Fast Dynamic Graph Algorithms

Gaurav Malhotra, Hitish Chappidi, Rupesh Nasre

IIT Madras, India
{gaurav, hitish, rupesh}@cse.iitm.ac.in

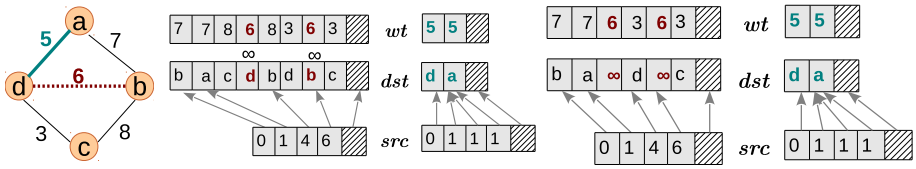
Abstract. We show that dynamic graph algorithms are amenable to parallelism on graphics processing units (GPUs). Evolving graphs such as social networks undergo structural updates, and analyzing such graphs with the existing static graph algorithms is inefficient. To deal with such dynamic graphs, we present techniques to (i) represent evolving graphs, (ii) amortize the processing cost over multiple updates, and (iii) optimize graph analytic algorithms for GPUs. We illustrate the effectiveness of our proposed mechanisms with three dynamic graph algorithms: dynamic breadth-first search, dynamic shortest paths computation and dynamic minimum spanning tree maintenance. In particular, we show that the dynamic processing is beneficial up to a certain percentage of updates beyond which a static algorithm is more efficient.

1 Introduction

Graphs are fundamental data structures to represent varied real-life phenomena such as interaction among molecules, friendships across persons, and city roads. Applications from various disciplines operate on these graphs to extract useful information such as placement of molecules, communities in social networks, and shortest path from one place to another. As data sizes grow, fast graph analytics rely on parallel graph processing. Former research has shown evidence that static graph algorithms contain enough parallelism to keep GPU cores busy [1,2,3,4]. However, several real-world graphs continue evolving. For instance, molecules change positions based on interaction and forces; new friendships get formed in social networks leading to new communities; while roads get blocked due to traffic management. A naïve way to deal with such dynamic updates is to rerun the static graph algorithm on each update to keep the information up-to-date. However, this is often time-consuming. Hard time-constraints in several applications (such as those dealing with streaming data or large-scale simulations) demand faster processing of dynamic updates, and quicker solutions.

In this work we deal with GPU-based dynamic graph algorithms. Following technical challenges get surfaced in such a setup, which fuel our work.

- **Graph representation:** Existing popular formats such as compressed sparse-row (CSR) storage for representing graphs are ill-suited for dynamic updates. A small change in the graph structure leads to a considerable data-movement in the CSR format. We devise a dynamic CSR storage format which continues its benefits for static graphs while allowing dynamic updates (Section 2).



(a) Graph G^0 (b) CSR and diff CSR of G^1 (c) CSR and diff CSR of MST^1

Fig. 1: Dynamic graph representation. Edge $b - d$ is deleted from G^0 and edge $a - d$ is added to get G^1. `src` stores index in the `dst` array.

- **Data-driven versus topology-driven processing:** It has been shown that a topology-driven processing *can* be beneficial for static graph algorithms [5]. However, a limitation of the existing topology-driven methods (wherein all graph elements are processed in each iteration) is their work-inefficiency. Especially in the context of dynamic updates, a topology-driven processing leads to wasted resources. We propose a data-driven approach to deal with dynamic updates. Unlike traditional static data-driven graph algorithms which use a single worklist, our dynamic graph algorithms necessitate use of multiple worklists for efficient data-driven processing (Section 3).
- **Synchronization:** Synchronization is a major challenge in graph algorithms in case of highly concurrent setting such as GPUs. This issue is exacerbated in the dynamic setting when the underlying graph structure is arbitrarily changed by threads. We propose a combination of lock-free updates and barrier-based processing to tame the synchronization cost (Section 4).
- **Interplay of incremental and decremental updates:** Fully dynamic graph algorithms may insert and remove graph elements that may affect their concurrent processing. For instance, in maintaining the minimum spanning tree, a higher-weight spanning tree edge may be deleted while a lower-weight new edge may be added. Taking advantage of such scenarios is critical for efficient dynamic processing of graphs.

We address the above challenges with effective mechanisms and illustrate that it is indeed beneficial to process dynamic graphs on GPUs. We apply the proposed techniques to three graph theoretic algorithms: dynamic breadth-first search (BFS), dynamic single-source shortest paths (SSSP) computation and dynamic maintenance of the minimum spanning tree (MST). Using several graphs from SNAP [6], we show that our GPU-based dynamic graph algorithms are faster than their static counterparts up to 15–20% of updates.

2 Graph Representation

An evolving graph is denoted using its version number as a sequence $G^0, G^1, ...$ Node addition and deletion can be simulated by edge addition and deletion. Similarly, edge updates can be simulated using deletion (of the edge) followed by addition. Therefore, we focus on edge addition and deletion in our setup.

