International Journal of Parallel Programming (2022) 50:515–561
https://doi.org/10.1007/s10766-022-00740-7

Scaling the Maximum Flow Computation on GPUs

Jash Khatri[1] · Arihant Samar[1] · Bikash Behera[1] · Rupesh Nasre[1]

Received: 16 February 2022 / Accepted: 4 November 2022 / Published online: 15 November 2022

Abstract

Maximum flow is one of the fundamental problems in graph theory with several applications such as bipartite matchings, image segmentation, disjoint paths, network connectivity, etc. Goldberg-Tarjan's well-known Push Relabel (PR) Algorithm calculates the maximum s–t (source–target) flow on a directed weighted graph. PR algorithm has been effectively parallelized on GPUs. However, computing the maximum flow even using the GPU parallel PR algorithm continues to be time-consuming for large graphs. For the maximum flow algorithm's error-tolerant applications, it is sufficient to compute the approximate maximum flow value. In this work, we propose multiple techniques for improving the push-relabel algorithm's performance on the GPUs keeping its error-tolerant applications in mind. Our proposed techniques improve performance by carefully reducing the impact of the particular property that hampers the performance of the GPU parallel PR algorithm. These techniques provide tunable knobs to control the amount of approximation added and the respective performance achieved. In the end, we propose the Pull Relabel algorithm, which is the natural symmetric counterpart of the Push Relabel algorithm. Further, we combine both algorithms to construct a Pull-Push Relabel maxflow algorithm and analyze its effect on the dynamically changing graphs. We illustrate the effectiveness of our proposed algorithm and techniques using several real-world and synthetic graphs from the DIMACS Challenge, SNAP, Konect, and Network Repository, along with three maximum flow applications (Maximum Bipartite Matching, Team Elimination Problem, and Supply–Demand Problem). The proposals achieve 1.05× to 94.83× speedup over the exact GPU parallel push-relabel algorithm, and 14.29×, 40.40× and 32.41× speed-up on the three applications.

Keywords Maximum flow computation · Graph algorithms · Approximations · Push relabel algorithm

The last author is partially supported by NSM Grant CS/1920/1123/MEIT/008606.

✉ Jash Khatri
khatrijash15@gmail.com

[1] IIT Madras, Chennai, India

1 Introduction

Flow network in graph theory refers to a directed graph where each edge has a capacity. Each edge receives a flow where the total flow passing through an edge cannot exceed its capacity. Such flow networks are useful for modeling many real-world systems such as road networks, electrical circuits, computer networks, and many similar systems. The maximum flow is an important metric that denotes the maximum amount of feasible flow from one end to another in such flow networks. Over years, several sequential and parallel Algorithms [1–7] have been developed for computing the maximum flow for the given flow network.

Push-Relabel algorithm is one of the well-known algorithms for computing the maximum flow for a given flow network. The state-of-the-art Goldberg-Tarjan Algorithm [8] computes the maximum flow values in time $O(V^2E)$. Due to its high time-complexity, computation of maximum flow is quite time-consuming even on moderately-sized graphs. For example, a single-threaded execution of the Goldberg-Tarjan algorithm takes several hours to terminate on a `Genrmf` graph with a million vertices and 3 million edges.

The graphics processing unit (GPU) has become an essential part of high-performance computing systems. Its large number of processor cores helps in the performance improvement of many scientific applications with several independent computation tasks. Due to this, the parallel implementations of the Goldberg-Tarjan Algorithm [5] have been proposed for many-core GPUs to make it more scalable. This has reduced the execution time of maximum flow computation significantly. Despite this, however, computing the maximum flow on huge graphs continues to be time-consuming. For instance, computing maximum flow on the `Soc-Orkut` graph having 3 M nodes and 106 M edges using a parallel implementation of Goldberg-Tarjan's algorithm on a GPU consumes several minutes to finish. In addition to this, there exist applications of the maximum flow [9, 10] which are resilient to some amount of error in the final maximum flow results. Hence, allowing a small percentage of error in the flow value can be tolerable from the application semantics perspective.

As part of this work, we propose a set of techniques to optimize and approximate the hybrid CPU-GPU implementation of the Goldberg-Tarjan's Algorithm (GT) [5]. We observe various properties entailed by the algorithm, and the practical aspects of the graph processing to uncover efficient computation of the `push` and `relabel` operations. In particular, we make the following contributions:

– We present the GPU acceleration of maximum flow computation exploiting various properties of the flow computation, as well as approximate computing. In particular, we propose (i) vertex removal techniques to speed up the computation, (ii) an edge removal, and a graph renumbering scheme to reduce thread divergence in the GPU implementation of the Goldberg-Tarjan's algorithm, (iii) an excess kernel cycle pruning technique, (iv) re-using and reducing the min-height computations in the algorithm, (v) ways to skip GPU's

global memory accesses, and (vi) parallel breadth-first-search for the global relabelling operation (compared to traditional sequential BFS)
- While existing systemic works deal primarily with static graphs, we address the problem of maintaining the maximum flow across structural graph changes. Towards this, we propose a hybrid pull-push relabel algorithm for computing the maximum flow on dynamically changing graphs.
- We apply our optimized maximum flow computation to three simple applications: Maximum Bipartite Matching, Team Elimination Problem and Supply–Demand Problem, and illustrate its effectiveness.
- We qualitatively as well as quantitatively assess the effectiveness of the proposed algorithm and the techniques using several real-world and synthetic large flow networks. We observe that our proposal accelerates the maximum flow computation by 1.05× to 94.83× and achieves a 14.29×, 40.40× and 32.41× speedup on the three applications when compared to the state-of-the-art GPU-based Push-Relabel algorithm's implementation provided in Gunrock [11]. We list several takeaways from this evaluation which can be helpful to the community.[1]

The rest of the paper is organized as follows. Section 2 provides a brief background on the push-relabel algorithm and formally defines the problem. In Sect. 3, we develop and demonstrate novel techniques to improve the push-relabel algorithm's performance on the GPUs, along with their implementation details. We demonstrate the hybrid pull-push relabel algorithm within the same section. Section 4 shows how to compute the maximum flow on the dynamically changing graphs. Section 5 presents the experimental evaluation of the proposed algorithm and techniques. Section 6 demonstrates a real-world applications of the maximum flow algorithm and evaluates it by applying the proposed techniques. Section 7 summarizes the related work on the maximum flow algorithms. Section 8 concludes the paper with a discussion on future research directions. In "Appendix A", we study the effectiveness of our techniques on the remaining two real-world applications of the maximum flow algorithm.

2 Problem Statement and Background

A flow network is a directed graph G(V, E, c) where V and E denote the sets of vertices and edges in the graph, and each edge (u, v) $\in$ E has a capacity c(u,v). In addition to this, the graph has a source vertex s $\in$ V and a sink vertex t $\in$ V. The flow function, or merely the flow, defined on each edge of the flow network, is denoted by f: V $\times$ V $\rightarrow \mathbb{R}$. The flow must satisfy the following constraints:

1. Capacity constraint: The flow along an edge can not exceed the edge capacity, i.e., $0 \leq f(u, v) \leq c(u, v), \forall (u, v) \in E$

[1] Our code is publicly available at https://github.com/Jash-Khatri/IJPP.

2. Anti-symmetry constraint: The net-flow across any edge (u,v) is zero, i.e.,f(u, v) = − f_b(v, u), ∀ (u, v) ∈ E, where f_b denotes the flow function on backward edges.
3. Conservation constraint: The flow entering into a vertex v is the same as the flow leaving out of v, except at the source s and the sink t, i.e., $\sum_{(v,w)\in E} f(v, w) - \sum_{(u,v)\in E} f(u, v) = 0, \forall\, v \in V - \{s, t\}$.

The val(f) denotes the value of a flow f and is defined as, val(f) = $\sum_{j \in V} f(s, j) = \sum_{j \in V} f(j, t)$. The flow f' is said to be the maximum flow for the given flow network G if there does not exist any flow f'' in the same flow network G such that val(f'') > val(f').

Algorithm 1: Goldberg-Tarjan's algorithm

```
Input: A Flow Network G(V, E, c), source s, sink t
Output: Maximum flow value
let e(v) = 0; ∀v ∈ V ;                                   // Excess flow
let h[s] = n ;                                  // Initialize the heights
let h[v] = 0; ∀v ∈ V - {s}
create a pre-flow f that saturates all out-going edges of s
while ∃ v ≠ s, t with e(v) > 0 do
    choose such a vertex v with the largest height.
    if ∃ (v, w) ∈ E with h(v) = h(w) + 1 then
        Push the excess flow from v to w.
    else
        Relabel the vertex v.
    end
end
return e(t);
```

2.1 Goldberg-Tarjan's Algorithm

The Golderg-Tarjan's (GT) Algorithm [8] is one of the well-known algorithms for computing the maximum flow in the given flow network. Its time complexity is $O(n^2m)$, where n denotes the number of vertices and m denotes the number of edges in the input flow network. It is presented in Algorithm 1.

A crucial property of the GT algorithm is that it relaxes the conservation constraints (discussed above) by allowing the difference between the incoming and the out-going flows to be non-negative during the execution of the algorithm instead of being strictly zero. This difference between the incoming and the out-going flows associated with each vertex is termed as *excess flow* and is denoted as e(v) in Algorithm 1. The GT algorithm also associates the height with each vertex, denoted as h[v]. A vertex with a higher height can push its excess flow to its neighbor vertices with lower height in the GT algorithm. The height for sources vertex is initialized to the number of nodes in the input flow network, while the rest of the vertices' height is initialized to zero. The pre-flow is computed using the above concept by saturating all the out-going edges from the source vertex during the initialization step, as

shown by Lines 1–4 in Algorithm 1. Hence, pre-flow contains the vertices in the flow network, which will have the positive excess values. When the conservation constraint is again satisfied for all the vertices, pre-flow becomes the maximum flow of the flow network, and the algorithm terminates. At this stage, the excess flow accumulated at the sink vertex, i.e., e(t), denotes the maximum flow value.

2.2 GPU Parallel Push-Relabel Algorithm

We briefly explain the parallel push-relabel algorithm for GPUs [5], as presented in Algorithm 2. The value $c_f(u, v)$ denotes the *residual capacity* of each edge of the flow network G(V, E, c), where $u, v \in V$. For the given flow f and the flow network G(V, E, c), it is calculated as c(u,v) - f(u,v), where $u, v \in V$. The *residual network* of a given input flow network is denoted as G_f(V, E_f, c_f), where $E_f = \{(u, v)|u \in V, v \in V, c_f(u, v) > 0\}$.

Algorithm 2: GPU Parallel Push-Relabel Algorithm

```
1  Initialize e, h and c_f
2  foreach (s, u) ∈ E do
3  |   ExcessTotal ← ExcessTotal + c(s,u)
4  end
5  copy e and c_f from the CPU main memory to the CUDA global memory
6  while e(s) + e(t) < ExcessTotal do
7  |   copy h from CPU → GPU
8  |   call push_relabel kernel ;                          // GPU
9  |   copy c_f, h, and e to CPU ← GPU
10 |   call global_relabel_cpu() ;                         // CPU
11 |   foreach u ∈ V do
12 |   |   if e(u) > 0 and h(u) > |V| then
13 |   |   |   ExcessTotal ← ExcessTotal - e(u)
14 |   |   end
15 |   end
16 end
17 return e(t);
```

The initialization step in Algorithm 2 is similar to that in Algorithm 1 where we set the initial values for heights(h), excess flow(e) for each vertex and calculate the residual capacity(c_f) for each edge in the input flow network and generate the residual network. Additionally, the *ExcessTotal* variable is used to keep track of the total amount of the excess flow present at all the vertices of the residual flow network. Since the pre-flow is computed by saturating all the out-going edges from the source vertex during the initialization step, the *ExcessTotal* variable is initialized to the total amount of the excess flow that each vertex connected to source vertex has during the pre-flow. The main `while` loop (Line 6) transfers all the necessary data to the GPU memory and then performs the concurrent push and relabel operations on each vertex of the residual network (except the source and the sink) by launching `push_relabel kernel`. The variable KERNEL_CYCLES indicates the number of times we repeat executing the `push_relabel kernel` code for a

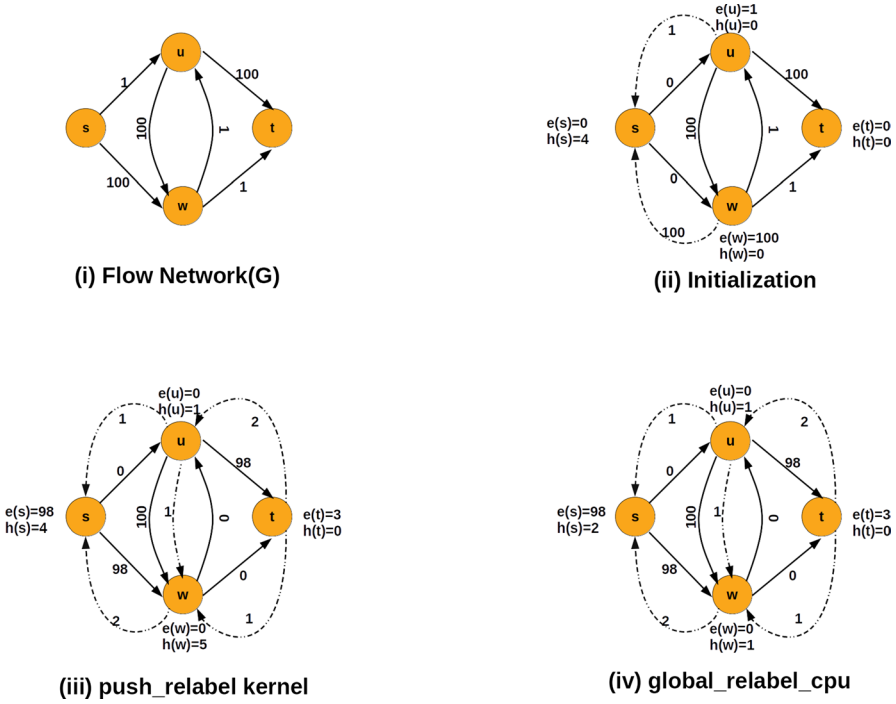


Fig. 1 Execution steps of GPU Parallel Push-Relabel algorithm on an instance flow network(G)

particular thread. We use the term KERNEL_CYCLES and cycles interchangeably in the rest of the writing. Hence, inside the `push_relabel kernel` each thread operates on some particular vertex, and it performs the push or relabel operation on that vertex for the KERNEL_CYCLES number of times. After every thread finishes KERNEL_CYCLES push or relabel operations on its assigned vertex, the `push_relabel kernel` terminates, and then the algorithm transfers the e, h, c_f back to the CPU memory from the GPU. The CPU then performs the global relabeling operation by calling the `global_relabel_cpu` function, which adjusts the heights of the vertices with their shortest distance to the sink. This is done by performing the backward breadth-first search (BFS) from the sink on the residual network. The algorithm then removes positive excess flow associated with the inactive vertices from the *ExcessTotal* variable. The vertex 'u' is said to be inactive if either $e(u) = 0$ or $h(u) > |V|$. Hence the vertex with positive excess flow can be inactive if $e(u) > 0$ and $h(u) > |V|$. After that, the algorithm checks for its termination condition, which is stated as follows: If the summation of excess flow value on the source (e(s)) and the excess flow value on the sink (e(t)) is equal to the ExcessTotal, then it implies that there are no active vertices in the residual network (except the source and the sink) and hence the main `while` loop terminates. The final excess flow value on the sink vertex (e(t)) provides the maximum flow.

