An Efficient Push-Relabel Implementation for Max-Flow Computations on GPUs

Avery VanAusdal
Texas State University
San Marcos, TX, USA
arv107@txstate.edu

Martin Burtscher
Texas State University
San Marcos, TX, USA
burtscher@txstate.edu

***Abstract*—Determining the maximum possible flow (max-flow) between a source vertex and a sink vertex in a network with given capacities on every edge is a fundamental graph problem found in many domains. The Push-Relabel (PR) algorithm is the leading approach for solving the max-flow problem and has been parallelized for GPUs. However, existing approaches suffer from scalability issues and may perform poorly on large graphs. This paper focuses on improving the efficiency of the PR algorithm on GPUs, presenting several implementation and parallelization improvements compared to the state of the art. Our improvements include fast global relabeling on the GPU, worklists to minimize wasted work and CUDA threads, and a two-level parallelization scheme to improve load balancing. Our approach is, on average, 5.75x faster than the fastest prior GPU implementation and 5.08x faster than the fastest CPU code.**

***Index Terms*—Maximum flow, graph algorithms, parallel algorithms, graphics processing units, optimization**

I. Introduction

A flow network is a graph with capacities on the edges that describe how much flow each link can handle, with the vertices acting as junctions. For example, a flow network may represent a network of routers, where each edge is an Ethernet cable with a given maximum throughput. The maximum flow problem (max-flow) finds the highest amount of flow that can be transferred from a source vertex to a sink vertex using all available links. In our example, we could use it to determine the maximum data transfer rate between two routers.

Flow networks can model many other real-world scenarios and systems, such as road networks and electrical circuits. Moreover, the maximum flow problem has found use in applications such as optimization [1], transportation planning [2], computer vision [3], and VLSI design [4], [5].

The Push-Relabel (PR) algorithm [6] is the leading approach for solving the max-flow problem. It is named after its two local vertex operations, which *Push* excess flow to neighboring vertices or *Relabel* themselves to find new valid routes. The algorithm iteratively repeats these operations until converging at an optimal solution [6]. The *global relabeling* heuristic is a well-known optimization to drastically improve the practical performance of PR [7]. The heuristic is run periodically and uses a backwards breadth-first search (BFS) to compute the exact distance from each vertex to the source or sink.

This work has been supported by the National Science Foundation under Award #1955367 and by an equipment donation from NVIDIA Corporation.

Like for many other graph problems, poor scalability and growing input sizes have motivated the exploration of GPU-based solutions for max-flow. Implementations of PR for GPUs [8] have been proposed that successfully improve the performance. PR's local operations make it more suitable for parallelization than other max-flow algorithms. However, most prior works found the BFS-based global relabeling to be faster when performed serially on the CPU and opt for a hybrid approach. In addition, graphs with millions of vertices or more still prove to be very time consuming.

To address these challenges, our work focuses on improving the implementation and parallelization of PR as well as the global relabeling heuristic. For example, to improve scalability, our code runs the global relabeling on the GPU. We reduce the amount of wasted work by using worklists for both the PR and the global relabeling kernels. Our implementation limits the impact of load imbalance caused by varying vertex degrees using a two-level parallelization scheme in the PR and global relabeling kernels. Vertices are processed in parallel by warps, i.e., groups of 32 threads, and each vertex's set of neighbors is processed in parallel by the threads of a warp. We further propose a new frequency for performing the global relabeling that adapts to the characteristics of the input graph without *a priori* knowledge.

This paper makes the following main contributions.

- It describes ECL-MaxFlow, a high-speed PR implementation with GPU-based global relabeling written in CUDA.
- It presents several key optimizations that improve the scalability and load balancing of our code.
- It introduces a new frequency for global relabeling that yields good performance across multiple classes of inputs.
- It shows that ECL-MaxFlow outperforms state-of-the-art CPU and GPU implementations on many real-world and synthetic inputs.

The ECL-MaxFlow CUDA code is publicly available in open source at https://github.com/burtscher/ECL-MaxFlow.

The rest of the paper is organized as follows. Section II provides background information about the max-flow problem and PR algorithm. Section III summarizes related work. Section IV explains our approach in detail. Section V describes the experimental methodology. Section VI presents and analyzes the results. Section VII concludes the paper.

II. Background

A flow network is a directed graph $G(V, E)$ with capacity values c associated with each edge, representing the maximum amount of flow f that the edge can carry. The maximum flow of the network is the maximum amount of flow that can be transferred from a source vertex s to a sink vertex t without exceeding any edge capacities. In a valid flow state, the amount of flow entering a vertex v must be the same as the amount leaving v, unless v is the source or the sink.