CSR: A traditionally popular representation for GPU-based graph processing is compressed sparse-row (CSR) storage format [4]. It essentially concatenates the adjacency lists of nodes and uses offsets to mark the beginnings of adjacency lists. A sample graph and its CSR representation is depicted in Figure 1.

Unfortunately, traditional CSR has limitations when applied to dynamic graph algorithms as in our case. First, for adding a new edge $u \rightarrow v$, we need to append v to u's adjacency list (two updates are required for an undirected edge). However, this necessitates moving the adjacency lists of all the nodes after u. Additionally, this changes the offsets of all the nodes after u, requiring updates to *src*, *dst* as well as *wt* array if applicable (see Figure 1). Second, adding multiple edges in parallel by multiple threads requires heavy synchronization to move elements in the CSR arrays. Third, CSR does not allow us to take advantage of interacting edge additions and removals.

diff-CSR: We address these issues (and retain benefits of CSR) by augmenting CSR representation with a diff-CSR. Thus, the initial graph G_0 is represented using CSR format. The next graph version G_1 is formed by following the transformations to the original CSR array by processing deleted edges followed by the newly inserted edges. The deleted edges are marked by overwriting with a sentinel ∞ in *dst* array. For inserting an edge $u - v$, our method checks if any deleted edge exists for each source node (u and v), and if it does, it is replaced with the new destination in the original CSR array. If such a deleted entry is not found, then the new edge is pushed into an additional diff-CSR array which contains only the additional insertions which could not be fitted into the original CSR array. These two arrays together represent the graph G_1.

For the subsequent versions G_i we reuse the original CSR array of G_{i-1} and allocate a new diff array if required. Such a representation can form a chain of diffs over the original array. In our setup, we merge the two diffs to create a single consolidated diff array. This allows us to remove the sequential bottleneck of chain-traversal while processing a graph update. For deletion of edges in the graph, the original and the diff arrays of G_{i-1} are checked for the corresponding node, and it is marked as deleted. Insertions are tried to be fitted into one of these arrays. Additional insertions are copied into a bigger diff array by copying all the edges already present in the previous array to the new diff array. As the diff arrays are small, such a copying does not incur any noticeable overhead.

MST Representation: In the case of dynamic MST computation, the MST itself needs to be maintained. We take advantage of our diff-CSR to store the MST. For instance, the MST of G^0 in Figure 1a consists of edges $\{a - b, b - d, d - c\}$ with the MST cost of 16 (note that edge $a - d$ is not present in G^0). After edge $a - d$ is added and edge $b - d$ is deleted to get version G^1 (Figure 1b), the modified MST denoted as MST^1 consists of edges $\{a - b, a - d, d - c\}$ with cost 15. MST^1 can be stored in diff-CSR format as shown in Figure 1c.

3 Data-driven Processing

GPUs are ideal for regular dense matrix computations. A topology-driven processing is useful in such scenarios as most of the elements are almost always *active*; that is, work is required to be done at these elements. On the other hand, in a data-driven approach, only the active graph elements are processed, making better use of the available resources. The down-side of a data-driven approach is that the active graph elements need to be stored explicitly and maintained during the parallel processing. This demands maintenance of a concurrent worklist in the presence of thousands of threads on the GPU. Thus, there is a tension between work-efficiency and synchronization cost in topology-driven versus data-driven graph processing. In the context of dynamic graph algorithms, we advocate the use of a data-driven approach. This stems from the fact that typically the updates are sparse; that is, there are only a few updates.

3.1 Incremental Graph Processing

In an incremental setting, edges are only added to the graph and no edge-deletion is performed. Given a graph G^0 and a statically computed information I^0, the goal here is to incrementally compute the modified information I^1 for the superset graph G^1. For BFS, I^1 is the updated level information; for SSSP, it is the updated shortest path from a designated source; while for MST, it is the modified minimum spanning tree. An efficient implementation of such incremental algorithms would exploit the following properties:

- BFS: In I^1, no vertex has a level larger than its level in I_0. In other words, level numbers can only reduce across incremental updates.
- SSSP: In I^1, no vertex has a distance longer than its distance in I_0.
- MST: The MST cost cannot increase due to incremental updates.

BFS: In incremental BFS, when a new edge $u \rightarrow v$ is added, the levels of only those nodes that are reachable via edge $u \rightarrow v$ *may* reduce. Such a processing can be readily modeled by mapping incremental work in the context of static graph processing. Thus, vertex u can be added to the worklist and concurrent level-synchronous BFS can proceed, exactly as in the static version. If v's level reduces, the processing may continue to the next level; otherwise, no other vertices need to be processed further. Figure 2 shows the processing steps when edge $a - d$ is added and a is the source in the graph of Figure 1a.