Example Figure 1 shows working of Algorithm 2 on a sample flow network. Step (i) shows the initial flow network G, with each edge augmented with its capacity.

The vertex s denotes the source vertex, and vertex t represents the sink vertex. Step (ii) shows G after the initialization step shown in Lines 1–4 of Algorithm 1. In this step, the algorithm initializes the height h and excess flow e for each vertex to zero, except for the source vertex. vertex s, the height is set to 4, equal to the number of vertices in G. After this, the algorithm saturates all the outgoing edges from the source vertex. Hence, after the initialization step, we have e(u) = 1 and e(w) = 100, along with residual edges u→s and w→s. The ExcessTotal variable is updated to value 101 in this step. The algorithm now executes the while loop to execute the push_relabel kernel on GPU. For this example, we took the KERNEL_CYCLES as 10. Hence, inside the push_relabel kernel, one GPU thread each for vertex u, w will either perform push or relabel operation on that vertex for ten times and will modify the value of height h and excess flow e for that vertex accordingly. The flow network after execution of push_relabel kernel is shown in Fig. 1 as Step (iii).

Next, the algorithm transfers the required data back to CPU and executes Step (iv): global_relabel_cpu function (Line 10). This function performs a breadth-first search (BFS) from the sink vertex t along the residual edge and sets the vertex's height as the level of the vertex in the BFS tree. Hence, after executing the global_relabel_cpu function, the heights of vertices s, u, w, t become 2, 1, 1, 0, respectively. The flow network G after execution of global_relabel_cpu is shown in Fig. 1 as Step (iv). After this step, the algorithm prunes the inactive vertices by executing Lines 11–15. However, as there are no inactive vertices after Step (iv), the flow network remains the same.

This completes one iteration of the while loop. Before starting the next iteration, the algorithm checks if $e(s) + e(t) < ExcessTotal$. However, $e(s) + e(t) = 98 + 3 = 101$. Hence, the algorithm halts by returning e(t) = 3. □

Algorithm 2 efficiently utilizes the large number of cores available on the GPU to perform the push and relabel operations concurrently on all the active vertices of the flow network, and computes maximum flow value faster than the sequential Algorithm 1. In this work, we work with this algorithm as the baseline but compute the max-flow faster by allowing a small error in the maximum flow value.

3 Our Techniques for Faster MaxFlow

This section describes in detail our techniques for the faster maximum flow computation. We summarize these in Table 1. Third column mentions if the technique is an approximation (A) or an optimization (O).

3.1 T1: Parallel BFS with Frequent Relabeling

The global relabeling heuristic periodically performs backward breadth-first search from the sink in the residual network G_f to compute the exact labels of the nodes. This BFS helps in reaching the excess flow in the vertices to the sink faster. It starts with the sink in the queue and every time we pop a vertex, say v from the queue, we check its neighbours. If a neighbour vertex u has not been visited and there is an

Table 1 Summary of techniques in this manuscript

Tech	Name	Opt./Approx	Expl. Sect.	Eval. Sect.
T1	Parallel BFS with Frequent Relabeling	O	3.1	5.2
T2	Reusing min-height computations	O	3.2	5.3
T3	Excess cycle removal	O	3.3	5.4
T4	Vertex Removal	O	3.4	5.5
T5	Edge Removal and Vertex Renumbering	A	3.5	5.6
T6	Memory access skipping	A	3.6	5.7
T7	Push-Pull Relabel Algorithm	O	3.7	5.8

edge from u to v in the residual graph, which simply means some flow can go from u to v, we assign u as a child of v, assign $h[u] = h[v] + 1$ and add u to the queue. Then the algorithm performs a second reverse BFS from the source so that vertices which are not visited yet have to send their excess flow back to the sink. In effect, a considerable fraction of over-relabelling involves BFS. The global relabelling is done within each iteration, as shown in Algorithm 2. During experimentation, we observe that global relabelling takes 26.95 s on average on the graphs of our dataset. Thus, considering that large graphs can consume a large number of iterations (over 30 iterations for many of our graphs), a significant amount of time is spent doing the relabelling itself.

Parallelizing BFS on GPU can considerably help. We use a method adapted from Luo et al. [12] which uses hierarchical queues and works efficiently with shared memory.

Algorithm 3 shows the parallel BFS function that we have designed for performing the global relabel operation. Algorithm 3 has four significant steps:

1. We have a current queue, *d_currentQueue*, of vertices which consists of all the vertices at some level. Initially, *d_currentQueue* consists of only the source. We launch a kernel in which each thread operates on one vertex of the queue. Each thread iterates through the neighbors of its vertex v. If the neighbour is not visited yet and there is an edge from the neighbour to v in the residual graph, we assign v as the parent of the neighbour. This is done in Line 5 of Algorithm 3. Note that no synchronization is required here as the processing simply assigns a parent if the conditions are valid. If the same vertex is a neighbour of two vertices being operated on by two threads simultaneously its parent will be one of the two vertices at random.
2. After going through all the vertices in the current queue, we need to count the number of children assigned to every vertex. This helps in parallel filling of the vertex queue for the next level. This is done in function `countDegrees` in Line 6 of Algorithm 3.

3. For a vertex v, we have $degree[v]$ number of children assigned to it. In the queue for the next level, the first child of v would be present at index $degree[0] + degree[1] + \cdots + degree[v-1]$. This can be done by computing the prefix sum of the degree array, shown in function `scanDegrees` in Line 7.
4. Based on the prefix sum values, we fill the queue *d_nextQueue* for the next level. Finally we exchange the two queues for the next iteration.

The parallel BFS described above help in reducing the execution time of global relabelling operation. This improvement in the time for one global relabelling allows us also to perform labeling more frequently. Originally, in one iteration first, the BFS labeling is performed. Then there are some set number of cycles for which push-relabel operation performed on the whole graph. In one cycle, a kernel is launched where one thread operates on a few vertices and performs a push or relabel operation, whichever is applicable if the vertex is active. Thus the global relabelling followed by some cycles of push-relabel operations constitute one iteration.

For example for a graph with 100k vertices, 100k cycles of push-relabel operations on the graph were being performed. Now we can reduce that to a lesser quantity, say just 100. This frequent relabelling though will increase the total number of iterations for convergence will also help keep the number of unhelpful relabel operations which are a bottleneck of the algorithm to a minimum, and hence improve the running time.

Algorithm 3: Parallel BFS algorithm

```
1  int queueSize = 1 ;
2  int nextQueueSize = 0 ;
3  int level = 0 ;
4  while queueSize do
       // Step 1: next layer phase
5      nextLayer(level, queueSize) ;
       // Step 2: counting degrees phase
6      countDegrees(level, queueSize) ;
       // Step 3: performing a scan on degrees
7      scanDegrees(queueSize) ;
8      nextQueueSize = incrDegrees[(queueSize - 1) / 1024 + 1] ;
       // Step 4: assigning vertices to nextQueue
9      assignVerticesNextQueue(queueSize, nextQueueSize) ;
10     level++ ;
11     queueSize = nextQueueSize ;
12     std::swap(d_currentQueue, d_nextQueue) ;
13 end
```

We assess the performance benefits obtained using parallel BFS with frequent relabeling in Sect. 5.2. Since parallelizing the BFS does not affect the performance of the original algorithm, this technique serves as an optimization over the original parallel maximum algorithm. We observe the geomean speedup of 94.83× using parallel BFS with frequent relabeling over the sequential BFS.

3.2 T2: Reusing Min-height Computations

This technique relies on the following property.[2]

Property 1 *If a cycle in the algorithm consists of only the push operations, then the height of each vertex in the residual network does not change in that cycle.*

Proof This is evident from the fact that push operations do not manipulate the vertices' heights. □

When the computation follows the scenario in Property 1, we re-use the min-height computed by each vertex from the previous cycles for the next cycle. Otherwise, we find the min-height neighbors for every vertex and perform the push or relabel operation accordingly. We also store the current min-height neighbor for each vertex during this step. When we re-use the min-height computed by each vertex in the previous cycles, two cases arise. In the first one, if the vertex has the valid min-height neighbor in its previous iteration, it can re-use its min-height computation for the current cycle. Hence, we can save recomputing the min-height for that particular vertex in the current cycle. In the second case, if the vertex has the invalid min-height neighbor in its previous iteration, then the thread will not push the flow from the vertex assigned to it to its min-height neighbor. In such a case, the given vertex needs to recompute its min-height neighbor and store it as its valid min-height neighbor. Since we re-use the min-height when we have all the pushes in the current cycle, this optimization technique's effectiveness can be given by Eq. 1.

$$E \propto S * T \tag{1}$$

E = Total performance gain,
S = Number of cycles consisting of only the push operation, and
T = Average number of vertices storing the valid min-height neighbors.

[2] We note that the observations made in the following properties are well-known, and not ours. However, most of the earlier works on parallel max-flow computation have not exploited these properties, to the best of our knowledge, towards approximation and optimization.

Algorithm 4: Reusing min-height computations in the Parallel Push_Relabel kernel

```
1  cycle = KERNEL_CYCLES ;
2  while cycle > 0 do
3      if u == 0 then
4          reset( numofrelabels[(counter+1) mod 2] ) ;
5      end
6      if e(u) > 0 and h(u) < |V| then
7          e' ← e(u) ;
8          h' ← ∞ ;
9          if isset(numofrelabels[(counter) mod 2]) or !isvalid(lowest_neighbor[u])
            then
10             foreach (u, v) ∈ E do
11                 h'' ← h(v) ;
12                 if h'' < h' then
13                     v' ← v ;
14                     h' ← h'';
15                     lowest_neighbor[u] ← v ;
16                 end
17             end
18         else
19             h'' ← h(lowest_neighbor[u]) ;
20             v' ← lowest_neighbor[u] ;
21             h' ← h'';
22             set( numofrelabels[(counter+1) mod 2] ) ;
23         end
24         if h(u) > h' then
25             push(u);
26         else
27             h(u) ← h' + 1;
28             set( numofrelabels[(counter+1) mod 2] ) ;
29         end
30     end
31     cycle ← cycle - 1 ;
32 end
```

Algorithm 4 shows the parallel push_relabel kernel, which reuses the min-height computations. Following changes to the original push_relabel kernel are warranted: Lines 3–5 reset the `numofrelabels` value of the next stage. This means initially, we assume that there are no vertices performing the relabel operation in the current cycle. Line 9 checks if the `numofrelabels` value for the current stage is set, which means there were relabel operations in the previous stage or the `lowest_neighbor` array holds invalid value for the vertex u. This indicates that the lowest neighbor information for vertex u is not updated in the array. If either of them is or both are true, then we continue with the min-height computation as per Lines 10–17. We also update the `lowest_neighbor` array during this operation to reflect the latest min-height neighbor for vertex u, shown in Line 15. If the situation mentioned above does not hold, then we know that there was no relabel operation in the previous cycle, as well as `lowest_neighbor[u]` carries a valid value. Hence, we

can reuse this value for the current cycle since the height has not changed since the previous cycle. This is shown in Lines 18–23. Line 28 sets the `numofrelabels` if the relabel operation is performed by any vertex in the current cycle. All the elements in `numofrelabels` and `lowest_neighbor` array are initialized to True and INVALID, respectively.

We observed 1.052× performance improvement using this technique on the entire data-set (discussed in Sect. 5.3). During experimentation, we also observed that there could be a large number of cycles (e.g., up to 463,000 for Genrmf graphs) during which we can reuse the min-heights. This technique is an optimization and not an approximation. This is because when we are reusing the min-height neighbors of a vertex, we will end up pushing either some positive flow value if the edge connecting the given vertex and its min-height neighbor is not saturated, or zero flow value if the edge connecting the given vertex and its min-height neighbor is saturated. In either case, we do not add any error in the maximum flow computation.

3.3 T3: Excess Cycle Removal

Our technique of removing excess cycles results in early termination, and is based on the following simple observation.

Property 2 *If the excess flow value associated with each vertex except the source and the sink vertices in the flow network is zero during the execution of an iteration of the push-relabel kernel, then the next kernel cycles are redundant.*

Proof If the excess flow value associated with each vertex except the source and the sink vertices in the flow network is zero, then each thread associated with any particular vertex will not be able to push any further excess flow to the sink. □

Based on Property 2, if we detect the excess flow value associated with all the vertices except the source and the sink to be zero in the flow network, then we elide the rest of the kernel cycles in that iteration. This can be easily done by setting the KERNEL_CYCLES variable to zero in the push-relabel kernel. By detecting and pruning the unnecessary cycles during the push-relabel kernel's execution, we adapt the early-termination property for the kernel.

A crucial point here is that the excess cycles removed (if any) will always be from the last iteration of the algorithm's execution. This is because if the algorithm needs further iteration for finishing its execution, there will be at least one vertex with a positive excess flow value in the flow network. Hence, the excess cycle removal technique will not prune the cycles from such an iteration. However, for the last iteration, there may be no active vertices (with positive excess flow value), and still, the algorithm keeps executing some cycles to complete the KERNEL_CYCLES number of cycles. Such cycles are detected and pruned by this technique.

During experimentation, we observe that the value of KERNEL_CYCLES is set based on the number of vertices in the input flow network. Since the number of kernel cycles needed to converge on a flow network varies and can not be predicted

easily, early termination detection helps avoid unnecessary computation resulting in improved performance.

We quantitatively assess T2's effectiveness in Sect. 5.4 resulting in a 2.07× speedup on the entire data-set. We observe that this technique works well on large real-world and RMAT graphs. However, we did not observe any significant performance improvement by applying this technique on the small graphs (with a few thousand vertices) and on the Genrmf graphs. Since we are pruning only the unnecessary cycles from the push-relabel kernel code, this technique also acts as an optimization.

3.4 T4: Vertex Removal

Our technique of removing vertices is based on the following three observations.

Property 3 *A vertex with zero out-degree does not contribute to the final maximum flow value unless it is the source vertex.*

Proof If a vertex has zero out-degree, then during the execution of the push-relabel algorithm, it may accumulate the excess flow with it, but it has no way to push that excess flow anywhere. Hence, such a vertex will not drive any flow to the sink. Hence, vertex with the out-degree zero does not contribute to the final maximum flow value. □

Property 4 *A vertex with zero in-degree does not contribute to the final maximum flow value unless it is the sink vertex.*

Proof If a vertex has zero in-degree, then during the execution of the push-relabel algorithm, it cannot accumulate the excess flow inside it from anywhere. Hence, it can not drive any flow to the sink vertex. Hence, we say zero in-degree does not contribute to the final maximum flow value.