Max-flow algorithms generally operate on the *residual graph* $G_f(V, E_f)$, which includes all vertices from the given graph G but replaces the edges E with *residual edges* E_f. The residual edges represent the directions in which excess flow can be moved between vertices based on the current flow state. When an edge $(v, n) \in E$ has less flow than capacity, it is included in E_f, and its residual capacity $c_f(v, n)$ is how much remaining capacity it has. Additionally, when an edge $(v, n) \in E$ has any flow, its inverse (n, v) is in E_f with a residual capacity $c_f(n, v)$ equal to that flow. This allows flow to be removed from an edge if the destination ends up temporarily receiving more flow than it can discharge.

Goldberg and Tarjan's Push-Relabel (PR) algorithm [6] is a well-known approach for solving the max-flow problem. It has a time complexity of $O(V^2E)$. The algorithm pushes as much flow from the source as it can and gradually pushes that flow to the sink. To do so, it introduces the concept of *excess flow*. The excess flow e of a vertex v is the difference between the incoming and outgoing flows for that vertex. Vertices $v \in V - \{s, t\}$ with an $e(v) > 0$ are sometimes called *overflowing*. The final state of the computation will have no overflowing vertices remaining. The algorithm is commonly split into two phases [9]. Phase I pushes as much excess as it can towards and into the sink, which finds the maximum flow value (and, therefore, the minimum cut value). Phase II then returns any excess flow remaining in the network to the source. To help guide the flow in the right direction, the algorithm associates a *height* h with each vertex. In Phase I, the height of the sink t is initialized to 0, and the rest of the height values are commonly initialized via a backwards BFS on G_f, using t as the BFS source. If v can reach t, its height will be a lower bound on the distance from v to t. If v cannot reach t, its height will be set to $|V|$. Vertices with $e(v) > 0$ and $h < |V|$ (i.e., vertices that are overflowing and can reach the source of the BFS) are *active* vertices.

Phase I starts by fully saturating all outgoing edges from the source s. *Saturating* an edge $(v, n) \in E$ means to add enough flow to fill its capacity, i.e., make $f(v, n) = c(v, n)$. The vertices that receive this flow comprise the initial set of active vertices. The main portion of the computation is the repeated application of push or lift operations, whichever is applicable, to active vertices. A vertex v can *push* its excess flow to a neighbor in G_f if the neighbor has a lower height than v, as shown in Algorithm 1. If no lower-height neighbors exist, a vertex instead *relabels* its own height to be 1 higher than its minimum-height neighbor, as shown in Algorithm 2, effectively allowing it to look for an alternate path leading to t in Phase I or to s in Phase II. We refer to this operation as a *lift* to differentiate it from "global" relabeling, which is described later in this section. When there are no remaining active vertices, Phase I is complete.

Algorithm 1 Push operation for vertex v

Require: $e(v) > 0$, $(v, n') \in E_f$, $h(v) > h(n')$
  $n' \leftarrow \text{argmin}_n\{h(n) \mid c_f(v, n) > 0 \text{ and } h(v) > h(n)\}$
  $\Delta \leftarrow \min\{e(v), c_f(v, n')\}$
  if $(v, n') \in E$ **then**
    $f(v, n') \leftarrow f(v, n') + \Delta$
  else if $(n', v) \in E$ **then**
    $f(n', v) \leftarrow f(n', v) - \Delta$
  end if
  $e(v) \leftarrow e(v) - \Delta$
  $e(n') \leftarrow e(n') + \Delta$

Phase II is similar to Phase I with a few changes. It starts with a BFS from the source with $h(s) = 0$, and any unreachable vertex is set to $h(v) = \infty$. The *push* operation is replaced with a similar *reduce* operation, which exclusively removes flow from incoming edges, as shown in Algorithm 3.

As the computation progresses, the structure of G_f changes as vertices lift their own heights, decreasing the accuracy and usefulness of the heights' distance estimates. The global relabeling heuristic, as suggested by Goldberg and Tarjan [6], addresses this issue by periodically recomputing the height labels using another backward BFS in G_f from t in Phase I and from s in Phase II. A common strategy is to perform this global relabeling after every $|V|$ lift operations [7] to balance the overhead cost of running the BFS with the amount of work saved. This heuristic has proven to be highly effective at boosting the performance of PR and is still used in modern implementations [10]. The push and lift operations can be run concurrently on all active vertices without locks through the use of atomic read-modify-write operations [11]. This has enabled the development of GPU-based PR algorithms that leverage the massively parallel architecture of GPUs for performance gains [8]. The BFS used for global relabeling can also be partially parallelized and run on the GPU. However, most prior works found their attempts at doing so to be

Algorithm 2 Lift operation for vertex v

Require: $e(v) > 0$, and $h(v) \leq h(n)$ for all n s.t. $(v, n) \in E_f$
  $h(v) \leftarrow \min\{h(n) \mid c_f(v, n) > 0\} + 1$