SSSP: Similar to incremental BFS, in incremental SSSP also, when an edge $u \rightarrow v$ is added, the distances of only those vertices that are reachable via edge $u \rightarrow v$ may reduce. Therefore, the processing can also piggyback on the static processing by inserting vertex u into the worklist of active vertices. An important difference with respect to BFS is that SSSP computation is asynchronous, enforcing a different synchronization requirement (discussed in Section 4).

MST: Unlike BFS and SSSP, incremental MST requires special consideration,

Iteration	Vertex levels				Active vertices
	a	b	c	d	
0	0	7	15	13	{a}
1	0	7	15	**5**	{d}
2	0	7	**8**	5	{c}
2	0	7	8	5	{}

Fig. 2: Incremental BFS on the graph from Figure 1a

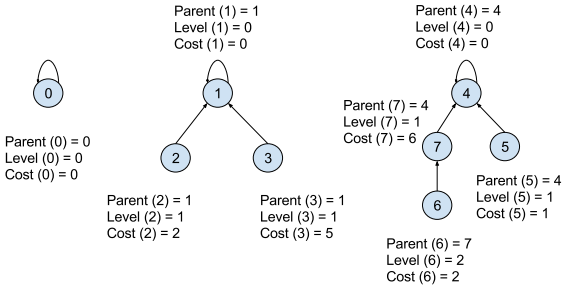


Fig. 3: Level-order traversal of the minimum spanning forest

which we discuss next. In MST, addition of an edge creates a cycle. Our implementation allows such a cycle to be created, and later traverses the cycle to remove the maximum weight edge (which could as well be the newly added edge). However, a naïve traversal to find the cycle is expensive. Therefore, we resort to preprocessing of the MST (in general, minimum spanning forest MSF) which reduces the traversal and improves incremental MST computation. The preprocessing ensures that there is a dedicated directed path (note that the MST is undirected) to traverse to find that cycle. In this step, we find out the representative vertices for each disjoint component of the MSF and start a level-order traversal from each vertex in parallel. During the traversal, we maintain three attributes with each vertex: *Parent*, *Level* and *Cost*. Whenever an unvisited vertex v is reached from the current vertex u, Parent(v) = u, Level(v) = Level(u) + 1, Cost(v) = Wt(u — v). This data-driven worklist-based process continues until all the vertices are visited as shown in Figure 3.

We perform incremental updates as below. We first add the incremental edges that connect two different MSTs in the MSF. This is because such cut-edges would always be part of the new MSF and are guaranteed not to create cycle. After all such inter-component edges are processed, any new edge would be intra-component and would necessarily create a cycle. This necessitates finding the maximum-weight cycle edge. To do this, we start from the two ends of the incremental edge; say, $first\_end$ and $other\_end$. Without loss of generality, assume that Level($first\_end$) $\geq$ Level($other\_end$). So, we start from the $first\_end$ and then following the vertices from the parent array, we update the

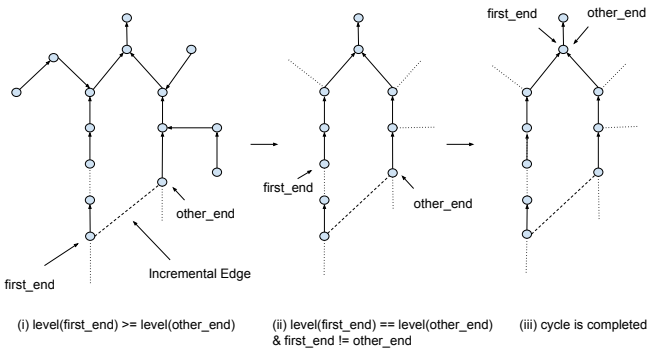


Fig. 4: Path traversal to find maximum weight edge in a cycle

parent of the $first\_end$ to be the new $first\_end$. We continue with this until we get to a point where Level($first\_end$) == Level($other\_end$). At this point, also if $first\_end == other\_end$, then we have found the cycle and we return with the maximum weight edge found on the path traversed. This is depicted in Figure 4.

When we add multiple incremental edges in parallel, there exists a chance that two incremental edges pick up the *same* maximum weight edge during its cycle traversal. In such cases, we must consider only one of the two incremental edges in that iteration and process the other edge later. Otherwise, there is a danger of forming a cycle in the maintained spanning tree.

3.2 Decremental Graph Processing

In a decremental setting, edges are only removed from the graph and no edge-insertion is performed. Given a graph G^0 and a statically computed information I^0, the goal here is to decrementally compute the modified information I^1 for the subset graph G^1. For BFS, I^1 is the updated level information; for SSSP, it is the updated shortest path from a designated source; while for MST, it is the modified minimum spanning tree. An efficient implementation of such decremental algorithms would exploit the following properties:

- BFS: In I^1, no vertex has a level smaller than its level in I_0. In other words, level numbers can only increase across decremental updates.
- SSSP: In I^1, no vertex has a distance shorter than its distance in I_0.
- MST: The MST cost cannot reduce due to decremental updates.