Property 5 *A vertex with zero in-degree and zero out-degree does not contribute to the final maximum flow value.*

Proof Suppose a vertex has zero in-degree and zero out-degree, then, during the execution of the push-relabel algorithm, it can neither accumulate the excess flow nor push the flow within it anywhere. Hence, the vertices with zero in-degree and zero out-degree do not contribute to the final maximum flow value. □

Using Properties 3, 4, and 5, we prune all the vertices with zero in-degree, zero out-degree, and both from the input flow network as a preprocessing step. We know that one of the primary bottlenecks in CPU–GPU systems are the inter-device data transfer over a relatively slow PCIe interconnect, which manifests itself in GT algorithm implementation as well. Vertex pruning improves CPU-GPU bandwidth by avoiding unnecessary data transfer.

We assess the effectiveness of T3 in Sect. 5.5. As evident, T3 does not result in approximation in the max-flow value.

3.5 T5: Edge Removal

In typical vertex-based processing of the PR kernel on GPU, a thread is assigned to a vertex or a set of vertices. A crucial step executed by each thread is to find the lowest height neighbor by traversing all the neighbors of the assigned vertex. Based on the number of neighbors each vertex has, the time taken for this operation varies from one thread to another. This leads to thread divergence for the warp threads on GPU when the degree distribution is skewed, leading to reduced performance.

Algorithm 5: Edge Removal as a preprocessing routine

```
   // wt is the weight with which average degree is multiplied
1  weighted_avg_degree = wt × (numEdges / numNodes) ;
2  foreach (u, v) ∈ E do
3  |  if min(deg(u), deg(v)) > 0 then
4  |  |  edge_stay = min(1.0 , weighted_avg_degree / min(deg(u), deg(v))));
   |  |  // th is the threshold on the edge_stay value of a particular
   |  |     edge
5  |  |  if edge_stay < th then delete(u, v) ;
6  |  end
7  end
```

To reduce this adverse effect, we use edge removal as the approximation technique on the input flow network. In this technique, we remove the edges connected to the vertices with probability proportional to their degrees. Algorithm 5 shows the routine used for performing the edge removal. Line 1 initializes the variable weighted_avg_degree based on the average vertex degree where `wt` is a user-defined constant. Lines 2–7 in the above routine checks for each edge (u, v) if the minimum degree of either of the two vertices attached to it is non-zero. If so, it computes its edge-stay value based on Eq. 2.

$$edge\_stay = min(1.0, weighted\_avg\_degree/min(deg(u), deg(v))))) \tag{2}$$

If the computed edge-stay value is above the user-defined threshold `th`, then we retain that edge; otherwise, we remove it. Note that there are two parameters, `wt` and `th`, that the users can tune to control the effect of edge removal on the flow network. Since this technique normalizes the degree distribution by removing edges from the high-degree vertices, it reduces thread divergence. This technique may perturb the maximum flow results as we are not preserving the maximum flow property by removing the edges from high-degree vertices.

Vertex Renumbering To further mitigate the effect of thread divergence, we apply vertex renumbering as the optimization technique on the input flow network. This technique renumbers the vertices in the input flow network such that vertices

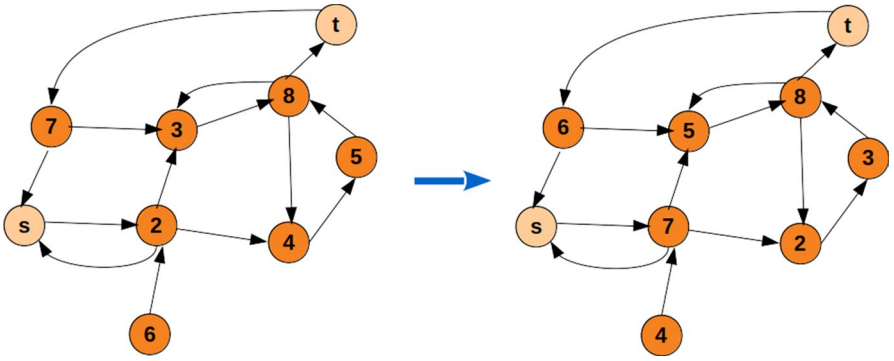


Fig. 2 Original graph and the modified graph with renumbered vertices

with a similar degree are assigned the nearby ids. This way, we group the vertices with similar degrees together. Due to such a grouping, warp threads access the vertices with almost similar degrees, reducing divergence.

Figure 2 shows an example. If we assume the warp size equal to 4, then the nodes 2–5 are assigned to threads belonging to the same warp (ignoring the source). Node 2 has an out-degree of 3, while nodes 3–5 have an out-degree of 1 each. Due to this (small) uneven degree distribution, the thread operating on node 2 would consume more time, causing thread divergence. To avoid this, T4 renumbers the nodes as shown in Fig. 2. In the renumbered flow network, nodes 2–5 have the same out-degree, i.e., 1. Hence, the warp threads operating on them will have almost the same amount of work.

We assess the effectiveness of both the above techniques in Sect. 5.6. We observed a speedup of 1.09× by applying edge removal and vertex renumbering on the entire dataset. incurring 0.00% error (minimal).

3.6 T6: Memory Access Skipping

Our profiling reveals that accessing arrays e (excess flow), h (height), c_f (residual capacity) is the most time-consuming among all the memory accesses. This is evident as these are stored in the GPU global memory, and push and relabel operations access these arrays very frequently. Second heuristic observation is that the magnitude of excess flow has a bearing on the usefulness of pushing it. Thus, a small excess flow often does not result in substantial change in the maxflow value. We combine these two observations to improve the overall memory accesses. Under our technique, called memory access skipping, a thread skips processing a particular vertex assigned to it if the excess flow value associated to that vertex is below a threshold. Processing a vertex involves performing either a push or a relabel on that vertex which, in turn, needs access to several parameters such as excess flow, residual capacities, the height of the vertices, etc., which are present in the GPU global memory. By not processing a vertex if its excess flow value is below the threshold, we save some costly accesses to GPU's global memory (note that caching is not

useful here due to irregular nature of the underlying computation). To contain the inaccuracy, tuning the threshold is critical in this case. Section 5.2 shows the effect of varying the threshold value on the speed-up and inaccuracy of parallel push-relabel algorithm. We observed a speedup of 1.09× by applying the memory access skipping technique on the entire dataset with the 4.4% error in the maximum flow value.

3.7 T7: Push-Pull Relabel Algorithm

We now discuss combining push-based and pull-based procedures.

3.7.1 Pull Relabel Algorithm

A pull relabel algorithm would be the natural symmetric counterpart of the push-relabel algorithm. Vertices pull flows from their neighbours to satisfy their flow deficit, similar to the original algorithm in which we push the excess flow from a vertex to its neighbour. Initially, the sink will perform saturating pulls from all its neighbours and then the pull or relabel operations will continue. This is similar to the push relabel algorithm, where we initially perform saturating pushes from source to all its neighbouring vertices. The negative excess travels to the source where the vertices can pull the required amount of flow.

If there is no path from the source to the current vertex to pull flow from, we need to pull the flow from the sink itself, much like how flow which cannot reach the sink is sent back to the source in the push-relabel algorithm.

Similar to how we enforce the condition that when we push a flow from u to v, $h(u) >= h(v) + 1$, when vertex u pulls flow from vertex v, the condition to be enforced is $h(u) <= h(v) - 1$. This is shown by Line 1 in Algorithm 6. The relabel operation in the original algorithm is to find the neighboring vertex to which flow can be pushed, that neighbour which has the minimum height, and assign to the vertex a height one more than the minimum height. Similarly, in this modification, we try to find among all neighbours from which flow can be pulled, that neighbour has the maximum height and assign one less than the maximum as the new label for the vertex. This is shown by Lines 1 and 2 in Algorithm 7.

Algorithm 6: pull(u, v): Pull function

```
1 assert e(u) < 0 and h[u] <= h[v] − 1
2 Δ = min(−e(u), c(v, u) − f(v, u))
3 f(v, u) += Δ
4 f(u, v) −= Δ
5 e(v) −= Δ
6 e(u) += Δ
```

Algorithm 7: pull_relabel(u): Relabel operation corresponding to the Pull function

```
1 assert e(u) < 0 and h[u] >= h[v]  ∀v | c_f(v, u) > 0
2 h[u] = −1 + max(h[v]  ∀v | c_f(v, u) > 0)
```

Combining the Two Algorithms If we can do both these push-pull operations simultaneously, we would achieve a bidirectional flow, since vertices close to the source would push their excess flow to the farther vertices (from the source) whereas vertices closer to the sink would pull flow from vertices far away from the sink to nullify their deficit. This is likely to speedup the computation.

3.7.2 Two Types of Global Relabelling

Unfortunately, the BFS relabelling with a push-relabel algorithm and that with a pull-relabel algorithm conflict. In the push-relabel algorithm, we want the excess to reach the sink, demanding a reverse BFS from the sink. After that, if a vertex has an excess and is not visited yet, we would like to return it to the source. Hence we ought to run a second reverse BFS from the source. We assign the child height as $h[child] = h[parent] + 1$ in the push-relabel algorithm.

On the other hand, in the pull-relabel algorithm, we would like the deficit vertices to pull flow from the source. This demands a forward BFS from the source. We ought to run a second BFS from the sink to assign the height for the unvisited vertices since they have to pull flow from the sink itself. We assign the child height as $h[child] = h[parent] - 1$ in the pull-relabel algorithm.

Individually, the two algorithms work perfectly. But combining them in naïve way results in an incorrect maxflow computation. We combine the two procedures (push and pull) by performing the labellings in alternate iterations. Despite separating them in odd / even iterations, a challenge is the simultaneous presence of both types of excesses: positive and negative.

We explain the difficulty in more detail now. Consider the push-styled BFS when we are performing a backward BFS from the sink. For a vertex v and its neighbour u, the default way is to check if in the residual graph there is an edge from u to v. But if v has pulled flow from u, causing u to have a negative excess, there may not be an edge from u to v in the residual graph. Hence we should also check if v has pulled flow from u. This will be the case if $e(u)$ is negative and there is some flow going from u to v. Similarly, during the forward BFS in the pull-styled BFS, we can assign a neighbour u as a child of v, if there is an edge from v to u in the residual graph or if v has pushed flow to u, u has a positive excess and there is flow from v to u.

This logic is presented in Algorithm 8. The value of *type* variable is 1 for a push-styled BFS and -1 for a pull-styled BFS. In push-type BFS, we will not consider the neighbour as a child if there is no flow possible from neighbour to u ($c_f(reverse(w)) < \epsilon$) and the neighbour does not have any negative excess with some flow going towards u.

Algorithm 8: Deciding if a neighbour is a child

```
   // w is the edge from current vertex v to its neighbour u
   // reverse(w) is the reverse edge, from neighbour to v
   // d_parent[v] is the parent edge id of vertex v
 1 if type == 1 then
 2 |   if c_f(reverse(w)) < ε and (e(u) >= 0 or c(reverse(w)) < ε) then
 3 |   |   continue ;
 4 |   end
 5 end
 6 else
 7 |   if c_f(w) < ε and (e(u) <= 0 or c(w) < ε) then
 8 |   |   continue ;
 9 |   end
10 end
11 if d_parent(u) == UNDEFINED then
12 |   h(u) = level + type ;
13 |   d_parent(u) = w ;
14 end
```

Thus, at a high level, the algorithm is to run a push-styled global relabelling, then perform a few cycles of push operations on those vertices with positive excess as shown by Lines 2 and 3 in Algorithm 9, and alternate with a pull-styled global relabelling followed by pull operations on the vertices with negative excess which is depicted using Lines 6 and 7 in Algorithm 9.

Algorithm 9: One iteration of Push-Pull relabel Algorithm

```
1 if iteration % 2 == 0 then
2 |   Perform push-styled BFS
3 |   Call Push-Relabel kernel // operating on vertices with positive excess
4 end
5 else
6 |   Perform pull styled BFS
7 |   Call Pull-Relabel kernel // operating on vertices with negative excess
8 end
```

We show the performance analysis of the Push-Pull relabel algorithm in Sect. 5.8. This algorithm correctly computes the maximum flow values, and hence it acts as an optimization for the original push-relabel algorithm. We observe the geomean speedup of 1.5× using the Push-Pull relabel algorithm over the baseline Push-Relabel algorithm.

3.8 GPU-Specific Processing of the Push-Relabel Algorithm

There are two main components in Algorithm 2 that are most suitable for GPU computing. The first is the push_relabel kernel shown as part of Line 8, whose

implementation is discussed as part of this section. This component can be offloaded to the GPU because it is highly parallel, as each thread can perform either push or relabel operations on its assigned active vertices independently. Hence we can exploit GPU's massive multithreading here to make it efficient. The second part is the global_relabel_CPU function shown as part of Line 10, which Gunrock performs on the CPU. However, as discussed in S.ection 2.2, the global_relabel_CPU function internally executes a BFS, and is hence amenable for efficient GPU parallelization [12]. We have discussed this optimization earlier in this section (Sect. 3.1).

The primary kernel is launched from the host as follows.

$$For\_Kernel <<< 512, 256, 0, stream >>> (loop\_size, op); \tag{3}$$

The definition of the For_Kernel is shown using algorithm 10. For the GPU push-relabel algorithm discussed in Sect. 2.2, the kernel is launched with 512 blocks, where each block has 256 threads within it. So there are a total of 131072 threads that work on all the vertices of the input graph during a single cycle. If the number of vertices is more than 131072, then each GPU thread processes multiple vertices. We have found this kernel launch configuration to be empirically efficient in our setup. The number of cycles discussed in Sect. 2.2 is set to the number of vertices (V) in the input graph. Hence, For_Kernel shown is called V number of times. The kernel is passed with two arguments: `loop_size`, which is the number of vertices in the input graph, and op, which is the __device__ function.

The launched threads work on a chunk of vertices in a strided manner if there are more vertices than the total number of threads launched. Each thread calls the op(i) device function which performs the primary operation (either push or relabel) on the given vertex i (Line 4). The op() function uses the input graph G, the excess array, the height array, the residual array, the source vertex, and the sink vertex to perform the appropriate operation on its assigned vertex. The op() function gets executed on the GPU, and the thread that calls the op() function performs either the push or the relabel operation on that vertex if it is an active vertex. The above details also present the implementation of push_relabel kernel demonstrated as part of Line 8 in Algorithm 2.

Algorithm 10: For_Kernel Definition

```
   Input: ___device___ function 'op', variable 'loop_size'
1  STRIDE = blockDim.x * gridDim.x ;
2  i = blockDim.x * blockIdx.x + threadIdx.x ;
3  while i < loop_size do
4  |   op(i) ;
5  |   i += STRIDE ;
6  end
```

Our current implementation can be enhanced to exploit shared memory, whose investigation is left as a future work.

4 Maximum Flow in Dynamic Graphs

In this section, we discuss how to model dynamic maxflow efficiently without recomputing it on the modified network. We consider two dynamic updates: reducing and increasing edge capacities. Note that edge additions and removals can be easily simulated by updating edge capacities.