Algorithm 3 Reduce operation for vertex v

Require: $e(v) > 0$, $(n', v) \in E_f$, $h(v) > h(n')$
  $n' \leftarrow \text{argmin}_n\{h(n) \mid f(n, v) > 0 \text{ and } h(v) > h(n)\}$
  $\Delta \leftarrow \min\{e(v), f(n', v)\}$
  $f(n', v) \leftarrow f(n', v) - \Delta$
  $e(v) \leftarrow e(v) - \Delta$
  $e(n') \leftarrow e(n') + \Delta$

less performant than offloading the work to a serial CPU implementation, even when accounting for the overhead of transferring data between the CPU and GPU [8].

III. Related Work

The first max-flow algorithm, by Ford and Fulkerson [12], works by repeatedly finding augmenting paths. An augmenting path is any $s \rightarrow t$ path in the residual graph along which additional flow can be sent. Once no such path exists, the maximum flow has been reached. Edmonds and Karp [13] found that augmenting along the shortest paths leads to a polynomial time complexity of $O(VE^2)$. Dinitz [14] proposed a method to find all of the shortest augmenting paths at once, leading to a time complexity of $O(V^2E)$.

Karzanov [15] introduced the concept of a *preflow*, allowing excess flow to be pushed across the graph in $O(V^3)$ time. Based on this, Goldberg and Tarjan [6] proposed the generic Push-Relabel (PR) algorithm with a time complexity of $O(V^2E)$. They also introduced the global relabeling heuristic for improving the algorithm's practical performance. Cheriyan and Maheshwari [16] proved that operating on the highest-labeled active vertices first reduces the worst-case time bound to $O(V^2\sqrt{E})$. Derigs and Meier [9] found the PR algorithm's practical performance to be significantly better than the prior augmenting-path-based methods. Their fastest implementations include global relabeling and a new *gap-search* heuristic, which uses gaps in the height values to quickly identify when vertices can no longer reach the sink.

Cherkassky and Goldberg [7] studied several optimizations and found that operating on the highest-labeled active vertices first is very effective when combined with the global relabel and gap-search heuristics. Even though it is already over two decades old, their optimized serial implementation, first called h_prf and later hi_pr [17], is still one of the fastest max-flow implementations ever published and has been used as a comparison baseline in numerous parallel CPU and GPU max-flow papers [8], [18]–[22].

Anderson and Setubal [23] proposed an efficient parallel version of PR for shared-memory systems as well as a technique for performing the global relabeling heuristic concurrently with the main algorithm using locks. Over a decade later, Bader and Sachdeva [18] improved upon Anderson and Setubal's approach with a cache-aware implementation. Hong [11] introduced a lock-free parallel version of PR that uses atomic read-modify-write operations. Hong proved that pushing to the lowest-height neighbor makes the algorithm robust to any interleaving of Push and Relabel operations without requiring any locking.

A. Push-Relabel on the GPU

Hussein et al. [24] and Vineet and Narayanan [25] introduced the first GPU implementations of PR, which are specifically optimized for obtaining graph cuts for computer-vision problems. Later, He and Hong [8] proposed the first *generic* PR algorithm for CUDA GPUs. Their approach is asynchronous, looping in the PR kernel a fixed number of times before exiting to perform global relabeling on the CPU. Their algorithm also adaptively switches between using the CPU and GPU depending on the available level of parallelism to maximize efficiency. On the CPU, they use Goldberg's fast serial hi_pr [17] code. He and Hong report a speedup of up to 2x compared to hi_pr. Stefanes and Alvino [22] extend He and Hong's hybrid approach by parallelizing the global relabeling and gap relabeling heuristics using OpenMP on the CPU. In addition, they optimize the PR kernel using kernel overlapping and loop unrolling. Stefanes and Alvino report a speedup of up to 3.28x compared to hi_pr.

The Gunrock [26] library of graph analytics codes includes a GPU implementation of Hong's lock-free parallel PR algorithm [11]. However, the authors found their implementation to be slower than a sequential max-flow implementation. The code has since been deprecated. Nevertheless, Khatri et al. [27] used the Gunrock code as a baseline to study various optimizations for GPU PR algorithms, including some approximation techniques. The most impactful optimization they found is changing the global relabeling frequency from every $|V|$ iterations to every 100 iterations.