BFS: Unlike in incremental BFS, the decremental version poses a challenge that the next shortest path (new parent) is not known. To address this issue, we use a multi-worklist approach, which we explain below. The static BFS processes G^0 in level-synchronous worklist-based manner and computes the level values in I^0.

Next, to compute I^1, the dynamic version processes all the deleted edges $a \rightarrow b$ to check if a is a parent of b in the BFS tree (BFST). If it is, then we check all the incoming nodes of b (to check for the new shortest path) and update b's level. If its level *increases* then it is added to a special worklist. Processing threads remove nodes from this worklist and check if any of their outgoing vertices is a child in BFST to the currently removed node. If a vertex is indeed a child, it is pushed to the special worklist. For each vertex in the special worklist, we update its level based on its incoming nodes, whereas the regular worklist is used to propagate distances to the outgoing neighbors. If a vertex's level increases, we mark it special. The change is propagated to the special nodes' children by checking their parents in the BFST as discussed.

SSSP: Similar to decremental BFS, decremental SSSP also uses a multi-worklist approach. A key difference from BFS is that SSSP computation is asynchronous, enforcing a different synchronization requirement (discussed in Section 4).

MST: Similar to BFS and SSSP, decremental MST also uses a data-driven worklist-based approach, but its operators are quite different [7]. We first mark and delete all the decremental edges in the previous version of the MST. This is done by a level-order traversal of the MST where we start from representative vertices of the MSTs in the MSF and vertices from which edges are being deleted. After deleting the decremental edges, we will have different MST components.

For every such MST segment, we will have one representative vertex. We need to connect MSTs (if possible) by using other non-tree edges from the original graph. Similar to the processing in the incremental setting, we first add the cut edges across different representative vertices from the non-tree edges of the graph. Threads operate on representative vertices and start finding the minimum weight edge representative vertex adjacent to them. Since each representative vertex chooses at most one representative vertex, cycles may be introduced while adding edges between them. We borrow the technique of choosing the lower of the vertex identifiers for same-weight edges from Vineet et al. [8]. This ensures that only 2-length cycles are possible as illustrated in Figure 5. Threads can quickly check for 2-length cycles by finding if a reverse-edge is chosen by the end-point of the chosen-edge. After this step, minimum weight edges from the remaining vertices are added to the MST. Finally, all the vertices connected by minimum weight edges form one super-vertex component and all these disjoint components form new vertices for the next iteration. This process continues until no more edges can be added between any two components.

3.3 Fully Dynamic Graph Processing

BFS: Fully-dynamic BFS can be built upon the concepts of incremental and decremental BFS. For each deleted edge $u \rightarrow v$ in parallel, we push v to the special worklist, while for each inserted edge $u \rightarrow v$, we push u to the regular worklist. Threads then extract vertices from the special worklist, and process their incoming edges. A vertex is marked as special if there is an increase in its level compared to the previous version. Then, similar to the decremental

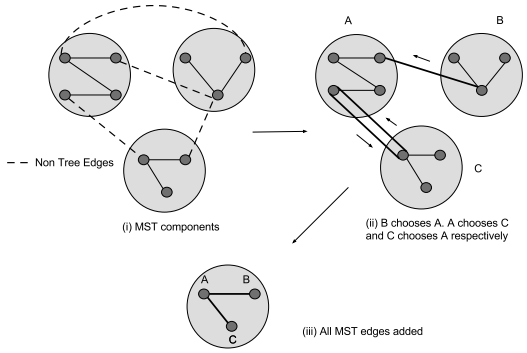
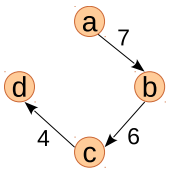
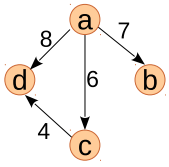


Fig. 5: Adding non-tree edges to get final MST





Iter	Vertex Distances				inWL	specWL
	a	b	c	d		
0	0	7	13	17	{a}	{c}
1	0	7	**6**	17	{a, c}	{}
2	0	7	6	**8**	{d}	{}
3	0	7	6	8	{}	{}

Fig. 6: Dynamic SSSP on graph versions G^0 and G^1

algorithm, the special node is pushed into the regular worklist for propagating the level information to its children. Threads extract vertices from the worklist and check if any of them is special. For each special vertex, similar to the decremental algorithm, we check if it has any child in the BFST in its outgoing edges. If there is, the child is pushed into the special worklist. For the remaining children, we propagate the (special) parent's change along its outgoing edges. If an outgoing neighbor's level reduces, it is pushed to the regular worklist. This repeats until both the worklists are empty.