4.1 Decremental Update

Consider in a flow network, we reduce the capacity of an edge (u, v) from w to w'. Now if the original flow from u to v is less than or equal to w', we do not need to change anything and the maxflow in the modified network continues to be as it was before. On the other hand, if the flow is greater than the new capacity w'. In such a case, we need to reduce the flow in this edge. This can be done by changing the excess values of the vertices. Formally, we set

$$e(u) += f(u, v) - w' \tag{4}$$

$$e(v) += w' - f(u, v) \tag{5}$$

$$f(u, v) = w' \tag{6}$$

The vertex u has a positive excess since it is not sending all of its flow in the edge now and vertex v now has a deficit since it is not receiving all the flow it used to get before. Earlier, vertex u was sending $f(u, v)$ units of flow; post-update, it is sending only w' units to v. Now we run the push-pull relabel algorithm till all the excesses become zero.

Figures 3 and 4 illustrate the two cases of the decremental updates discussed above. Red edges undergo the decremental updates.

Figure 3 demonstrates the first case for the decremental update. The initial capacity of the edge (u, v) is 100, and the flow through the same edge is 2. Hence, its current capacity is 98, and the back edge (v, u) has capacity 2, which shows the amount of flow f(u, v). Now, if we reduce the capacity of the edge (u, v) from 100 to 50, the current capacity reduces from 98 to 48, but the flow passing through the edge (u, v) still remains the same, that is 2. Hence the final maxflow value for the graph remains unaffected.

Figure 4 demonstrates the second case for the decremental update. Here also, the current capacity of the edge (u, v) is 98, and the back edge from (v, u) has capacity 2. However, this time we reduce the capacity of the edge (u, v) from 100 (w) to 1 (w'). Now, as w' is less than f(u, v) = 2, we set the flow along the edge, i.e., f(u,v) to w', i.e., 1 based on the Eq. 6. This is shown by setting the weight of the back edge (v, u) to 1 and of edge (u, v) to 0. We then increment the excess flow value for the vertex u by f(u, v) - w' = 2 - 1 = 1 using Eq. 4 and then reduce the excess flow value for the vertex v by w' - f(u, v) = 1 - 2 = -1 using Eq. 5. The final graph in Fig. 4 (on the right) is labeled

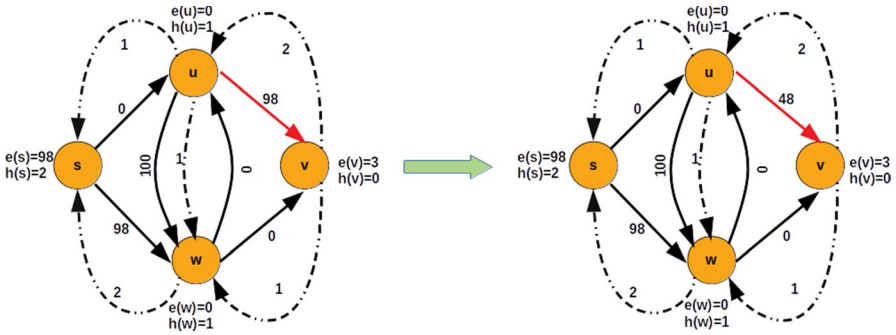


Fig. 3 Decremental Update Case-1 (The original flow from u to v is less than or equal to the reduced edge capacity)

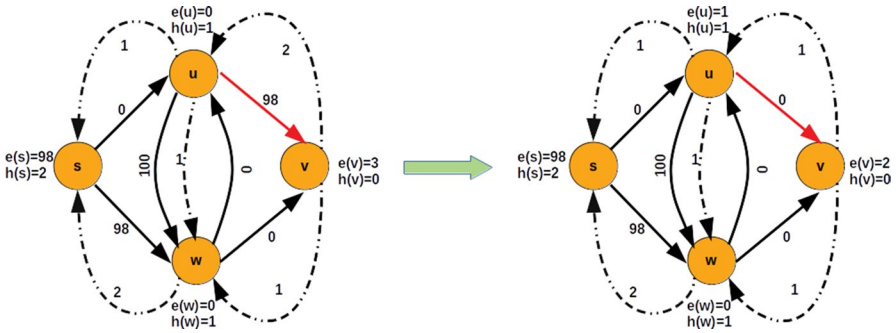


Fig. 4 Decremental Update Case-2 (The original flow from u to v is greater than the reduced edge capacity)

with each vertex's corresponding excess flow value. Now, if we run the push-pull relabel algorithm on this graph, the excess flow present on the vertex u will eventually be sent back to the source vertex (s), and the excess flow present over the target vertex (v) will be the final maxflow value, which is 2. Note that the final maxflow value obtained is correct after updating the weight of the edge (u, v) from 100 to 1.

4.2 Incremental Update

When we increase the edge capacity from w to w', we check if the original flow was smaller than the initial capacity w. If it was, then increasing the capacity further does not affect the optimal solution. On the other hand, if it was equal to the initial capacity w, we would like to check if we can send more flow through this edge. Hence we increase the flow to the new capacity w'.

$$e(u) += w - w' \tag{7}$$

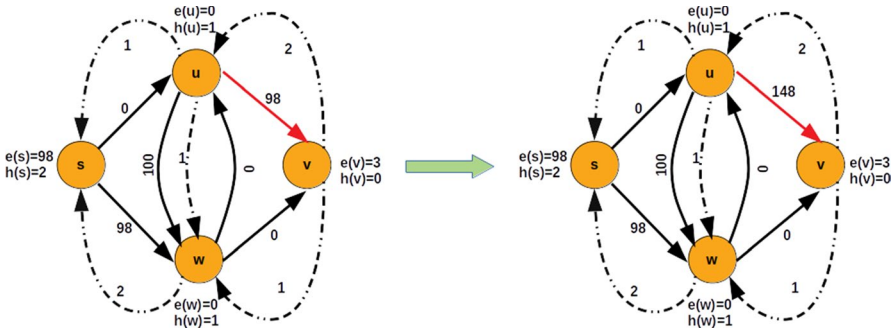


Fig. 5 Incremental Update Case-1 (The original flow from u to v is smaller than the initial capacity of the edge (u,v))

$$e(v) += w' - w \tag{8}$$

$$f(u, v) = w' \tag{9}$$

Vertex u has a deficit since it receives a smaller flow than what it sends to v, and vertex v now has an excess since it receives more flow than the one it used to receive before. Earlier, vertex u sent $f(u, v) = w$ units of flow; post-update it sends w' units to v. Now we run the push-pull relabel algorithm till all the excesses become zero.

Figures 5 and 6 illustrate the two cases of the incremental updates discussed above. Red edges undergo the incremental update.

Figure 5 demonstrates the first case for the incremental update. The initial capacity of the edge (u, v) is 100, and the flow through it is 2. Hence the current capacity of the edge (u, v) is 98, and the back edge (v, u) has the capacity of 2, which indicates the flow f(u,v). Now, if we increase the capacity of the edge (u, v) from 100 to 150, its current capacity increases from 98 to 148, but the flow passing through it still remains the same, which is 2. Hence the final maxflow value for the graph shown on the right in Fig. 5 remains unaffected.

Figure 6 demonstrates the second case for the incremental update. Here, the current capacity of the edge (u, v) is 0, and the back edge (v, u) has capacity 1, indicating the flow f(u, v). Now, we increase the capacity of the edge from 1 (w) to 10 (w'). Since the initial capacity w of (u, v) is same as the initial flow f(u, v), we set the flow along the edge (u, v), i.e., f(u,v) to w', i.e., 10 based on Eq. 9. This is shown by setting the weight of the back edge (v, u) to 10 and edge (u, v) to 0. We then decrement the excess flow value for the vertex u by (w - w') = (1 - 10) = -9 using Eq. 7 and then increment the excess flow value for v by (w' - w) = (10 - 1) = 9 using Eq. 8. The graph on the right in Fig. 6 is labeled with each vertex's corresponding excess flow value. Now, if we run the push-pull relabel algorithm on the graph shown in this modified graph, the excess flow present on v will be pushed to the sink vertex (t) along the edge (v, t), making the excess flow value of t as (3 + 9) = 12. Note that this is the final maxflow value considering u as the source and t as the sink, and this value is correct after incrementing the weight of the edge (u, v) from 1 to 10.

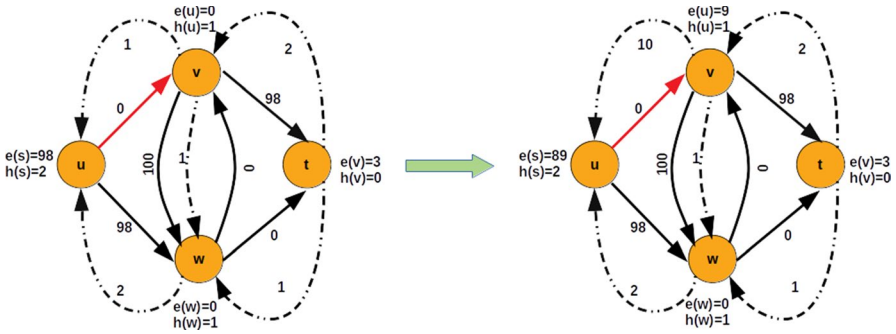


Fig. 6 Incremental Update Case-2 (The original flow from u to v is equal to the initial capacity of the edge (u,v))

5 Experimental Evaluation

In this section we evaluate the performance of our optimizations and approximation techniques vis-a-vis exact versions of the GPU parallel push-relabel algorithm.

Input Networks We evaluate the performance of our techniques on various types of networks such as the R-MAT network generated using GTgraph [13]. We also assess the performance of our techniques on the Genrmf networks generated using 1^{st} DIMACS Implementation Challenge [14]. In addition, we evaluate the performance of our techniques on many real work networks such as social networks, road networks, web graphs, etc. For unweighted graphs, we add an edge-weight as 1. Table 2 lists the various graphs used along with their sizes. Note that the complexity of Push-Relabel is high $O(V^2E)$ which resists exploring very large graphs.

Baseline Gunrock [11] is a state-of-the-art framework for implementing efficient graph algorithms on GPUs. We compare our optimization and approximate techniques with the exact implementation of GPU parallel maxflow from Gunrock.

Machine Configuration We perform experiments on a machine with an Intel Xeon E5-2640 v4 @ 2.4 GHz CPU having 64 GB RAM and Nvidia GeForce RTX 2080 Ti GPU having 4,352 cores spread across 68 SMXs with 11 GB memory. The machine runs CentOS 7 (64-bit).

5.1 Overall Results

Table 3 shows the overall speedup obtained by applying the techniques discussed in Sect. 3 on the dataset shown in Table 2. We observe that edge removal and vertex renumbering techniques work well on the flow networks (due to their skewed degree distribution). The vertex removal technique shows the performance improvement on the flow networks with a large number of zero in-degree and zero out-degree vertices. The excess cycle removal technique achieves performance benefits especially on large real-world flow networks. Reusing of min-height computation and memory access skipping work well on the synthetically generated Genrmf graphs.

Table 2 Network Dataset (d_avg and d_max are the average and the maximum degrees, while maxWt is the maximum edge-weight in the graph.)

t#	Network	#nodes (million)	#edges (million)	d_avg	d_max	maxWt
t1	road-belgium-osm	1.4	1.5	10	2	1
t2	roadNet-CA	2	2.8	12	2	1
t3	socfb-Harvard1	0.015	0.825	1K	109	1
t4	soc-lastfm	1.2	4.5	5K	7	1
t5	soc-livejournal	4	28	3K	13	1
t6	soc-orkut	3	106	27K	70	1
t7	soc-pokec	2	22	15K	27	1
t8	soc-youtube-snap	1.1	3	29K	5	1
t9	web-stanford	0.282	12	39K	16	1
t10	web-uk-2005	0.130	12	850	181	1
t11	web-wikipedia2009	2	5	3K	4	1
(t12–t24)	Genrmf	0.05-−0.5	0.2-−2.33	9	10	$14.8 * 10^5$
(t25–t28)	R-MAT	0.5-−2.0	30–80	60–100	5301–7057	99

Table 3 Overall Results (NA: Not Applicable)

Tech	Name	Speedup w.r.t. baseline	Inaccuracy	Evaluation Section
T1	Parallel BFS with frequent labelling	94.83	NA	5.2
T2	Reusing the Minimum height computations	1.05	NA	5.3
T3	Excess cycle removal	2.07	NA	5.4
T4	Vertex Removal	1.52	NA	5.5
T5	Edge Removal & Vertex Renumbering	1.09	∼0.0%	5.6
T6	Memory accessskipping	1.09	4.4%	5.7
T7	Push-Pull Relabel Algorithm	1.50	NA	5.8

Parallelizing BFS for global relabeling along with the push-pull relabel algorithm helps the maxflow computation converge quickly on the massive Genrmf graphs.

We delve deeper into the fine-grained effect of each technique, as well as their combined effect in the following subsections.

5.2 Effect of Parallel BFS with Frequent Relabeling (T1)

Figure 7 shows the performance comparison between three implementations: (i) baseline parallel with sequential BFS [11], (ii) with parallel BFS, and (iii) with parallel BFS performing frequent labeling. With the parallel BFS alone, we can observe a 3.93× speedup over the baseline. However, with parallel BFS and frequent labeling, we observe a significant improvement in the running time (94.83× speedup) of PR. This significant improvement is a combined effect of a considerable reduction

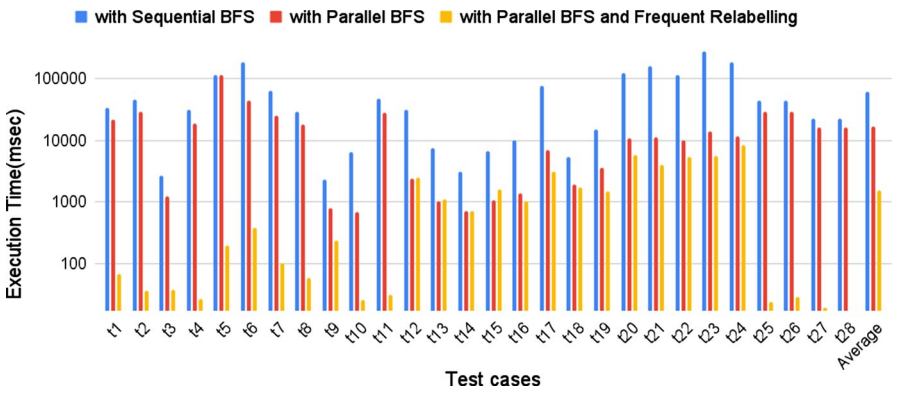


Fig. 7 Performance comparison of GPU-based maximum flow algorithm with (i) sequential BFS, (ii) parallel BFS, and (iii) parallel BFS with frequent relabeling

in the number of push+relabel operations, the total number of cycles taken for convergence in the push-relabel kernel, and the decrease in execution time of BFS in the global relabeling function. Figure 8 shows the total number of push and relabel operations performed by the algorithm with and without frequent relabelling. We observe that for the Genrmf graphs and a few real-world graphs (such as t3, t5, and t9), there is a significant reduction (20 times in many cases) in the total number of push+relabel operations, which directly translates to a reduction in execution time of the overall maxflow computation. However, for other real-world and RMAT graphs, the total number of push+relabel operations is smaller (smaller than 10). For these graphs, the primary reason for performance improvement due to Frequent Relabeling (FR) is the reduction in the number of cycles taken for execution of push-relabel kernel (10 to 500 times fewer), as shown in Fig. 9. Since executing fewer cycles necessitates more frequent relabeling in an iteration and performing more iterations in the push-relabel kernel function, we indirectly avoid running unnecessary cycles in the kernel. The secondary reason for performance improvement is the time saving done by parallelizing the BFS in the global_relabel_op function, which is shown in Fig. 10.

