POSTER: Optimizing Graph Processing on GPUs using Approximate Computing

Somesh Singh
IIT Madras, *ssomesh@cse.iitm.ac.in*

Rupesh Nasre
IIT Madras, *rupesh@iitm.ac.in*

Abstract

Parallelizing graph algorithms on GPUs is challenging due to the irregular memory accesses and control-flow involved in graph traversals. In this work, we tame these challenges by injecting approximations. In particular, we improve memory coalescing by renumbering and replicating nodes, memory latency by adding edges among specific nodes brought into shared memory, and thread-divergence by normalizing degrees across nodes assigned to a warp. Using a suite of graphs with varied characteristics and five popular algorithms, we demonstrate the effectiveness of our proposed techniques. Our approximations for coalescing, memory latency and thread-divergence lead to mean speedups of 1.3×, 1.41× and 1.06× achieving accuracies of 83%, 78% and 84%, respectively.

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

1 Approximation Techniques

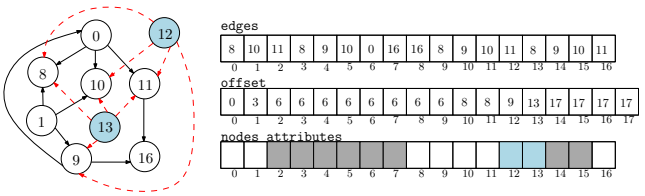
Memory Coalescing. Existing optimization techniques to improve coalescing [2] have limitations, as those proposals deal with *exact* computation. We break this barrier by allowing certain approximations in the computation.

The original input graph is represented in the Compressed Sparse Row (CSR) format (similar to the prior art [1]), having an *offset* array, an *edges* array, and zero or more auxiliary arrays to store *edge attributes* and *nodes attributes*, as applicable. A primary primitive in graph operations is *neighbor-enumeration*, wherein a warp assigned to a set of vertices iterates through their neighbors to propagate information. We focus on this primitive for improving coalescing.

At a high level, our technique of improving coalescing creates a modified layout of the graph by *renumbering* the graph nodes and *replicating* a select set of nodes and edges. It creates copies of the nodes, subject to a criterion, and inserts the copies of the selected nodes, along with their edge-lists,

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295736



(a) Modified graph **(b)** CSR representation

Figure 1. Example graph after pre-processing

in the vicinity of those nodes in the CSR representation arrays that would facilitate the coalesced accesses to the global memory. We introduce additional edges going out from these *replicated* nodes to make sure that the nodes read in a coalesced fashion are used effectively.

To improve neighbor-enumeration, we *renumber* the nodes such that the nodes to be accessed by the warp-threads are assigned nearby id's; this results in improved coalescing. We start with a node having the highest outdegree and perform breadth-first-search (BFS) traversals on the graph to obtain a BFS forest. The nodes at the same level in the BFS forest are assigned id's incrementally in a round-robin fashion — 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. Our technique ensures that the id's under the renumbering scheme start at a multiple of the warp-size (32) for each level in the BFS forest. Since not all levels will have number of nodes in multiples of the warp-size, the renumbering scheme creates *holes* in the CSR representation arrays. We exploit these holes to enhance the degree of coalescing by replicating *specific* nodes in these holes. The replication introduces some approximation.

The nodes chosen to be replicated to fill the holes are those having a good connectivity to the nodes in the chunk, of size 32, containing the holes. Node replication involves adding new edges from the new replicated node to all the non-hole nodes in that chunk. Note that the modified graph is independent of the algorithm and the above transformation can be performed as a pre-processing step at the time of graph-generation. This modified graph is fed as an input to the *exact* graph algorithm. An example modified graph is shown in Figure 1.

Memory Latency. Graph processing is often memory intensive. For the vertex-based processing of graphs, a node is *cheap* to process when its neighbors are cached. Our proposed technique increases the reuse of the subgraph brought into shared memory, and also increases the useful work done.