SSSP: The incremental and the decremental algorithms can be combined and this can be used when both insertions and deletions are happening simultaneously. The algorithm is listed in Algorithm 1 which uses the results of iteration $i-1$ in iteration i. For each deleted edge (u, v) where u is the SPT parent of v, we push v to the special worklist. We also check if the vertex has a new outgoing edge (due to incremental updates). If it does, we push it into the (regular) worklist. We then process vertices from the special worklist, we relax all the incoming edges of each node and update the node's distance. We then check if the node's distance increased. If so, then the node is marked as special and pushed into the worklist. We loop through the worklist for special vertices. For special vertices, similar to the case of the decremental algorithm, we check if it is a parent to any node in the SPT in its outgoing edges. If so, then the child is pushed to the special worklist. For the others, we propagate the change along the outgoing edges. The two worklists are processed until they get empty.

Algorithm 1 Fully Dynamic SSSP

```
1  function SSSP-static(G^0)
2      for each iteration G^i other than G^0
3          readGraph(G^i)
4          copyResult(dist_{i-1}, dist_i)
5          preprocess(G_i)
6          while(! both worklists are empty)
7              for each vertex a in specWL
8                  specFunc(a)
9              for each node a in inWL
10                 regularFunc(a)
11             swap(inWL,outWL)
12
13 function preprocess(G^i)
14     for each vertex a in G^i
15         if newoutgoing[a] = true
16             push a to inWL
17     for each deleted edge a → b
18         if a is the parent of b in SPT
19             push b to specWL
20
21 function regularFunc(a)
22     if(a is special)
23         for each outgoing neighbor b of a
24             if a is the parent of b in SPT
25                 push b to specWL
26     else
27       for each outgoing neighbor b of a
28             if dist[a] + wt(a,b) < dist[b]
29                 dist[b] := wt(a,b) + dist[a]
30                 push b to outWL
31
32 function specFunc(a)
33     prevdist := dist[a]
34     dist[a] := ∞
35     for each parent b of a
36         update dist[a] via b
37         if dist[a] > prevdist
38             push a to outWL
39             mark a as special
40         else if dist[a] < prevdist
41             push a to outWL
```

Consider the example shown in Figure 6. Edge $b \rightarrow c$ is deleted and edges $a \rightarrow d$ and $a \rightarrow c$ are added. Vertex c is a child of vertex b in SPT and we push c to the special worklist. Then vertex a is pushed into the regular worklist because it has new outgoing edges. We extract vertex c from the special worklist, process its incoming vertices, and update its distance to 6. Then we push c to the worklist because of reduced distance. Then we process all the nodes in the worklist. This reduces d's distance to 8 and it is pushed to the worklist. This finally leads to the fixed-point.

MST: In contrast to BFS and SSSP, for dynamic MST, we first follow the steps of the incremental algorithm. However, instead of picking only the maximum weighted edges from either of the ends, if we get a decremental edge having weight greater than the incremental edge during the cycle traversal, then we replace the incremental edge with the decremental edge. This mechanism is valid because we are replacing a valid incremental edge whose weight is less than the decremental edge, if found. However, if we do not find any decremental edges or any decremental edges whose weight is less than the incremental edges, then we continue with our above proposed incremental algorithm. After processing all the incremental edges in this fashion, we process the remaining decremental edges which have to be deleted.

4 Synchronization Considerations

Irregular procedures such as graph algorithms necessitate thread-synchronization for safe concurrent processing. The synchronization requirement is heavy when the graph undergoes structural updates, as in our case. Thus, for instance, while a thread is updating an MST to include a new edge, some other thread may be deleting the same edge. The synchronization issue gets exacerbated on GPUs as logical locks are prohibitively expensive.

BFS: Level-by-level static BFS can be implemented without using atomic instructions as the data-races are benign. However, synchronization may still be necessary while maintaining the frontier, depending upon its implementation. Thus, if the frontier is implemented as a bit-vector (one bit per vertex), then no explicit atomics are necessary (as single word writing is atomic in CUDA and most other hardware). However, if the frontier is implemented as a compact worklist containing vertex identifiers, then synchronization in terms of either atomics or prefix-sum barriers is necessary to insert vertices. Removal of elements need not require synchronization as all the elements can be read in parallel and the worklist can be emptied by setting its size variable to 0. However, a barrier is necessary between reading and resetting the worklist. All these synchronization requirements are applicable in case of the dynamic setting also. In addition, efficient processing of special and regular worklists demands careful synchronization. In particular, we need to insert two barriers between the iterative processing of special and regular worklists (after for loops at lines 7 and 9 in BFS code similar to Algorithm 1).

SSSP: Synchronization considerations of dynamic SSSP are similar to those of BFS. However, since SSSP is implemented in an asynchronous manner, it demands usage of atomicMin instruction while updating distances.