5.3 Effect of Reusing the Minimum Height Computations (T2)

Figure 11 shows the time taken by a parallel push-relabel algorithm with and without re-using the minimum height computations. Note that this technique does not have any preprocessing cost associated with it.

The frequency of cycles that consist of only the push operations (S) along with valid min-heights (T) are presented in Fig. 12. For Genrmf flow networks, this value ranges between 1K and 463K. Hence, we observe that the GPU parallel version's performance, which reuses the min-height computations, is consistently better than Gunrock's on Genrmf. In contrast, this value for most of the real-world networks and RMAT graphs is close to 0. Hence, we did not observe any noticeable performance improvement (1.05×).

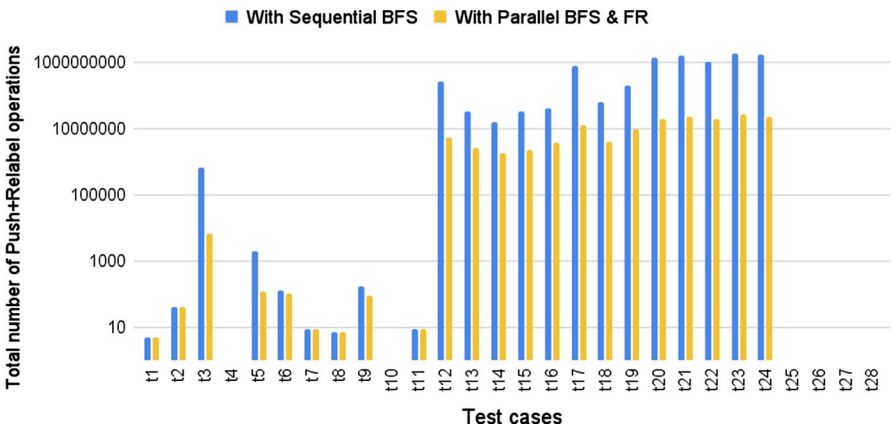


Fig. 8 Total number of push and relabel operations performed with Sequential BFS and with parallel BFS and Frequent Relabeling(FR)

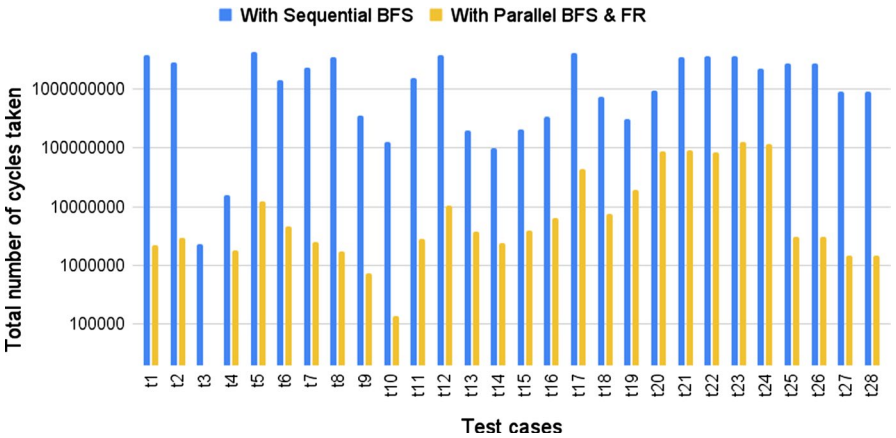


Fig. 9 Total number of cycles taken for convergence with Sequential BFS and with parallel BFS and Frequent Relabeling(FR)

Takeaway 1 If the frequency of push cycles is high, then reusing the minimum height computations achieves good performance improvement on the corresponding flow networks.

5.4 Effect of Removing Excess Cycles (T3)

Figure 13 shows the time taken by a parallel push-relabel algorithm with and without excess cycle removal. Note again that this technique is applied to the algorithm itself, and hence it does not have any associated preprocessing cost.

As discussed in Sect. 3.3, excess cycle removal (if any) will always be from the last iteration of the algorithm's execution. During the experimentation, we observed

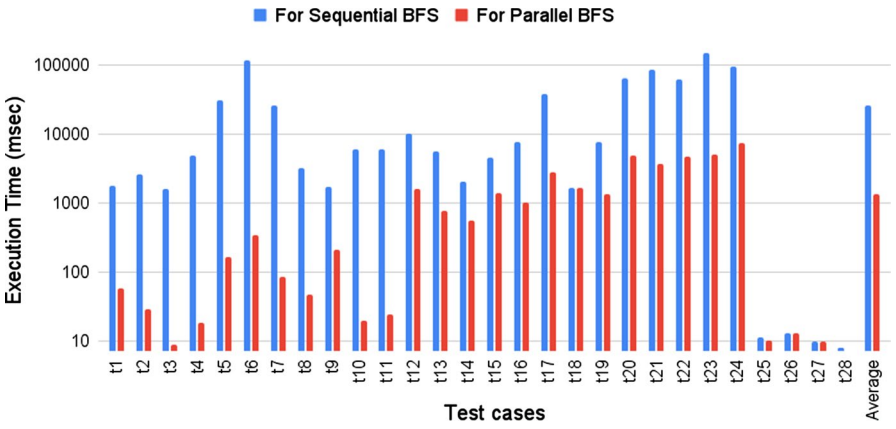


Fig. 10 Performance comparison between sequential and parallel BFS

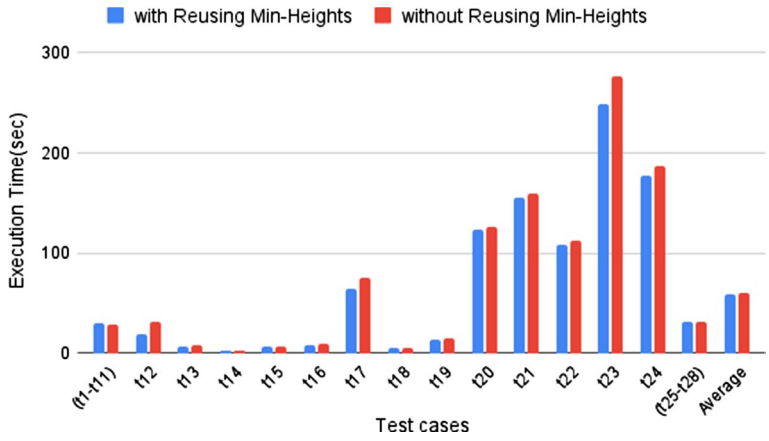


Fig. 11 Effect of reusing the minimum height computations

that the baseline algorithm converges quickly within a single iteration for most real-world and RMAT flow networks. Also, during that single iteration, this technique can prune between 1×10^8 to 4×10^9 cycles. This is presented in Figs. 14 and 15. Due to such a high number of cycles being pruned for real-world and RMAT flow networks, we observe that our version consistently outperforms Gunrock on these graphs.

In contrast, we observe limited benefits due to T3 on Genrmf networks. We observe that for a few Genrmf flow networks (2–5), the number of excess cycles pruned in the last iteration is less than 10^6 (Fig. 14). Hence, there is a little benefit due to this technique on these graphs. For the rest of the Genrmf flow networks (1 and 6–13), excess cycles detected in the last iteration are between $1 * 10^7$ and $0.5 * 10^9$ (Fig. 14). However, most of these graphs take more than 30

Fig. 12 Frequency of cycles where min-heights can be reused

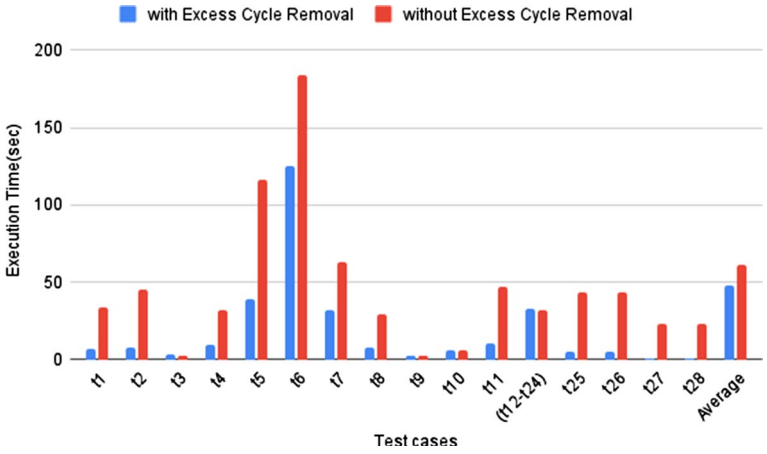


Fig. 13 Effect of removing the excess cycles

iterations to converge (Fig. 15). Due to this, the total number of cycles executed in the first n-1 iteration for the convergence is much greater than the cycles pruned in the last iteration using this technique. Thus, this technique offers limited performance benefits to them too. Overall, across all graphs, we observe a geomean speedup of 2.07× on the entire data-set by using this technique.

Takeaway 2 Excess Cycle Removal provides notable performance improvements for the flow networks converging quickly in a few iterations.

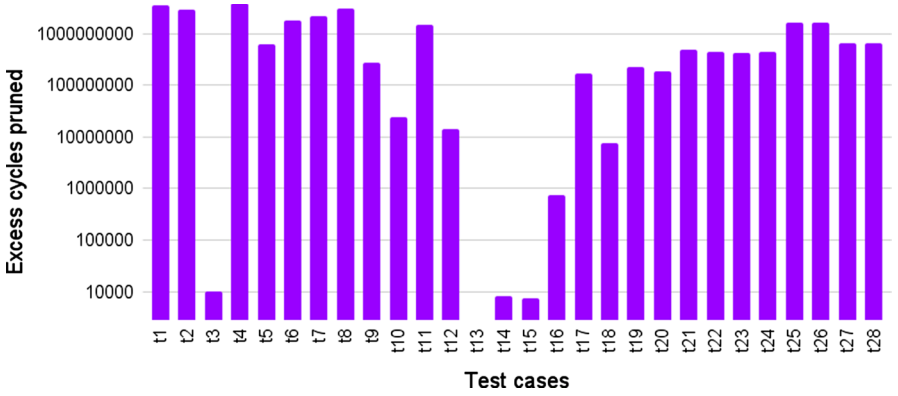


Fig. 14 The number of excess cycles pruned in the last iteration (note the logscale)

Fig. 15 Number of iterations taken by a parallel algorithm to converge

5.5 Effect of Vertex Removal (T4)

Figure 16 summarizes the experimental results obtained by applying the vertex removal on our dataset. As discussed in Sect. 3.4, this technique does not affect the maxflow since it removes the vertices with zero in-degree or zero out-degree or both. Note that the vertex renumbering time is not included in the speedup calculation, as it is not implemented as a part of the algorithmic processing but done as a one-time preprocessing step. However, the preprocessing time taken for vertex removal for each graph is shown in the same Fig. 16 for reference. We observe that the preprocessing time for the vertex removal is between 1% to 900% of the total execution time for various graphs in our dataset. However, the average preprocessing time of vertex removal is lesser than the average execution time of the baseline algorithm (both with and without vertex removal).

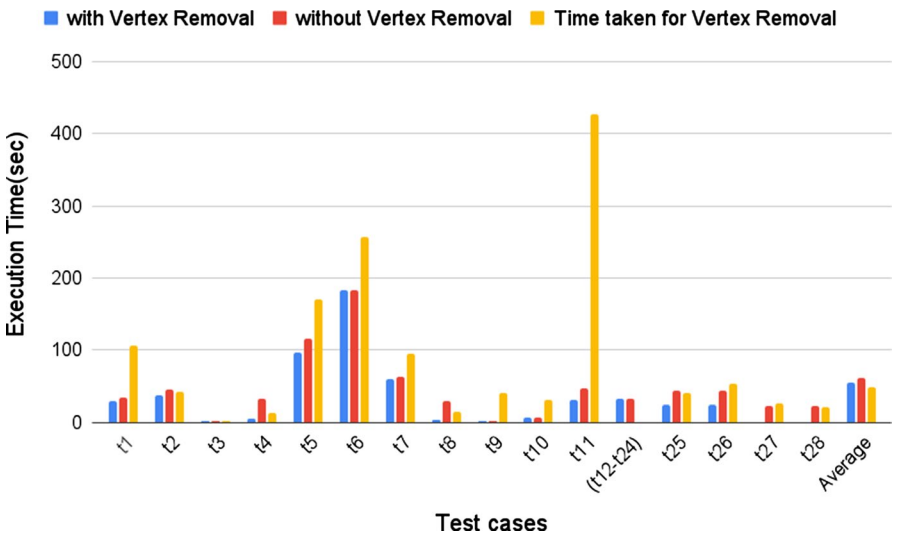


Fig. 16 Effect of vertex removal

As shown in Fig. 17, we observe that many real-world and RMAT flow networks have a large amount(between 50K to 900K) of zero in-degree and zero out-degree vertices, whereas the number of zero in-degree and zero out-degree vertices for Genrmf graphs is zero. By pruning such vertices, we can reduce the size of the `e` and `h` arrays specified in Algorithm 2. By doing so, we also save time for inter-device data transfer. By removing such vertices, we observe that the push-relabel algorithm works 52% faster on average than running it on the original flow networks of the dataset.

Takeaway 3 Vertex removal helps reduce the execution time for the graphs with a high number of zero in-degree and out-degree vertices.