Hsieh et al. [10] study a set of optimizations applied to He and Hong's parallel GPU algorithm [8]. They call their workload-balanced push-relabel algorithm WBPR. They parallelize the search for the minimum-height neighbor using an entire warp (a group of 32 threads in CUDA) and a parallel reduction. Moreover, they propose two alternatives to using a compressed sparse row (CSR) representation when searching the incoming and outgoing edges for the minimum-height neighbor in G_f. The first alternative, called reverse CSR (WBPR-RCSR), stores incoming edges in compressed-sparse-column (CSC) format with an extra set of pointers to the corresponding forward edge. The second alternative, called bidirectional CSR (WBPR-BCSR), combines the CSR and CSC formats into one bidirectional representation to increase locality when traversing all neighbors. This comes at the cost of needing to search the adjacency list to find the matching reverse edge.

B. Differences in our Approach

Like the other GPU PR codes mentioned in Section III-A, ECL-MaxFlow is based on Hong's lock-free parallel algorithm [11]. However, unlike Hong's follow-up GPU work [8], [28], our PR kernel is not asynchronous. While their kernels freely loop for a preset number of attempted PR cycles, our kernel applies a single push or lift operation per active vertex before exiting, enabling our code to check the termination condition more often. This can cut down on unnecessary iterations. Gunrock [26], Stefanes and Alvino [22], Khatri et al. [27], and WBPR [10] also use synchronous kernels.

All prior GPU-based PR implementations that we are aware of check all vertices in every iteration. ECL-MaxFlow minimizes the amount of work and the number of launched threads by storing only the active vertices in a worklist. Vertices that receive excess flow or could not get rid of all of their excess flow are added to the next iteration's worklist in our PR

kernel. WBPR [10] is the only other code that also uses an active-vertex worklist, but its kernel scans all vertices in every iteration to populate the worklist.

Hong [8], [28], Stefanes and Alvino [22], and WBPR [10] offload the BFS-based global relabeling operation to the CPU, limiting scalability and incurring transfer overhead. Gunrock [26] and Khatri et al. [27] are the exceptions, keeping the global relabel step on the GPU. However, Gunrock's implementation runs the BFS on a single GPU thread, greatly limiting performance. Khatri et al. employ a parallel BFS algorithm that utilizes hierarchical queues and shared memory. In each iteration, their method marks any unvisited neighbors of vertices in the queue, counts how many children each queued vertex has, performs a prefix sum on those values, and uses the sums to fill the next queue in parallel. ECL-MaxFlow also performs the BFS on the GPU. However, we insert elements into the next worklist during the BFS kernel by atomically incrementing the worklist size. The value returned by the atomic addition corresponds to the index at which the element is to be stored in the worklist.

The literature encompasses a variety of different strategies for determining the frequency at which the global relabeling heuristic is performed. When Goldberg and Tarjan [6] first proposed the heuristic, they suggested two strategies. One is to perform the global relabeling after every $|V|$ lift operations. The other is to do so every time an edge into the sink is saturated or an edge out of the source has its flow reduced to zero. Most papers since then describe variations of the first strategy. Cherkassky and Goldberg [7] report (from internal experimentation) that adjusting the number of lift operations needed could favor one class of inputs over another. They considered the suggested threshold of exactly $|V|$ lifts to be a good compromise and used it for their experiments. Later, Goldberg augmented this strategy for hi_pr [17] to account for the density of the graph. The code also considers $|E|$ and activates the global relabeling based on a weighted combination of the total number of lifts and the number of edges traversed to perform those lifts.

Instead of tracking the global number of lifts, all of the generic GPU PR works mentioned attempt a fixed number of push or lift operations on each vertex before exiting to perform a global relabel. We refer to this number as *cycles* since the various works use different definitions of what an "iteration" is. He and Hong [8], for their experiments on a set of well-defined synthetic graphs, chose to use 32 *cycles* for the very-high-degree acyclic-dense graphs and 4096 *cycles* for their other inputs. Gunrock [26] and WBPR [10] both wait for $|V|$ *cycles* between global relabels. Stefanes and Alvino [22] use 98 *cycles* and Khatri et al. [27] use 100 *cycles*. We found running global relabeling every $|V|^2/(1000 \times |E|)$ *cycles* to be performant for our implementation and well balanced for different classes of input graphs.

Like WBPR [10], our PR kernel uses an entire warp to find the minimum-height neighbor of each active vertex. This helps minimize the impact of varying vertex degrees on the workload balance. Our code and WBPR use a minimum reduction per warp to find the collective result for a designated thread to use for pushing or lifting. However, while WBPR performs the reduction in the GPU's shared memory, our code uses CUDA's warp-shuffle primitives that do not access memory. Moreover, and unlike any prior work, we wrote our BFS kernel so the threads of a warp process the neighbors of a single vertex in parallel, which increases memory-access coalescing.