MST: Dynamic MST poses more synchronization challenges. First, all primitive data type updates rely on atomic instructions. Performing level-order traversal to find maximum weight edge in a cycle needs BFS-like synchronization. The underlying data structure to keep track of MSTs also necessitates careful synchronization. For instance, various components of the minimum spanning forest (MSF) are efficiently stored in a concurrent union-find data structure. Updating parent pointers of vertices (union) and identifying if two vertices belong to the same component (find) need to be separated either by a barrier or protected using atomics. Note that find is not a read-only operation when path-compression is enabled. Further, if the incremental and the decremental phases are separated by a barrier, it helps reduce intra-phase synchronization. Note, however, that the fully-dynamic version takes advantage of the decremental edges while inserting new edges for efficiency (as discussed in Section 3.3).

5 Experimental Results

We implemented dynamic BFS, SSSP and MST in CUDA. The experiments are run on an Intel Xeon X5675 with Tesla M2070 GPU with 6 GB RAM and 14 SMs containing 448 cores. We compare our dynamic BFS and SSSP with the static versions from LonestarGPU 2.0 [9], and dynamic MST with our implementation of the static version by Vineet et al. [8]. We call the static versions as *Base*. Our code is publicly available[1] which contains optimized incremental, decremental as well as fully-dynamic versions.

In the evaluation below, the base implementation creates the graph with all the dynamic updates and then runs the static version on it. We present results directly for fully-dynamic version. We select an edge for addition or removal by selecting two random vertex identifiers, and checking if the edge already exists. We add edge-weight as a random number between 1 and 100. Figure 7 shows the characteristics of various graphs from SNAP [6].

5.1 Performance

diff-CSR: To evaluate our diff-CSR representation, we added and removed randomly selected edges to various graphs. Figure 8 shows throughput (number of updates performed per second) for various number of updates. We find that (i) the throughput improves almost linearly, (ii) the throughput reaches a plateau after 10 million updates, and (iii) is largely the same independent of the graph.

[1] `http://www.cse.iitm.ac.in/~rupesh/?mode=Research`

Graph	$\|V\| \times 10^6$	$\|E\| \times 10^6$	Graph	$\|V\| \times 10^6$	$\|E\| \times 10^6$
Livejournal	3.9	70.0	R4-2e20	1.0	8.3
R4-2e23	8.3	67.0	Amazon2008	0.7	7.0
Soc-Pokec	1.6	61.0	Wiki	2.4	5.0
Patent	6.0	33.0	Amazon0505	0.4	4.8
Flickr	0.4	17.0	Road-CA	2.0	3.0
Rmat20	1.0	16.5	Youtube	1.2	3.0
Skitter	1.6	11.0	Road-TX	1.4	2.0

Fig. 7: Input graphs

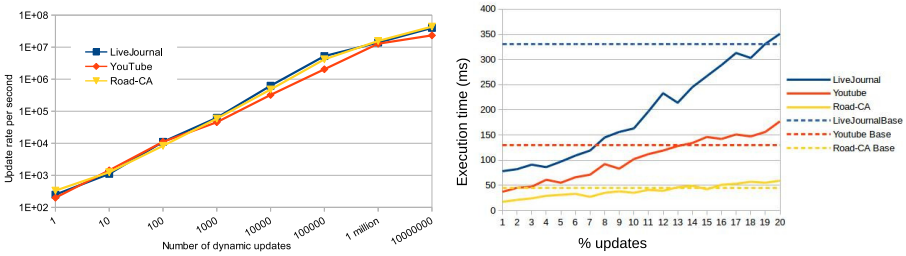


Fig. 8: diff-CSR throughput and performance of fully-dynamic BFS

BFS: Figure 8 shows the performance of fully-dynamic BFS. It plots the execution time with varying number of dynamic updates (as a percentage of $|E|$ from 1..20) for a subset of graphs (to avoid clutter, but others have similar behavior). The plot also indicates the execution time of the static version by the dotted lines (with the same color). We observe that the dynamic version takes much lesser time compared to its static counterpart for a few updates. As the number of dynamic updates increases, the amount of processing and, in turn, the execution time of the dynamic version increases almost linearly. In practice, we believe the number of dynamic updates would be small and our dynamic version would prove useful.

SSSP: Figure 9 shows the performance of fully dynamic SSSP with varying insertion and deletion percentage for a few graphs. Similar to BFS, dynamic SSSP performs better than its static counterpart until a graph-dependent threshold.

MST: Figure 9 shows the performance of the fully dynamic MST with varying insertions and deletions. The plot differs from the earlier ones (BFS and SSSP) in two aspects. One, dynamic MST is a complicated algorithm, its benefits get reduced due to higher synchronization costs. Therefore, it works better only up to a few thousand dynamic updates (which may be good for several applications). Hence, we plot directly the number of updates (1..20K) rather than a percentage (which would be very small). Second, we plot the performance over several graphs. Hence, instead of showing two lines per graph, we plot normalized exe-