5.6 Effect of Edge Removal and Vertex Renumbering (T5)

We analyze the effect of removing the edge attached between the vertices with a high degree on several real-world as well as synthetic flow networks. As explained in Sect. 3.5 the edge removal technique has two tunable parameters. Figures 18 and 19 show the effect of changing the values of `wt` and `th` on the various flow networks. Note that the wt is the weight with which the average degree is multiplied, and th is the threshold on the edge stay value of a particular edge. Figure 18 depicts how a variation in `wt` affects performance. For smaller values of `wt`, the value of weighted_avg_degree computed as per Algorithm 5 is lower, and hence the edge_stay value will be close to 0 for most of the edges even if they are connected to the vertices with the slightly skewed degree. Hence if `wt` values are low, more edges will be removed from the flow network. This results in a higher speedup. However, it comes at the cost of accuracy. On the other hand, when we set `wt` to a high value,

Fig. 17 Number of vertices removed (zero for all the Genrmf graphs)

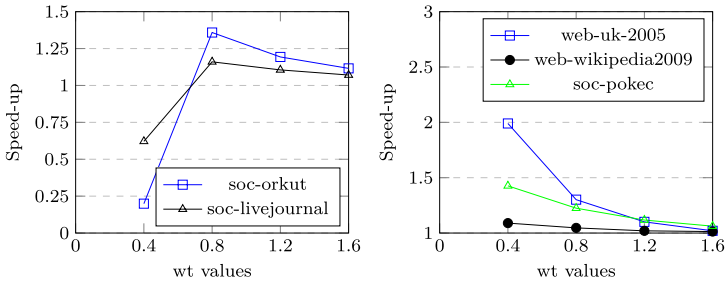


Fig. 18 Flow-network-wise effect of parameter `wt` for `th`=0.7 during edge removal

the reverse happens. Similarly, when the value of `th` is high (close to 1.0), most of the edges have their edge_stay values smaller than `th`; hence more edges get removed from the flow network. This also improves performance trading off accuracy. The reverse happens when the `th` value is close to 0.0.

Figure 20 shows the effect of applying the edge removal and vertex renumbering on the flow networks of the dataset. We observe that both the techniques together provide a geomean speedup of 1.09× on the data set with a 0% loss in the accuracy of the maxflow value. As explained in Sect. 3.5, edge removal normalizes the skewed degree distribution by removing the edges from the high degree vertices. Similarly, vertex renumbering techniques renumber the vertices such that vertices with a similar degree are assigned the nearby ids. Hence, both techniques also help in reducing thread divergence along with compressing the flow network. Edge removal, if done aggressively, bears the potential to add higher inaccuracy to the maxflow results. However, vertex renumbering does not impact the maxflow results as it preserves the flow network's structure. We observe that the combined preprocessing time taken for both edge removal and the vertex renumbering for most of the

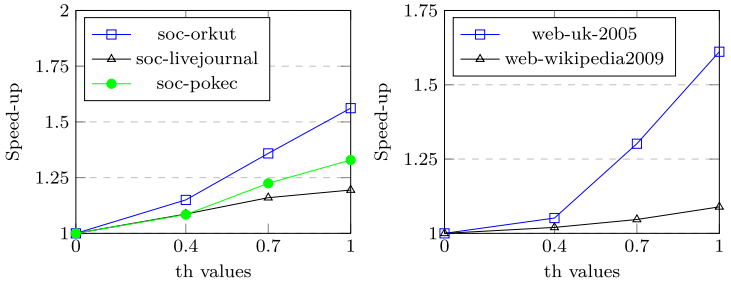


Fig. 19 Flow-network-wise effect of parameter `th` for `wt`=0.8 during edge removal

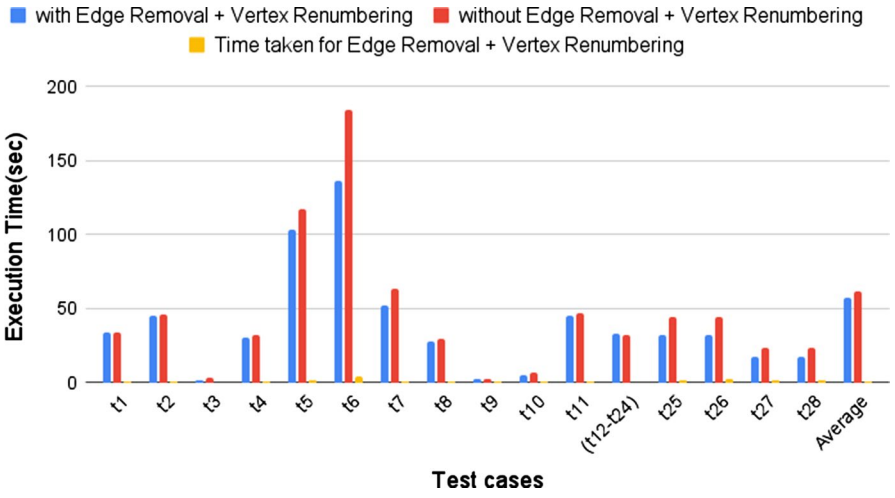


Fig. 20 Effect of Edge removal and Vertex renumbering

flow networks of the dataset is less than one second, and the maximum preprocessing time taken is 4 s. The preprocessing time taken for each of the graphs of the dataset is shown in the same Fig. 20. Hence, both techniques provide end-to-end speedup as the preprocessing overhead incurred is minimal.

Takeaway 4 Both edge removal and vertex renumbering helps in reducing thread divergence in the push-relabel GPU computation.

5.7 Effect of Memory Access Skipping (T6)

Figures 21 and 22 respe ctively present the performance and the inaccuracy analysis of memory access skipping on our dataset. We observe a geomean speedup of 1.088× over the baseline with an average inaccuracy of 4.4%. From Fig. 21 we observe that the approximate PR algorithm with controlled memory skips consistently outperforms the exact PR. Figures 23 and 24 show the effect of varying the

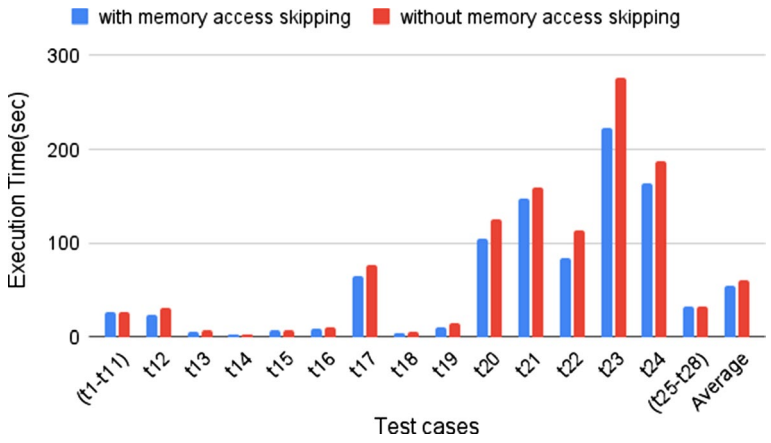


Fig. 21 Effect of memory access skipping on performance

threshold *t* on the inaccuracy and speed respectively, on the differently-sized Genrmf graphs. Note that t is the threshold on the excess flow value. Thread skips processing a particular vertex assigned to it if the excess flow value associated with that vertex is below a threshold t. For higher values of *t*, the speedup is higher, as the algorithm processes a smaller excess value at the vertices, leading to fewer accesses to GPU's global memory. As evident, this performance improvement comes at the cost of some accuracy. The reverse holds for smaller values of the threshold.

Takeaway 5 Memory access skipping helps in reducing the costly accesses to GPU's global memory in exchange for a small inaccuracy.

5.8 Effect of Push-Pull Relabel Algorithm (T7)

We summarize our results for the push-pull relabel (PPR) algorithm using Fig. 25. We did not observe any performance improvements on the real-world networks using PPR. However, it showed a noticeable performance improvement on the synthetically generated Genrmf and RMAT graphs (1.5×) with no inaccuracy in the maxflow values.

5.9 Maximum Flow in Dynamic Graphs

Dynamic graph algorithms involve edge addition and removal. In the case of maxflow, dynamism can be simulated by updating the edge-capacities (including value 0). In this work, we deal with capacity updates. We used the same synthetically generated genrmf graphs and changed the capacities of the edges present in the graph with values selected from a uniform distribution. For timing the algorithm processing, we generated 10 batch files where each file has 1000 edge updates. We assigned a new capacity by selecting a random number from the range 0..200. Once the

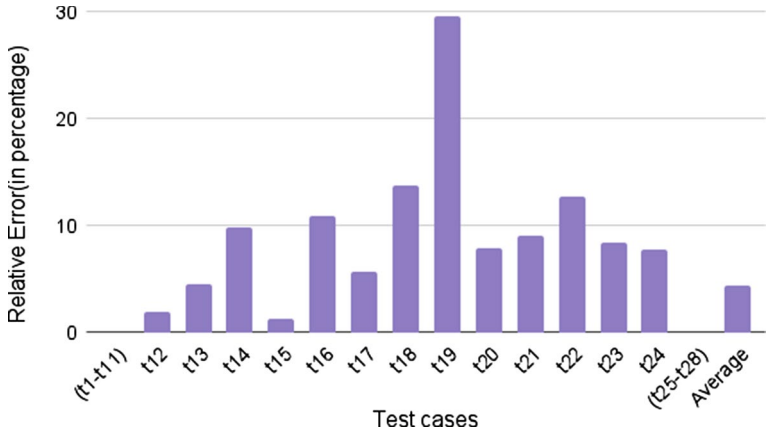


Fig. 22 Effect of memory access skipping on inaccuracy

Fig. 23 Flow-network-wise effect of varying the threshold t during memory access skipping on the inaccuracy

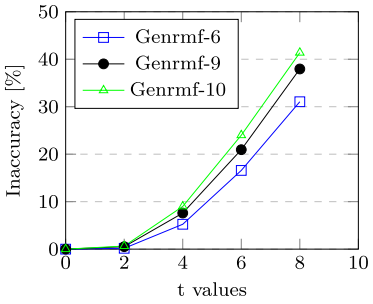
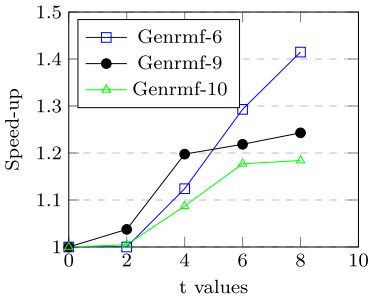


Fig. 24 Flow-network-wise effect of varying the threshold t during memory access skipping on the speedup



(static) algorithm converges on the original graph, the dynamic updates are applied in parallel for one batch file. Different batches are handled in sequence. For the i^{th} batch file, we measure the *cumulative* time from the point it converged on the original graph to the point it converged for the i^{th} batch file.

Figure 26 shows the effect of dynamic processing against the static one on the six Genrmf graphs. The horizontal lines in both the plots are the time taken for the static processing for the respective Genrmf graph. The two plots indicate us that for

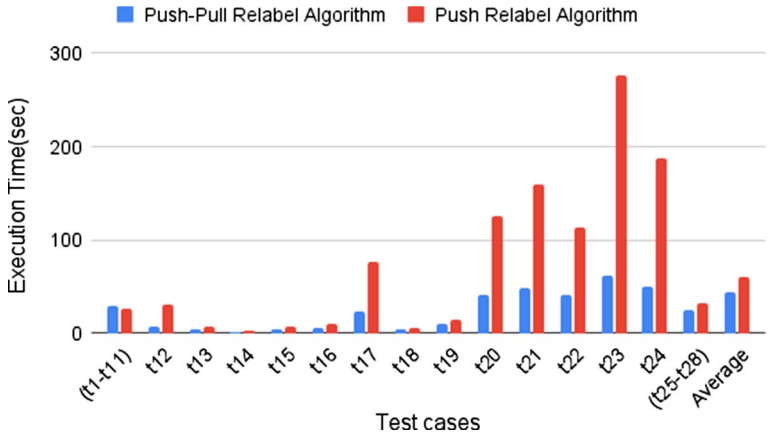


Fig. 25 Effect of push-pull relabel Algorithm on performance

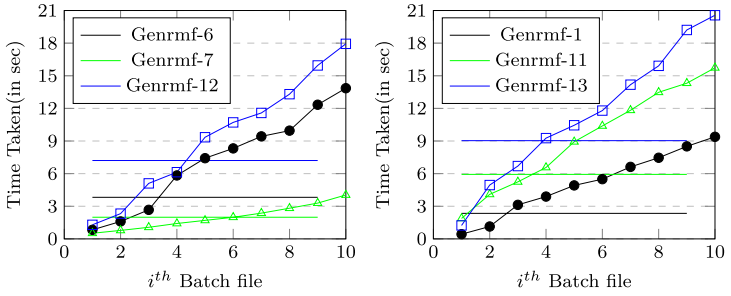


Fig. 26 Effect of dynamic processing against the static processing on Genrmf graphs

several updates, the dynamic processing is attractive. However, beyond a point, the cost of updating maxflow becomes higher than running the static computation on the modified graph. This threshold point differs across graphs as it depends upon the graph structure, edge capacities, as well as the updates themselves. It is important to note that since maximum flow is a global property of the graph, and not a local one, changing capacities of a few edges may also lead to substantial changes in the flow structure.

5.10 Putting it All Together

This section explores the effect of combining all our techniques. Figure 27 shows the performance improvements attained after combining all the techniques from T1 to T7 along with proper tuning of the knobs where applicable. The combined techniques exhibit a geomean speedup of 96.62× over our entire dataset with 0% inaccuracy. We observe that the average execution time per graph for the baseline implementation is 61.172 s, while that with our proposed techniques is 1.563 s.

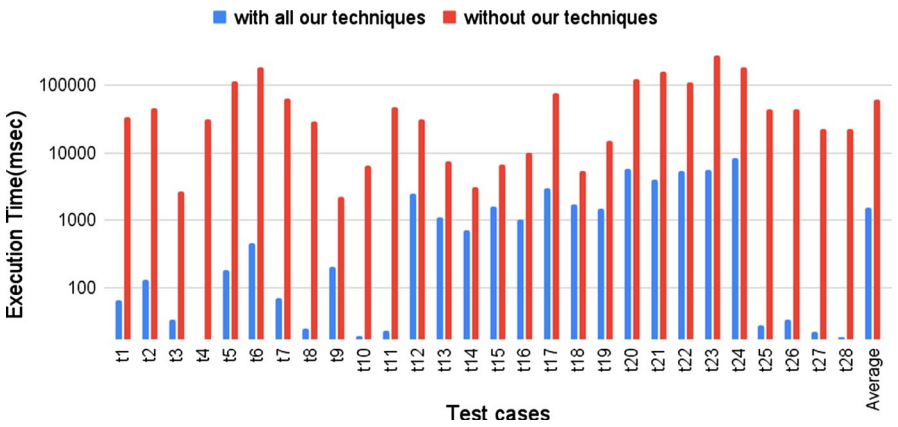


Fig. 27 Effect of all the merged techniques on performance

6 Effect on Applications

We evaluate our proposed techniques on the three maxflow applications. The first application, i.e., the maximum bipartite matching, is part of the current section, while rest two applications are present in "Appendix A".

6.1 Application 1: Maximum Bipartite Matching

A bipartite graph G(V, E) refers to the graph whose vertices can be divided into two disjoint sets S1, S2 where S1, S2 $\subseteq$ V and S1 $\cup$ S2 = V, and every edge $\in$ E connects two vertices such that one is in set S1 and another in set S2. Figure 28 (left) presents an example of a bipartite graph with eight nodes. The first four nodes (1 to 4) belong to the set S1, and the last four nodes (5 to 8) belong to the set S2. A matching in a bipartite graph is a set of the edges chosen in such a way that no two edges share the same vertex. For example, in the bipartite graph shown in Fig. 28 (left), the set of edges {(4, 5), (3, 6)} forms one of the possible matchings. A maximum bipartite matching maximizes the number of such edges. In the bipartite graph shown in Fig. 28 the set of edges {(4, 5), (3, 6), (2, 7), (1, 8)} forms a maximum bipartite matching, and the cardinality of this set is 4. The objective of our application is to find the cardinality of the set of edges forming the maximum bipartite matching for a given bipartite graph.