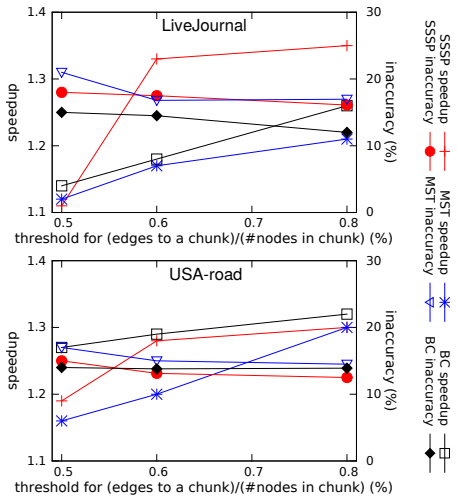


Figure 2. Effect of the coalescing technique

If the nodes brought into shared memory are connected, it ensures faster propagation of information. To this end, we arrange the nodes in decreasing order by their clustering coefficients (CC). We move the nodes having a large CC, and its neighbors, along with the other required information into the shared memory. Further, we add new edges among the neighbors of a large-CC node inside shared memory to improve propagation of information. In case of weighted graphs, the weight of the new edges introduced in shared memory is the average of the edge-weights of the edges in the original chunk. Each subgraph thus brought into shared memory is processed in an iterative manner, with results written back to the global memory. The next outer loop iteration processes another unprocessed high-CC node.

Clearly, higher the CC of the nodes brought into the shared memory, fewer are the extra edges added, and smaller is the approximation. Thus, to minimize the overall error in the final computed answer, we bring in the nodes, along with their neighbors, in the decreasing order of their CC values.

Thread-divergence. In vertex-based processing of graphs, thread-divergence is prevalent due to skewed degree distribution. We mitigate thread-divergence by making the node-degrees uniform within a warp. As a preprocessing step, we perform bucket sort on the *nodes array* using node-degree as the key. Each of the buckets is further sorted by node degree. We assign the nodes to the warps in the order of their position in the sorted array. If the node degrees assigned to a warp are different, we add a few extra edges or remove existing edges (leading to approximation). The extra edges are added to those nodes that are deficient in their connectivity. By noting that most graph algorithms are propagation-based, we choose the destination nodes of the newly added edges to be the 2-hop neighbors. In the case of weighted graphs, weight of a new edge, (a, c) is the sum of the edge-weights of the edges (a, b) and (b, c); a, b, c being nodes in the graph. For nodes having degree greater than the average degree in the warp, we remove some out-edges, picked uniformly at random. Structural change to the graph in this manner reduces thread-divergence, and may improve convergence.

Algo.	Technique	Speedup (w.r.t. exact parallel)	Inaccuracy
SSSP	Coalescing	1.35	21%
	Memory latency	1.40	24%
	Thread divergence	1.06	16%
MST	Coalescing	1.30	23%
	Memory latency	1.36	26%
	Thread divergence	1.03	15%
BC	Coalescing	1.32	17%
	Memory latency	1.42	24%
	Thread divergence	1.06	16%

Figure 3. Overall results

2 Experiments

Figure 2 presents the effect of the coalescing technique for two graphs – LiveJournal and USA road network, and three algorithms – single-source shortest paths computation (SSSP), minimum spanning tree computation (MST) and betweenness centrality computation (BC). In general, as we increase the threshold on the number of edges to a chunk from a node for node-replication, only the nodes having a high connectivity with a chunk get replicated in the chunk. This reduces the number of extra edges added within a chunk, resulting in improved accuracy. Also, increasing the threshold up to a point improves the speedup due to several holes in the chunks being occupied and the replicated nodes having a good connectivity with the chunk nodes. However, setting the threshold too high results in diminishing benefits.

Figure 3 presents the overall results. The speedup obtained in the case of approximations aimed at reducing memory latency is higher compared to the other two techniques. However, it also suffers from larger inaccuracy. The increased inaccuracy stems from the addition of extra edges to the sub-graphs moved to shared memory and from the repetitive processing of the modified sub-graphs inside the shared memory. The increase in the speedup is due to the reuse of the node- and edge-data brought into shared memory with high connectivity among the nodes within the sub-graph.

The performance benefit is low with a relatively high inaccuracy for approximation aimed at reducing thread-divergence. The low inaccuracy is attributed to the removal of edges from some of the nodes, which results in loss of information about the connectivity of the graph and also the edge attributes that get removed. We do get some benefits in execution time by reducing thread-divergence, with an overall relatively small inaccuracy.

References

[1] S. Singh and R. Nasre. 2018. Scalable and Performant Graph Processing on GPUs Using Approximate Computing. *TMSCS* 4, 3 (2018), 190–203.

[2] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. 2013. Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-coalesced Memory Accesses on GPU. In *PPoPP '13*. ACM, 57–68.