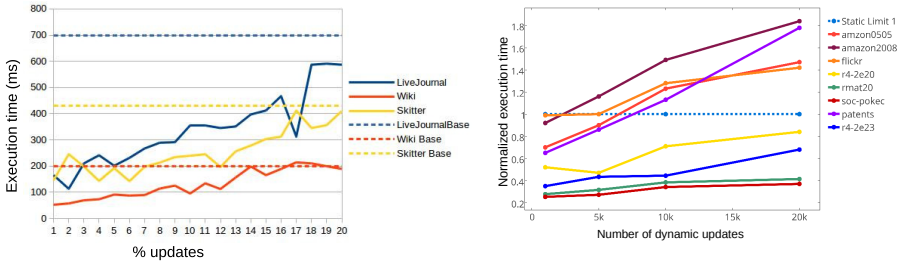
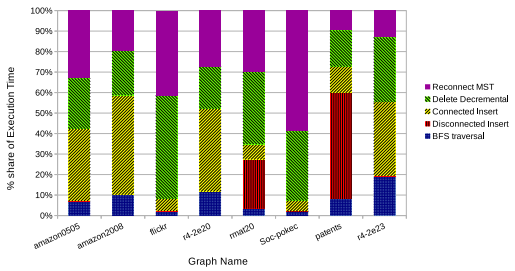


Fig. 9: Performance of fully dynamic SSSP and MST



Graph	# C	MST Cost
R4-2e23	1	369436934
Soc-Pokec	1	21453012
Patent	2238415	90836221
Flickr	1	9918223
Rmat20	91750	323677030
R4-2e20	1	1535353366
Amazon2008	1	14378130
Amazon0505	1	6389886

Fig. 10: Dynamic MST performance, and computed statistics

cution times. We observe a trend similar to that of BFS and SSSP, with dynamic MST performing better than the static version for a few thousand updates.

In Figure 10 we show the execution time split-up for dynamic MST across five stages of the algorithm: (i) level order BFS traversal to initialize *Parent*, *Level* and *Cost* arrays, (ii) connecting incremental edges across the disjoint MST components, (iii) inserting incremental edges within the MST component, (iv) marking and deleting decremental edges in the MST to form disjoint MST components, and (v) reconnecting the MST components to get the MST or MSF. We observe that fully dynamic MST is dominated in execution time by the decremental part which involves complicated processing in deleting the decremental edges and then reconnecting the MST components. Secondly, for graphs like rmat20 and patents, where there are multiple components, time consumed in adding incremental edges *across* the MST components is more than the time consumed for adding incremental edges *within* the MST tree component. This is because in these graphs most of the incremental edges have been added across the MST components rather than within an MST component. Figure 10 also shows the statistics obtained by our MST computation. `#C` indicates the number of connected components, and `Cost` indicates the total MST (MSF) cost. It shows that `Patent` and `Rmat20` are disconnected.

Overall, we illustrate that our dynamic versions of graph algorithms provide benefits over recomputing the graph analytic information from scratch. In particular, when the number of updates is relatively small – which happens in social networks, dynamic molecular simulations and control-flow graphs across code versions, our dynamic methods offer promising results.

6 Related Work

There exists a body of work on speeding up processing of evolving graphs [10,11,12]. While Chronos [10] introduces a novel memory layout to improve cache locality during serial or parallel graph processing, much of the other work restricts type of queries or are designed for a specific algorithm (e.g., Ren et al. [11] and Kan et al. [12] consider queries that depend upon the graph structure alone).

There are many implementations of parallel static graph algorithms on a variety of architectures, including distributed-memory supercomputers [13], shared-memory supercomputers [14], and multicore machines [15]. Harish and Narayanan [16] pioneered CUDA implementations of graph algorithms such as BFS and single-source shortest paths computation. BFS has received significant attention [17,18,4]. Hong et al. [3] propose a warp-centric approach for implementing BFS. In Pregel-like graph processing systems [19] some of the underlying algorithms like Page Rank, SSSP and DMST have been proposed for distributed processing. Vineet et al. [8] and Nobari et al. [20] propose computing the minimum spanning tree and forest, respectively, on GPUs. MST computation on temporal graphs [21] has also been proposed in the sequential setting.

Ashari et al. [22] propose an adaptive CSR layout for sparse matrix-vector multiplication. Their method reduces thread-divergence on GPUs by sorting vertices based on their degrees and binning the vertices with similar degrees. Adaptive CSR also uses dynamic parallelism supported in the latest GPUs to improve work-efficiency. King et al. [23] propose a dynamic CSR layout for graphs with changing structures. The difference between dynamic CSR and our diff-CSR (Section 2) is that dynamic CSR keeps track of additional segments to accommodate new edges. This leads to fragmentation when the segments are not full, and the authors propose a defragmentation step to compact the segment. In contrast, diff-CSR maintains a diff in the same CSR format, but the diff is separately maintained from the original CSR. diff-CSR may also incur fragmentation due to deletions, but since it never allocates more memory than required in a step, it does not incur fragmentation on insertion. cuSTINGER [24] proposes to store dynamic graphs on GPUs. It also uses arrays to store adjacency lists; however, diff-CSR uses two arrays (original and diff) . Further, unlike diff-CSR, cuSTINGER separates insertions and deletions, and needs a host-device copying of adjacency lists when the current storage space is insufficient for the dynamic updates. cuSTINGER also compacts the storage at the end of a batch deletion; diff-CSR does not perform compaction, but retains the deleted markings.

