *Sampling* techniques are used extensively for approximate computation on graphs. Riondato and Upfal [26] propose a progressive sampling based family of algorithms to approximate the BC values.

7 CONCLUSION

We proposed graph transformation techniques for efficient graph processing on GPUs using approximate computing. Our techniques improve memory coalescing, memory latency and thread divergence by graph reordering and graph transformation. Using a suite of five popular graph algorithms and five large graphs, we illustrated that our proposed techniques reduce execution times of parallel implementations of graph algorithms appreciably. We believe that approximate methods bear the potential to go beyond the benefits of traditional optimizations, with the growing importance of edge-computing and low-energy devices.

REFERENCES

[1] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. 2014. Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 781–792.

[2] Vignesh Balaji and Brandon Lucia. 2019. Combining Data Duplication and Graph Reordering to Accelerate Parallel Graph Processing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 133–144.

[3] Maciej Besta, Simon Weber, Lukas Gianinazzi, Robert Gerstenberger, Andrey Ivanov, Yishai Oltchik, and Torsten Hoefler. 2019. Slim Graph: Practical Lossy Graph Compression for Approximate Graph Processing, Storage, and Analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*. ACM, New York, NY, USA, Article 35, 25 pages.

[4] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A Quantitative Study of Irregular Programs on GPUs. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, Washington, DC, USA, 141–151.

[5] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 752–768.

[6] Shrinivas Devshatwar, Madhur Amilkanthwar, and Rupesh Nasre. 2016. GPU Centric Extensions for Parallel Strongly Connected Components Computation. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit (GPGPU '16)*. ACM, New York, NY, USA, 2–11.

[7] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, New York, NY, USA, 345–354.

[8] Andrey Gubichev, Srikanta Bedathur, Stephan Seufert, and Gerhard Weikum. 2010. Fast and Accurate Estimation of Shortest Paths in Large Graphs. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM '10)*. ACM, New York, NY, USA, 499–508.

[9] Harshvardhan, B. West, A. Fidel, N. M. Amato, and L. Rauchwerger. 2015. A Hybrid Approach to Processing Big Data Graphs on Memory-Restricted Systems. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Hyderabad, India, 799–808.

[10] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. Association for Computing Machinery, New York, NY, USA, 267–276.

[11] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. 2013. On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-world Graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 92, 11 pages.

[12] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*. ACM, New York, NY, USA, 239–252.

[13] L. Kou, G. Markowsky, and L. Berman. 1981. A fast algorithm for Steiner trees. *Acta Informatica* 15, 2 (01 Jun 1981), 141–145.

[14] Jure Leskovec and Rok Sosič. 2014. SNAP: A general purpose network analysis and graph mining library in C++. http://snap.stanford.edu/snap.

[15] Wai-Hung Liu and Andrew H. Sherman. 1976. Comparative Analysis of the Cuthill–McKee and the Reverse Cuthill–McKee Ordering Algorithms for Sparse Matrices. *SIAM J. Numer. Anal.* 13, 2 (1976), 198–213.

[16] Kamesh Madduri and David A. Bader. 2006. GTgraph: A suite of synthetic random graph generators. http://www.cse.psu.edu/~madduri/software/GTgraph/. [Online; accessed May 28, 2013].

[17] Adam McLaughlin and David A. Bader. 2014. Scalable and High Performance Betweenness Centrality on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 572–583.

[18] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 117–128.

[19] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* 48, 4, Article 62 (March 2016), 33 pages.

[20] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*. IEEE Computer Society, Washington, DC, USA, 463–474.

[21] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Morph Algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 147–156.

[22] Sadegh Nobari, Thanh-Tung Cao, Panagiotis Karras, and Stéphane Bressan. 2012. Scalable Parallel Minimum Spanning Forest Computation. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 205–214.

[23] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 622–636.

[24] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 12–25.

[25] Dimitrios Prountzos and Keshav Pingali. 2013. Betweenness Centrality: Algorithms and Implementations. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 35–46.

[26] Matteo Riondato and Eli Upfal. 2016. ABRA: Approximating Betweenness Centrality in Static and Dynamic Graphs with Rademacher Averages. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, USA, 1145–1154.

[27] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2015. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*. IEEE, Washington, DC, USA, 403–412.

[28] Somesh Singh and Rupesh Nasre. 2018. Scalable and Performant Graph Processing on GPUs Using Approximate Computing. *IEEE Transactions on Multi-Scale Computing Systems* 4, 3 (2018), 190–203.

[29] Somesh Singh and Rupesh Nasre. 2019. Optimizing Graph Processing on GPUs Using Approximate Computing: Poster. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 395–396.

[30] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Trans. Parallel Comput.* 4, 1, Article 3 (Aug. 2017), 49 pages.

[31] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. ACM, New York, NY, USA, 369–380.

[32] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014), 1543–1552.