For each "forward" edge $(v, n) \in E$, E_f conditionally includes a "reverse" edge (n, v) to allow flow to be removed from (v, n) when needed. The residual capacities c_f of all edges in E_f are derived from the flow and capacity values of the forward edges in E since $c_f(v, n) = c(v, n) - f(v, n)$ and $c_f(n, v) = f(v, n)$. hi_pr [17] and the mentioned GPU works with public implementations, Stefanes and Alvino [22], Gunrock [26], Khatri et al. [27], and WBPR [10], store and operate on the c_f values of forward and reverse edges. When pushing flow along an edge $e \in E_f$, they subtract residual capacity from $c_f(e)$ and add it to $c_f(inverse(e))$. Since they need to find the inverses of edges often, hi_pr, Stefanes and Alvino, Gunrock, and Khatri et al. use $2 \times |E|$ pointers that bidirectionally map each edge to its matching inverse edge. WBPR-RCSR stores c_f for reverse edges alongside the c_f for their forward edges, so they only need $|E|$ pointers to map from reverse edges to their forward counterparts. In their BCSR format, they need no extra pointers, but they must run binary search to find reverse edges.

Our graph representation is the same as the RCSR format Hsieh et al. propose for WBPR, storing the forward and reverse edges in separate compressed adjacency lists. However, our code directly uses the flow and capacity values of the forward edges in E. We dynamically derive $c_f(e)$ at runtime based on whether e is a forward or a reverse edge. For a forward edge (v, n), if $f(v, n) < c(v, n)$, then $(v, n) \in E_f$. For a reverse edge (n, v), if $f(inverse(n, v)) > 0$, then $(n, v) \in E_f$. This allows us to save an atomic operation for every push by only updating the forward edge's flow value instead of updating two residual capacities. Since c is constant throughout the computation, it can be accessed non-atomically.

IV. Approach

As mentioned in prior work [8] and supported by our own observations, the amount of parallelism available on sparse graphs is limited by a lack of concurrent tasks, i.e., active vertices and frontier vertices. Despite typically only a small portion of the vertices needing computation at a time, the prior GPU works repeatedly process all vertices (see Section III-B), leading to many idle CUDA threads. To minimize the amount of wasted work, our GPU kernels use worklists to only process the vertices that likely need computing.

We further employ a 2-level parallelization scheme specific to GPUs to minimize the workload imbalance caused by different vertices having varying numbers of neighbors. In CUDA, warps are groups of 32 adjacent threads that execute in lockstep. We process the vertices in the relevant worklist in parallel, using an entire warp per vertex, and also process each vertex's set of neighbors in parallel across the threads of

Algorithm 4 Parallel push-relabel algorithm for GPUs

1: $GR\_FREQ \leftarrow \max(100, |V|^2/(1000 \times |E|))$
2: Initialize e, h, f, c, $time$
3: Initial saturating pushes from s, add neighbors to WL_{PR}
4: call **global_relabeling_loop()** ▷ Phase 1
5: $iteration \leftarrow 0$
6: **while** $WL_{PR} \neq \emptyset$ **do**
7: $iteration \leftarrow iteration + 1$
8: $WL_{next} \leftarrow \emptyset$
9: call **push_relabel_kernel()**
10: $WL_{PR} \leftarrow WL_{next}$
11: **if** $iteration \bmod GR\_FREQ == 0$ **then**
12: call **global_relabeling_loop()**
13: **end if**
14: **end while**
15: call **global_relabeling_loop()** ▷ Phase 2
16: $iteration \leftarrow 0$
17: **while** $WL_{PR} \neq \emptyset$ **do**
18: $iteration \leftarrow iteration + 1$
19: $WL_{next} \leftarrow \emptyset$
20: call **push_relabel_kernel()**
21: $WL_{PR} \leftarrow WL_{next}$
22: **if** $iteration \bmod GR\_FREQ == 0$ **then**
23: call **global_relabeling_loop()**
24: **end if**
25: **end while**

Algorithm 5 Code of **push_relabel_kernel** for warp W

1: $v \leftarrow WL_{PR}[W]$
2: **if** $e(v) > 0$ and $h(v) < |V|$ **then** ▷ Active vertex
3: $n'' \leftarrow \infty$
4: **for all** $(v, n) \in E_f$ **do** ▷ Split among threads in W
5: **if** $h(n) < h(n'')$ **then**
6: $n'' \leftarrow n$
7: **end if**
8: **end for**
9: $n' \leftarrow$ Parallel_Reduction(n'') ▷ Min. height neighbor
10: **if** $localIdx == 0$ **then** ▷ Designated thread
11: **if** $h(v) > h(n')$ **then**
12: Push(v, n') ▷ Phase II uses Reduce
13: $WL_{next} \leftarrow WL_{next} \cup \{n'\}$
14: **if** $e(v) > 0$ **then**
15: $WL_{next} \leftarrow WL_{next} \cup \{v\}$
16: **end if**
17: **else**
18: $h(v) \leftarrow h(n') + 1$ ▷ Lift(v)
19: **if** $h(v) < |V|$ **then**
20: $WL_{next} \leftarrow WL_{next} \cup \{v\}$
21: **end if**
22: **end if**
23: **end if**
24: **end if**