Closest to our work is the work on morph algorithms on GPUs [25] wherein structurally changing graphs are analyzed on the GPUs. While similar in spirit,

our work proposes a new dynamic graph representation and highlight new synchronization challenges. Automatic code generation for morph algorithms has been proposed by Unnikrishnan et al. [26].

7 Conclusion

We illustrated the promise in processing dynamic graph algorithms on GPUs. To address challenges posed by the structural updates, we proposed a backwards-compatible dynamic CSR representation, advocated data-driven processing, carefully chose the synchronization primitives, and took advantage of the interplay of incremental and decremental updates. By implementing and optimizing three popular graph algorithms in CUDA, we illustrated the promise in our proposed techniques. Using a collection of real-world and synthetic graphs, we showed that the proposed techniques work effectively and provide performance benefits over static graph algorithms up to a certain percentage of structural updates. We believe our techniques can be applied to other propagation-based algorithms such as Page Rank, Betweenness Centrality, and Coloring.

Acknowledgments

We thank the reviewers and our shepherd Nancy Amato for their comments which considerably improved our work. This work is partially supported by IIT Madras Exploratory Research Grant CSE/16-17/837/RFER/RUPS.

References

1. Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A High-performance Graph Processing Library on the GPU," in PPoPP, 2015.
2. 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 PACT, 2012.
3. S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in PPoPP, 2011, pp. 267–276.
4. D. G. Merrill, M. Garland, and A. S. Grimshaw, "Scalable gpu graph traversal," in PPoPP, 2012.
5. R. Nasre, M. Burtscher, and K. Pingali, "Data-Driven Versus Topology-driven Irregular Computations on GPUs," in IPDPS, 2013, pp. 463–474.
6. J. Leskovec and R. Sosič, "SNAP: A general purpose network analysis and graph mining library in C++," http://snap.stanford.edu/snap, Jun. 2014.
7. K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The Tao of Parallelism in Algorithms," in PLDI, 2011, pp. 12–25.
8. V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan, "Fast minimum spanning tree for large graphs on the GPU," in HPG, 2009, pp. 167–171.
9. M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in IISWC, 2012, pp. 141–151.

10. W. Hant, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: a graph engine for temporal graph analysis," in ECCS, 2014, p. 1.
11. C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng, "On querying historical evolving graph sequences," Proceedings of the VLDB Endowment, vol. 4, no. 11, pp. 726–737, 2011.
12. A. Kan, J. Chan, J. Bailey, and C. Leckie, "A query based approach for mining evolving graphs," in Proceedings of the Eighth Australasian Data Mining Conference-Volume 101. Australian Computer Society, Inc., 2009, pp. 139–150.
13. A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L," in SC, 2005, pp. 25–.
14. D. A. Bader and K. Madduri, "Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2," in ICPP, 2006, pp. 523–530.
15. M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," SIGPLAN Not. (PLDI), vol. 42, no. 6, pp. 211–222, 2007.
16. P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in HiPC, 2007, pp. 197–208.
17. L. Luo, M. Wong, and W.-m. Hwu, "An effective GPU implementation of breadth-first search," in DAC, 2010, pp. 52–55.
18. S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in PACT, ser. PACT'11, 2011.
19. M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," Proc. VLDB Endow., vol. 8, no. 9, pp. 950–961, May 2015.
20. S. Nobari, T.-T. Cao, P. Karras, and S. Bressan, "Scalable parallel minimum spanning forest computation," in Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 205–214. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145842
21. S. Huang, A. W.-C. Fu, and R. Liu, "Minimum spanning trees in temporal graphs," in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 419–430. [Online]. Available: http://doi.acm.org/10.1145/2723372.2723717
22. A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications," in SC, 2014, pp. 781–792.
23. J. King, T. Gilray, R. M. Kirby, and M. Might, "Dynamic Sparse-Matrix Allocation on GPUs," in ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings, J. M. Kunkel, P. Balaji, and J. Dongarra, Eds., 2016, pp. 61–80.
24. O. Green and D. A. Bader, "cuSTINGER: Supporting dynamic graph algorithms for GPUs," in 2016 IEEE High Performance Extreme Computing Conference (HPEC), Sept 2016, pp. 1–6.
25. R. Nasre, M. Burtscher, and K. Pingali, "Morph Algorithms on GPUs," in PPoPP, 2013, pp. 147–156.
26. U. Cheramangalath, R. Nasre, and Y. N. Srikant, "Falcon: A Graph Manipulation Language for Heterogeneous Systems," ACM Trans. Archit. Code Optim., vol. 12, no. 4, pp. 54:1–54:27, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2842618