The maximum bipartite matching problem can be solved by converting it to the maximum flow problem. To achieve this, we add one extra node as a source (S) and connect it to all the nodes with the first set S1 of the given bipartite graph. Similarly, we generate one extra node as a sink (T) and connect it to all the nodes of the second set S2. The modified graph is shown in Fig. 28 (right). Once we have constructed the flow network as stated above, we run the push-relabel algorithm on it. The obtained maximum flow value corresponds to the size of the maximum bipartite matching on the given bipartite graph (due to the MaxFlow-MinCut theorem [15]). The execution

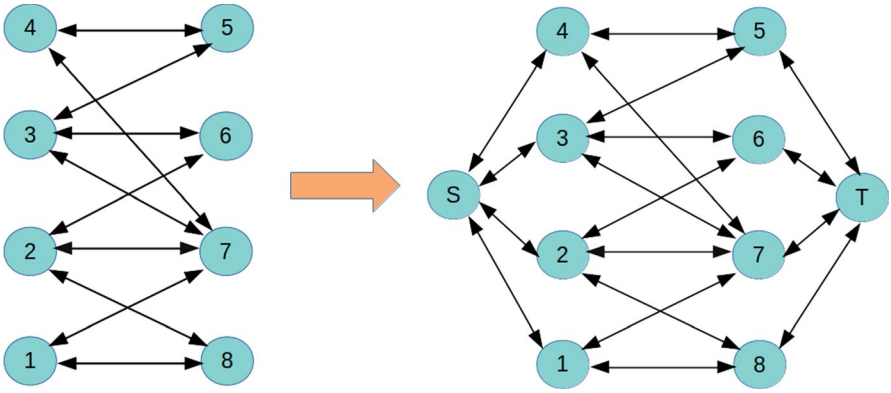


Fig. 28 Example bipartite graph and its corresponding flow network

of the push-relabel algorithm on the graph shown in Fig. 28 (right) with `S` as the source vertex and `T` as the sink vertex outputs 4 as the maximum flow value which corresponds to the size of the maximum bipartite matching (note that there could be multiple optimal solutions).

Since not all the graphs are bipartite, for this application, we use the designated bipartite graphs available at Konect [16] as shown in Table 4. We preprocess the existing bipartite graphs to add a source vertex S, a sink vertex T, and the edges connecting these two vertices with the two sets S1 and S2. After running the push-relabel algorithm on this expanded network considering S and T as source and sink, we obtain the size of the maximum bipartite matching, using both the baseline and our optimized processing.

We use two criteria for evaluating our techniques: speedup and inaccuracy. We name the baseline maximum flow algorithm as Algorithm A. We apply all our techniques discussed in Sect. 3 to the baseline, naming the modified algorithm as Algorithm B. We consider the maxflow value generated by Algorithm A to be the actual maximum flow and observe the loss in accuracy of the maxflow values computed by Algorithm B. Figure 29 shows the relative error in the maximum flow computation by Algorithm B. From Fig. 29, we observe that the maximum flow value generated by Algorithm B is very close to that of Algorithm A. On an average, we observe an accuracy loss of 0%.

Figure 30 compares the performance of the two algorithms. We observe that the average execution time for Algorithm A is 163.489 s, while that for Algorithm B is 692.888 s. On an average, we observe a geomean speedup of 14.29× due to our proposed techniques, clearly indicating their benefit.

7 Related Work

We divide the relevant related work in sequential maxflow, parallel maxflow, and approximate computing techniques.

Table 4 Bipartite graphs dataset

t#	Network	#nodes (million)	#edges (million)
t1	Actor movies	0.51	1.47
t2	IMDB	0.87	2.71
t3	BookCrossing (implicit)	0.44	1.15
t4	TV Tropes	0.15	3.23
t5	YouTube	0.12	0.29
t6	Occupations	0.22	0.25
t7	Producers	0.18	0.20
t8	vi.sualize.us user–item	0.51	2.29
t9	vi.sualize.us tag–item	0.57	2.29
t10	Teams	0.93	1.36
t11	Genres (DBpedia)	0.26	0.46
t12	Github	0.17	0.44
t13	DBpedia locations	0.22	0.29

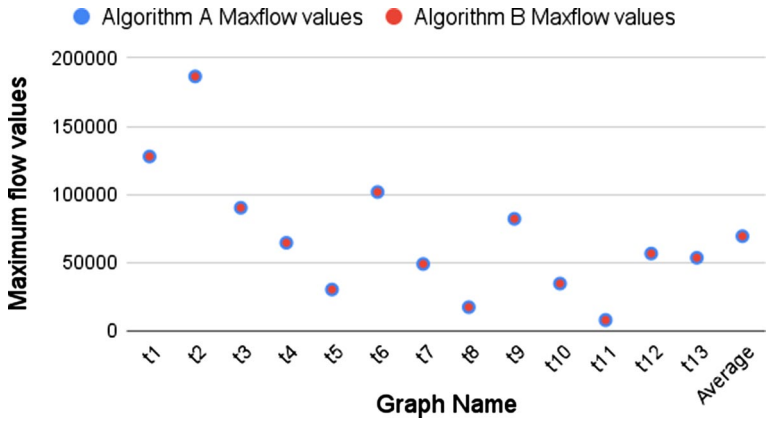


Fig. 29 Accuracy comparison of Algorithms A (baseline) and B (optimized with our techniques). Note that their maxflow values are almost the same and overlap

7.1 Sequential Maximum Flow Computation

The first maximum flow algorithm was proposed by Ford and Fulkerson [17], which used the concept of augmenting paths from source to destination to calculate the maximum flow. This algorithm was further improved by Edmonds and Karp [6] by using a breadth-first search to find the shortest augmenting paths from source to sink for sending the flows. The crucial insight was that the asymptotic time complexity of $O(VE^2)$ could be achieved by sending flow across the shortest augmenting paths (where V is the number of vertices and E is the number of edges). Dinitz and Yefin [3] proposed the concept of layered networks for finding all the shortest augmenting paths in a single step. Their proposed algorithm consists of a phase for finding

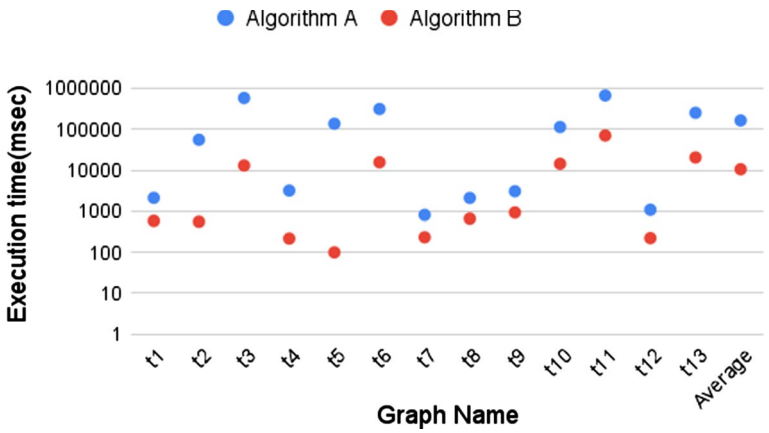


Fig. 30 Performance comparison of Algorithms A (baseline) and B (optimized)

a layered network followed by finding a maximum flow on the layered network and improving the original flow. They showed that the maximum number of phases for their proposed algorithm could be $n-1$, where n is the number of vertices in the input network. The computation of the maximum flow using the concept of layered networks improved its asymptotic time complexity to $O(V^2E)$. The maximum flow algorithm proposed by Karzanov [4] introduced the concept of pre-flows and the push operation, which led to an $O(V^3)$ algorithm. Later, the push-relabel algorithm was proposed by Goldberg and Tarjan [8] with the asymptotic time complexity of $O(V^2E)$. The complexity bound for Goldberg and Tarjan's algorithm was further improved by using various techniques [7, 8].

7.2 Parallel Maximum Flow Computation

Several implementations of parallel maximum flow algorithms have also been proposed for multi-core and many-core platforms. Anderson and Setubal proposed the efficient parallel implementation of push-relabel algorithm for the shared-memory multi-processor [2]. They introduced the method for incorporating the concurrent global relabeling heuristic in the parallel push-relabel algorithm. They also introduced a data structure for their proposed concurrent implementation which can dynamically change the size of the tasks assigned to a particular thread based on the total amount of work available. They demonstrated that their parallel implementation gives good speedups on the machine known as Sequent Symmetry systems for various input graphs. Bader and Sachdeva proposed a cache-aware optimization of Anderson and Setubal's approach. They also provided the design and implementation of a new shared-memory parallel algorithm for the gap relabeling heuristic [1].

Vineet and Narayanan presented the faster parallel implementation of the push-relabel algorithm for performing the graph cuts on the GPUs. They demonstrated that their GPU parallel implementation performs over 60 graph cuts per second on 1024×1024 images [18]. Bo Hong presented an atomics-based lock-free

multi-threaded algorithm for the multi-core architectures [19]. They demonstrated that the lock-free property is enabled by the re-designed push and relabel operations. They also showed that multi-processor architecture supporting atomic read-update-write operations could execute their proposed multi-threaded algorithm without using any locks. Later, Zhengyu He and Bo Hong proposed the first generic parallel push-relabel algorithm for CUDA devices [5]. Their proposed push-relabel algorithm exploits the large number of available processor cores of the GPU and adaptively switches between the CPU and GPU during the course of the execution to maximize the execution efficiency. They claimed their algorithm to be up to 2 times faster than the push-relabel algorithm proposed by Goldberg and Tarjan.

7.3 Approximate Computing Techniques

The survey of approximate computing techniques is given by Mittal [20]. The survey discusses strategies for finding approximable program portions and monitoring the output quality using approximate computing in different processing units, processor components, memory technologies, as well as programming frameworks. Besta et al. [21] developed the framework for lossy graph compression for approximating the graph processing. They introduced the compression kernels, which allows the users to express and implement several graph compression schemes. They also propose the accuracy metrics for accessing the quality of lossy compression. They evaluated different compression schemes in terms of reduction in graph size, accuracy, and performance, and showed that the framework could become a platform for designing and analyzing lossy graph compression methods facilitating approximate graph processing, storage, and analytics.

Singh and Nasre [22] proposed several techniques to improve performance of graph applications on GPUs using approximate computing. They improved memory coalescing and reduced thread divergence for error-tolerant graph applications on GPUs. Singh and Nasre [23] also discussed several device-independent and algorithm-independent techniques to add controlled approximations to graph algorithms to improve their efficiency. They discussed four instances of their proposed model, which, when implemented in the graph algorithms, provide the approximation in the algorithm's results based on tunable knob. They showed that proper adjustments of the tunable knobs could provide good performance gains at the cost of some inaccuracy.

8 Conclusion and Future Work

In this work, we studied the effect of various optimization and approximation techniques for improving the Push-Relabel algorithm's efficiency on GPUs. Our techniques improve the GPU parallel Push-Relabel algorithm's performance by pruning the unnecessary computations, reusing the prior computations, reordering and compressing the flow network, using parallel BFS with frequent relabeling over sequential BFS, etc. In the end, we also proposed the push-pull relabel algorithm

and demonstrated how it could be used to compute the maxflow on the dynamically changing graphs. We expect the takeaways listed in Sect. 5 to be helpful in deciding when to apply the particular optimization and approximation technique to improve the GPU parallel push-relabel algorithm's performance. We also demonstrated that our proposed techniques compute the maximum flow value around 1.05× to 94.83× faster than the exact GPU parallel push-relabel algorithm on the several real-world and synthetic flow networks while preserving the maximum flow value.

We further plan to extend our research in the following directions. First, analyze the speedup accuracy trade-off of the techniques proposed in Sect. 3 on the other maximum flow algorithms apart from the push-relabel algorithm. Second, we plan to investigate the algorithm-specific optimization techniques for the different maximum flow algorithms like Edmond-Karp and Dinic's algorithms to compute the maximum flow value faster.

Appendix A: Additional Maximum Flow Applications

A.1 Application 2: Team Elimination

Another application of maxflow is Team Elimination. In this use case, given a set of teams that play together in a league format, the objective is to find out if a given team is eliminated or not at any point in time. At any given moment, the i^{th} team has w[i] wins and g[i][j] games left to play against team j. A team is eliminated if it cannot possibly finish the season in the first place or tied for the first place. Our goal is to determine which teams are eliminated. Consider an example [24] presented in Table 5, having four teams. The columns indicate the number of wins, losses, the number of remaining games, and against whom the games would be played. Team-1 already has 83 wins (which is the highest so far) and has 8 games left (1 against Team-2, 6 against Team-3, and 1 against Team-4). Team-1 cannot be eliminated. Team-4 can finish with at most 80 wins (77 current wins + 3 games remaining). But Team-1 already has 83 wins. Hence, Team-4 can be eliminated. A more complex scenario occurs for Team-2. Team-2 can finish the season with as many as 83 wins, which appears to be enough to tie with Team-1. But this would require Team-1 to lose all of its remaining games, including the six against Team-3, in which case Team-3 would finish with 84 wins. Hence, Team-2 can also be eliminated.

Team Elimination can be reduced to maxflow. To check if a particular team x is eliminated or not, we create a flow network corresponding to that team and then run the maximum flow algorithm on it. In the generated flow network for a given team x, we have one node for each pair (i, j) denoted as i-j, of teams which are called *game nodes*, excluding team x. Similarly, we have nodes corresponding to each team except team x, which are called *team nodes*. We connect an artificial source s to each game node i-j and set its capacity to g[i][j] (number of games left for i to play against j). We then connect each game node i-j with the respective team nodes i and j and set the edge's capacity to infinity (INF as shown in Fig. 31). Finally, we connect each team node to an artificial sink t. We include an edge from i^{th} team node to the sink with capacity w[x] + r[x] - w[i], where r[x] is the number of remaining

Table 5 An example for the Team Elimination application [24]

Team-No	Wins	Losses	Left	Against			
				Team-1	Team-2	Team-3	Team-4
Team-1	83	71	8	–	1	6	1
Team-2	80	79	3	1	–	0	2
Team-3	78	78	6	6	0	–	0
Team-4	77	82	3	1	2	0	–

games for team x. Based on the above explanation, the flow network generated for Team-4 given in the Table 5 is shown in Fig. 31. Note that the flow network generated for any team will always have four layers where Layer 1 has only the source, Layer 4 has only the sink, and layer 2 and layer 3 have $(n-1)(n-2)/2$ and $(n-1)$ vertices respectively, where n is the total number of teams.