Algorithm 6 Implementation of **global_relabeling_loop**

1: **if** Phase I **then** ▷ BFS from sink
2: $h(t) = 0$
3: $h(v) = |V|$ for $v \in V - \{t\}$
4: $WL_{GR} \leftarrow \{t\}$
5: **else if** Phase II **then** ▷ BFS from source
6: $h(s) = 0$
7: $h(v) = \infty$ for $v \in V - \{s\}$
8: $WL_{GR} \leftarrow \{s\}$
9: **end if**
10: $BFS\_ID \leftarrow (-1 \times PR\_iterations)$ ▷ Unique per BFS
11: $level \leftarrow 0$
12: **while** $WL_{GR} \neq \emptyset$ **do**
13: $level \leftarrow level + 1$
14: $WL_{next} \leftarrow \emptyset$
15: call **global_relabeling_kernel**($level$, BFS_ID)
16: $WL_{GR} \leftarrow WL_{next}$
17: **end while**

a warp. WBPR [10] uses a similar approach in their PR kernel, whereas we use it in both our PR and global relabeling kernels. Since the worklists typically do not contain many vertices, even on large graphs, launching only 1 thread per worklist item will likely under-utilize modern GPUs. Launching 32 threads per element (i.e., an entire warp) helps ECL-MaxFlow exploit the GPU hardware more.

Algorithm 4 presents our implementation of the PR algorithm. The **push_relabel_kernel**, **global_relabeling_loop**, and **global_relabeling_kernel** functions are defined in Algorithms 5 to 7. After initializing the necessary data structures on the GPU, a trivial kernel performs the initial saturating pushes from the source vertex s (Line 3 of Algorithm 4), where each edge leaving the source $(s, n) \in E$ receives as much flow as it can handle $f(s, n) \leftarrow c(s, n)$. The kernel also adds each neighbor that receives flow to the active-vertex worklist WL_{PR}. Then, the first global relabeling is called (Line 4) to set the initial heights before the main Phase I loop begins. Since we store active vertices on WL_{PR}, we use $WL_{PR} = \emptyset$ as the termination condition for both phases. After every GR_FREQ calls of **push_relabel_kernel**, the global relabeling is repeated (Lines 11-13). GR_FREQ is set (Line 1) to the greater value of 100 and $|V|^2/(1000 \times |E|)$. This frequency is a combination of the graph's vertex count and average degree with an additional scaling factor. We found this frequency to be effective for our implementation and well balanced across the different types of graphs we tested.

When Phase I completes, a trivial kernel adds all remaining overflowing vertices to WL_{PR}. If none are found, we skip Phase II and the algorithm terminates. Otherwise, Phase II starts with an initial global relabeling (Line 15) to replace Phase I's heights with estimated distances to s (Line 5 of Algorithm 6). While PR in Phase I can add or remove flow, Phase II only needs PR to remove flow from edges until all remaining excess has returned to s [9]. This means the corresponding GPU kernels in Phase II only use reverse edges to find neighbors in the residual graph (Line 4 of Algorithm 5

Algorithm 7 Implementation of **global_relabeling_kernel** for warp W

1: $v \leftarrow WL_{GR}[W]$
2: **for all** $(n, v) \in E_f$ **do** ▷ Split among threads in W
3: $last\_visit \leftarrow$ atomicMin($time(n)$, BFS_ID)
4: **if** $last\_visit \neq BFS\_ID$ **then** ▷ First visit this BFS
5: $h(n) \leftarrow level$
6: $WL_{next} \leftarrow WL_{next} \cup \{n\}$
7: **end if**
8: **end for**

and Line 2 of Algorithm 7).

The GPU kernels described in Algorithms 5 and 7 both use worklists to store vertices that likely need processing in the next iteration. Algorithm 5 adds vertices to WL_{next} if they are still active, i.e., v could not push all of its excess flow away (Lines 15 and 20), or became active, i.e., neighbor n received excess flow from a push (Line 13). Algorithm 7 adds neighboring vertices to WL_{next} when they are visited for the first time in this global relabel operation (Line 6). We launch both kernels with enough threads to assign an entire warp to each vertex in the relevant worklist. Algorithm 5 performs a parallel reduction within each warp (Line 9) to find the minimum-height neighbor n' in G_f for vertex v. Our reduction uses CUDA's warp-level shuffle functions to quickly exchange data between the warp's threads. WBPR [10] uses shared memory to perform its reduction.

We implement the worklists WL_{PR}, WL_{GR}, and WL_{next} as arrays. Each kernel call processes one worklist while filling WL_{next}. Our kernels insert elements into WL_{next} by atomically incrementing WL_{next}'s size. The element is placed into WL_{next} at the index returned by the atomic addition. After each kernel finishes, we swap the pointers of the processed worklist and WL_{next}. WL_{next} is emptied before the next kernel call by setting its size to 0. Algorithms 5 and 7 share the same WL_{next} pointer, so we only need to allocate three worklist arrays of $|V|$ integers in total.

We avoid duplicates in the worklist using integer timestamps associated with each vertex and initialized to 0. When Algorithm 5 adds a vertex v to WL_{next}, it first ensures v has not already been added during this iteration. It does this using the atomicMax operation, which simultaneously increases v's timestamp to the current iteration number $iter$ and returns the replaced timestamp value old_iter. If a thread finds that $old\_iter \neq iter$, then this thread is the first to attempt to add v to WL_{next} in this iteration, so v can safely be inserted.

Algorithm 7 uses the same timestamp array as Algorithm 5, with some changes to avoid interfering with each other. While Algorithm 5 uses atomicMax with the positive value $iter$, Algorithm 7 uses atomicMin with the negative value BFS_ID. By using the same BFS_ID for every kernel call launched by the same global relabeling operation, the timestamp also acts as a "visited" flag (Line 4) without needing to allocate additional storage. This allows our algorithm to use the timestamp array without ever resetting its values.

Our implementation stores the original set of edges E in the widely-used CSR format. However, the residual graph G_f also requires the use of a "reverse" edge (n, v) for each "forward" edge $(v, n) \in E$ to allow flow to be removed from (v, n). We store these reverse edges separately, also in CSR format, with pointers that map each reverse edge's index to its forward counterpart. This allows us to easily process forward and reverse edges separately. As mentioned in Section III-B, and unlike prior GPU works, our implementation dynamically calculates residual capacity, which determines if an edge is present in E_f. A forward edge e is in E_f if it has remaining capacity $f(e) < c(e)$, and a reverse edge re is in E_f if its inverse has flow $f(inverse(re)) > 0$. In Phase I, our GPU kernels iterate through both CSRs to find all neighbors in G_f (Line 4 of Algorithm 5 and Line 2 of Algorithm 7). Since Phase II only uses reverse edges, our GPU kernels are able to entirely skip one of the CSRs in Phase II, reducing the number of neighbors processed and improving performance.

V. Experimental Methodology

A. Codes

We compare the performance of ECL-MaxFlow with the serial CPU code hi_pr [17], the GPU code from Gunrock [26], the GPU code by Khatri et al. [27], and the hybrid code WBPR-RCSR by Hsieh et al. [10] for computing the maximum flow. We obtained the GPU codes from their authors' public GitHub repositories [29]–[31] and hi_pr from an Internet Archive snapshot of the author's website [17].

In all codes we evaluate, reading the input graphs, building internal representations, and allocating data structures on the CPU and GPU are not included in the reported execution times. However, data transfer between the CPU and GPU during the computation are included in the measured runtimes.

The code from Khatri et al. includes several optimizations that introduce approximation. Since the rest of the compared codes use exact methods, we disable all approximation techniques for our experiments.

B. Hardware and Software

We evaluate the performance of the GPU codes on two systems. Since WBPR uses the CPU for global relabeling, we also list the processor and main memory of each system. System 1 has an RTX 4090 GPU with 16,384 processing elements and 24 GB of global memory. It is based on an AMD Ryzen Threadripper 2950X CPU with 48 GB of main memory. System 2 has an NVIDIA A100 GPU with 6912 processing elements and 40 GB of global memory. It is based on an Intel Xeon Gold 6226R CPU with 64 GB of main memory. We run the serial code hi_pr [17] on System 3, which has a higher serial performance than the other two systems. System 3 uses a Ryzen Threadripper 3970X with 256 GB of main memory.

We compiled the GPU codes with nvcc 12.6 on System 1 and with nvcc 12.0 on System 2. For our code, we used the "-O3 -arch=sm_89" flags on System 1 and the "-O3 -arch=sm_80" flags on System 2. We compiled the prior GPU

codes with the same architecture flag as our code. We compiled hi_pr with gcc 13.3.0 and the provided "-O4" flag.