To check if the team is eliminated or not, we check if the maximum flow in the generated network saturates the arcs leaving the source vertex s. If the maximum flow does not saturate all the arcs leaving s, then there is no scenario in which team x wins the division. Hence such a team can be eliminated. For the example graph shown in Fig. 31, maxflow from s to t is 2, and it does not saturate all the outgoing edges from s. This is because the sum of the edge weights of all the outgoing edges from s is 7. So Team-4 can be eliminated.

A.1.1 Push Relabel Algorithm on Team Elimination Application

We experimented with real-world leagues [25]. However, they have a maximum of 25–30 teams, leading to small maxflow networks. Hence, computing maxflow for these cases was extremely fast. Therefore, it was statistically not useful to compare

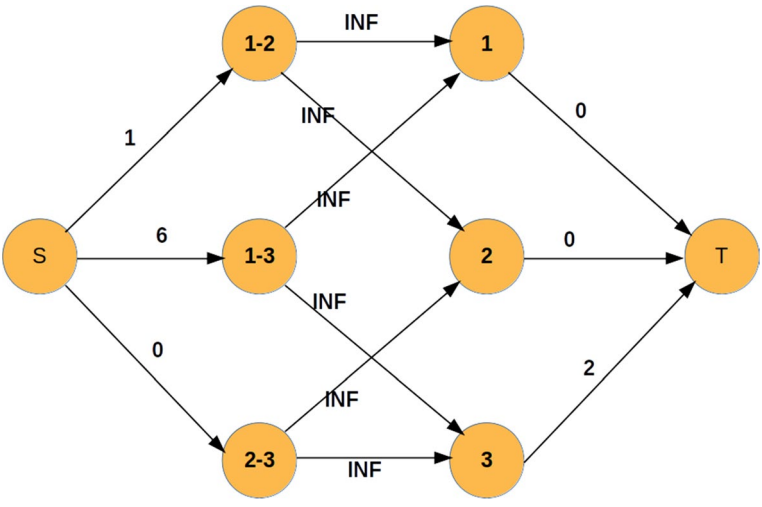


Fig. 31 Constructed flow network for Team-4 (x = 4)

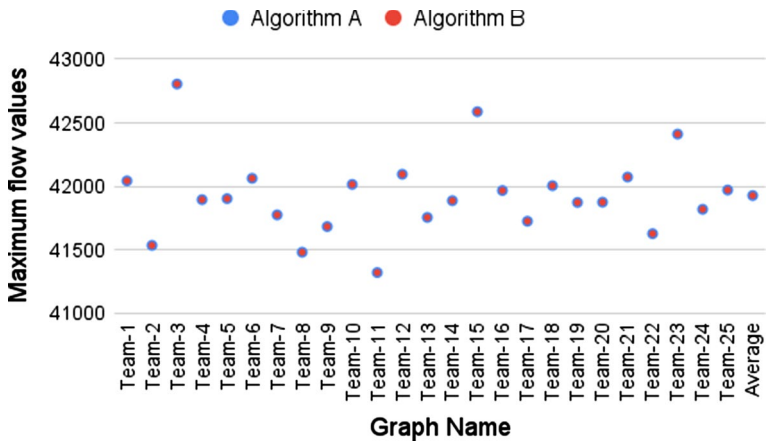


Fig. 32 Accuracy comparison of Algorithms A (baseline) and B (optimized). Note that their maxflow values are almost the same and overlap

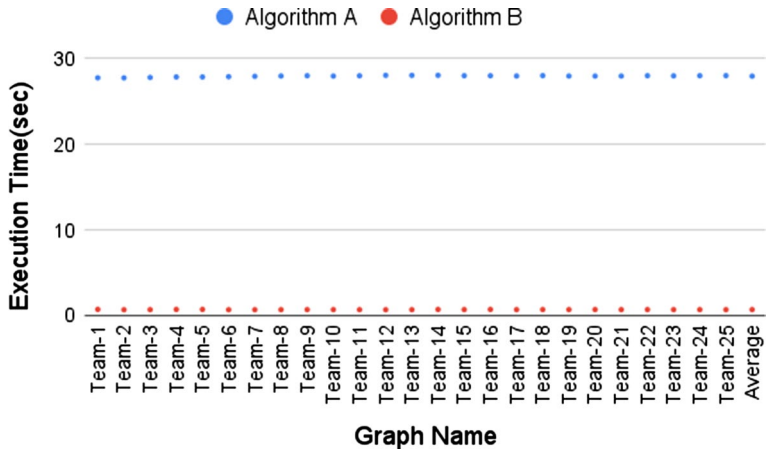


Fig. 33 Performance comparison of Algorithms A (baseline) and B (optimized)

the non-optimized and the optimized versions. Therefore, we resorted to generating larger datasets. We synthesized data for a league that has 1500 teams with a random distribution. This led to the creation of 1500 graphs, each of size approximately 1 million vertices and 3 million edges. Out of these 1500 graphs, we ran the algorithm on a random set of 25 graphs and collected the results. For the purpose of evaluation, we call the baseline Push Relabel algorithm as Algorithm A (exact) and the Push Relabel Algorithm with our set of techniques applied as Algorithm B (approximate). We apply all our techniques discussed in Sect. 3 on the top of Algorithm A, naming the modified algorithm as Algorithm B.

We use two criteria for evaluating our techniques: inaccuracy in Fig. 32 and execution time in Fig. 33. From Fig. 32, we can observe that the maximum flow value

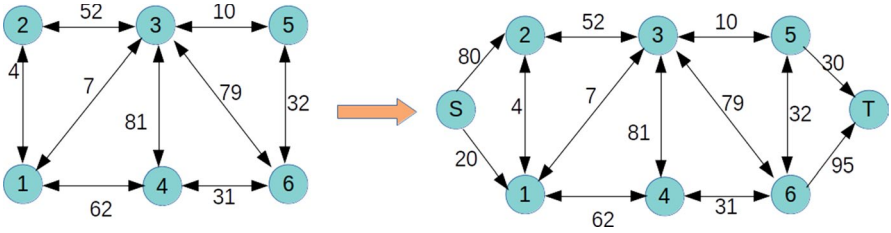


Fig. 34 Example road network and its corresponding flow network

generated by Algorithm B is very close to that by Algorithm A. On an average, we observe an accuracy loss of 0%. Also, we observe that the average execution time for Algorithm A is 27.951 s, while that for Algorithm B is 0.692 s. On an average, Algorithm B achieves a geomean speedup of 40.4×, clearly indicating the benefits of our proposals.

A.2 Application 3: Supply Demand Problem

The third application is supply–demand in road networks. Given a road network, each vertex represents a physical location, edges between nodes represent the roads connecting the physical locations, and each edge has a capacity that represents the number of items that can be sent via that road. Nodes are of three types: (i) supply nodes which are locations generating the items with some capacity, (ii) demand nodes which are locations needing a certain number of these items, and (iii) regular nodes which are neither supply nodes nor demand nodes but act as transit junctions. The optimization problem is to find out the maximum amount of goods that can be sent from supply nodes to demand nodes through the road network.

Figure 34 (left) presents an example of a road network with two supply nodes and two demand nodes. Now, let us consider the following supply and demand values:

- Node 1 Supply Rate: 20
- Node 2 Supply Rate: 80
- Node 5 Demand Rate: 30
- Node 6 Demand Rate: 95

For converting this into an applicable maxflow network, we generate one extra node as a source (S) and connect it to all the supply nodes with edge capacities equal to the supply rate of the respective supply node. Similarly, we generate one extra node as a sink (T) and connect it to all the demand nodes with edge capacities equal to the demand rate of the respective demand node. Once we have constructed the flow network as stated above, we run the push-relabel algorithm on it. The flow network constructed for the example is shown in Fig. 34 (right).

In our implementation, we use synthetic graphs to test the application. We generate road networks with 100,000 nodes and approximately 200,000 edges. We

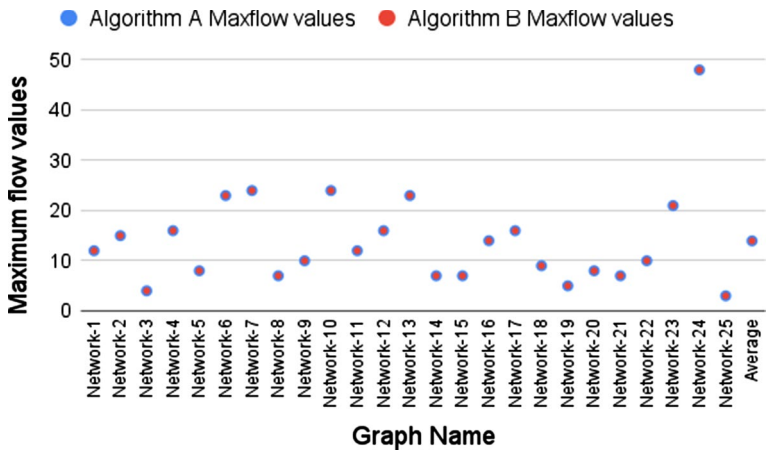


Fig. 35 Accuracy comparison of Algorithms A (baseline) and B (optimized). Note that their maxflow values are almost the same and overlap

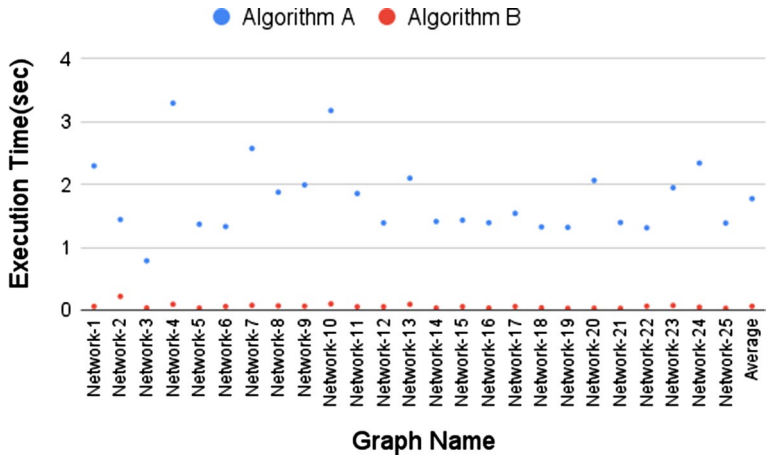


Fig. 36 Performance comparison of Algorithms A (baseline) and B (optimized)

generate a random number from the range [1, 100] to assign edge capacities of the original road network. For each run of the algorithm, a random and disjoint sets of supply and demand nodes are generated, each with a randomly generated supply and demand capacity.

As in the first application, we use two criteria for evaluating our techniques: speedup and inaccuracy. We name the baseline maximum flow algorithm as Algorithm A. We apply all our techniques discussed in Sect. 3 to the baseline, naming the modified algorithm as Algorithm B. We consider the maxflow value generated by Algorithm A to be the actual maximum flow and observe the loss in accuracy of the maxflow values computed by Algorithm B. Figure 35 shows the relative error

of maximum flow computation by Algorithm B. From Fig. 35, we observe that the maximum flow value generated by Algorithm B is primarily the same as that of Algorithm A. On an average, we observe an accuracy loss of 0% in the maximum flow values computed by Algorithm B.

Figure 36 compares performance of the two algorithms. We observe that the average execution time for Algorithm A is 1.773 s, while that for Algorithm B is 0.06 s. On an average, we observe a geomean speedup of 32.41× due to our proposed techniques, clearly indicating their benefit.

Acknowledgements We thank the anonymous reviewers for their helpful comments which considerably improved the overall work.

References

1. Bader, D., Sachdeva, V.: A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. ISCA PDCS, p. 8, 01 (2005)
2. Anderson, R.J., Setubal, J.C.: On the parallel implementation of Goldberg's maximum flow algorithm. In: Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA'92, pp. 168–177, New York, NY, USA (1992). Association for Computing Machinery
3. Dinitz, Yefim: Algorithm for solution of a problem of maximum flow in networks with power estimation. Sov. Math. Dokl. **11**, 1277–1280 (1970)
4. Karzanov, Alexander: Determining the maximal flow in a network by the method of preflows. Doklady Math. **15**, 434–437 (1974)
5. He, Z., Hong, B.: Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–10 (2010)
6. Edmonds, Jack, Karp, Richard M.: Theoretical improvements in algorithmic efficiency for network flow problems. J. ACM **19**(2), 248–264 (1972)
7. Goldberg, Andrew V.: Recent developments in maximum flow algorithms (invited lecture). In: Proceedings of the 6th Scandinavian Workshop on Algorithm Theory, SWAT'98, pp. 1–10. Springer, Berlin, Heidelberg (1998)
8. Goldberg, Andrew V., Tarjan, Robert E.: A new approach to the maximum-flow problem. J. ACM **35**(4), 921–940 (1988)
9. Hussein, M., Varshney, A., Davis, L.: On implementing graph cuts on cuda. First Workshop on General Purpose Processing on Graphics Processing Units, pp. 10, 10 (2009)
10. Karp, R.M., Vazirani, U.V., Vazirani, V.V.: An optimal algorithm for on-line bipartite matching. In: Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing, STOC'90, pp. 352–358, New York, NY, USA (1990). Association for Computing Machinery
11. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: a high-performance graph processing library on the gpu. SIGPLAN Not. **51**(8), 2016 (2016)
12. Luo, L., Wong, M., Hwu, W.: An effective gpu implementation of breadth-first search. In: Proceedings of the 47th Design Automation Conference, DAC'10, New York, NY, USA, pp. 52–55 (2010). Association for Computing Machinery
13. Bader, D.A., Madduri, K.: Gtgraph : A synthetic graph generator suite. http://www.cse.psu.edu/~madduri/software/GTgraph (2006)
14. Johnson, David S., McGeoch, Catherine C.: Network Flows and Matching: First DIMACS Implementation Challenge. American Mathematical Society, USA (1993)
15. Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge (2009)
16. Konect. http://konect.cc/networks/
17. Ford, D.R., Fulkerson, D.R.: Flows in Networks. Princeton University Press, Princeton (2010)

18. Vineet, V., Narayanan, P.J.: Cuda cuts: Fast graph cuts on the gpu. In: 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, pp. 1–8 (2008)
19. Hong, B.: A lock-free multi-threaded algorithm for the maximum flow problem. In: 2008 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–8 (2008)
20. Mittal, Sparsh: A survey of techniques for approximate computing. ACM Comput. Surv. **48**(4), 2016 (2016)
21. Besta, M., Weber, S., Gianinazzi, L., Gerstenberger, R., Ivanov, A., Oltchik, Y., Hoefler, T.: 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, New York, NY, USA (2019). Association for Computing Machinery
22. Singh, S., Nasre, R.: Graffix: Efficient graph processing with a tinge of gpu-specific approximations. In: 49th International Conference on Parallel Processing - ICPP, ICPP'20, New York, NY, USA (2020). Association for Computing Machinery
23. Singh, S., Nasre, R.: Optimizing graph processing on gpus using approximate computing: Poster. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPoPP'19, pp. 395–396, New York, NY, USA (2019). Association for Computing Machinery
24. Princeton cos 226 course. https://www.cs.princeton.edu/courses/archive/spr03/cs226/assignments/baseball.html
25. Kaggle. https://www.kaggle.com