C. Inputs

We used two sets of inputs to evaluate the performance of the codes. The first set, shown in Table I, is composed of three types of graphs used in the first DIMACS Implementation Challenge [32]. The source and sink are the first and last vertex by index, respectively. These graph types are commonly used for evaluating performance in max-flow works [7], [8]:

1) **Acyclic-Dense graphs**: These are complete directed acyclic dense graphs where every vertex is connected to every other vertex. Each edge has a random integer capacity between 1 and $10,000$.
2) **Genrmf graphs**: These graphs are made of b frames, which are square grids of $a \times a$ vertices. The source vertex is a corner of the first frame, and the sink vertex is the opposite corner of the last frame. Each vertex is bidirectionally connected with its neighbors in the same grid. The vertices of each grid are also connected one-to-one with the vertices in the next frame in a random permutation. The cross-grid edges have randomized integer capacities from c_1 to c_2, while neighbor edges have capacities of $c_2 \times a^2$. We used $c_1 = 100$ and $c_2 = 10,000$.
   - **Genrmf-long**: These graphs have an $a : b$ ratio of $1 : 8$, creating many small frames between s and t.
   - **Genrmf-wide**: These graphs have an $a : b$ ratio of $1 : 1$, creating a balance between the size of the frames and their quantity.
3) **Washington-RLG graphs**: These graphs are rectangular grids with r rows and c columns. Each vertex is connected to three random vertices in the next row. The source is connected to all vertices in the first row, and all vertices in the last row are connected to the sink. The cross-row edges have randomized integer capacities from 1 to c_1, while edges connected to the source or sink have capacities of $3 \times c_1$. We used $c_1 = 10,000$.
   - **Washington-RLG-long**: These graphs have an $r : c$ ratio of $1 : 2$, creating a rectangle with the source and sink along the long axis.
   - **Washington-RLG-wide**: These graphs have an $r : c$ ratio of $1 : 1$, creating a square grid.

The second set of inputs is listed in Table II. We obtained them from the Center for Discrete Mathematics and Theoretical Computer Science at the University of Rome (DIMACS) [33], the Galois framework [34], the Stanford Network Analysis Platform (SNAP) [35], and the SuiteSparse Matrix Collection (SSMC) [36]. We modified these graphs where needed as follows. We eliminated self-loops and duplicate edges between the same two vertices. We added any missing back edges to make the graphs undirected. We removed any vertices that are not part of the largest connected component, since those vertices cannot contribute to the flow of the network. For unweighted graphs, we inserted random weights between 100 and $10,000$ to be used as edge capacities. For graphs with existing weights, we divided their weights by 3 to prevent a 32-bit integer overflow from occurring in certain situations. Table II lists the name and the resulting vertex count, edge count, average degree, and maximum degree of each graph. We selected these mostly real-world graphs because they cover a wide range of types and sizes.

Since the undirected graphs do not have specified source and sink vertices, we use 5 distinct pairs of source and sink vertices from each graph. The first pair is the highest and second-highest degree vertices, the second pair is the third-highest and fourth-highest degree vertices, and so on. For the uniform-degree graph 2d-2e20.sym, the pairs are chosen randomly. Using multiple pairs helps to diversify the required computation. We run our code and hi_pr nine times per input and use the median runtime for comparison. Due to their lower performance, we only ran the other GPU codes once per input.

TABLE I
DIRECTED INPUT GRAPHS FROM DIMACS

Graph Name	Vertices	Edges	d-avg	d-max
ac_n2000	2,000	1,999,000	999.50	1999
ac_n4000	4,000	7,998,000	1999.50	3999
ac_n6000	6,000	17,997,000	2999.50	5999
ac_n8000	8,000	31,996,000	3999.50	7999
ac_n10000	10,000	49,995,000	4999.50	9999
genrmf-long_a32_b256	262,144	1,276,928	4.87	5
genrmf-long_a48_b384	884,736	4,347,648	4.91	5
genrmf-long_a64_b512	2,097,152	10,350,592	4.94	5
genrmf-long_a80_b640	4,096,000	20,268,800	4.95	5
genrmf-long_a96_b768	7,077,888	35,085,312	4.96	5
genrmf-wide_a64_b64	262,144	1,290,240	4.92	5
genrmf-wide_a96_b96	884,736	4,377,600	4.95	5
genrmf-wide_a128_b128	2,097,152	10,403,840	4.96	5
genrmf-wide_a160_b160	4,096,000	20,352,000	4.97	5
genrmf-wide_a192_b192	7,077,888	35,205,120	4.97	5
wash...-long_r512_c1024	524,290	1,572,352	3.00	512
wash...-long_r896_c1792	1,605,634	4,816,000	3.00	896
wash...-long_r1280_c2560	3,276,802	9,829,120	3.00	1280
wash...-long_r1664_c3328	5,537,794	16,611,712	3.00	1664
wash...-long_r2048_c4096	8,388,610	25,163,776	3.00	2048
wash...-wide_r1024_c1024	1,048,578	3,144,704	3.00	1024
wash...-wide_r1536_c1536	2,359,298	7,076,352	3.00	1536
wash...-wide_r2048_c2048	4,194,306	12,580,864	3.00	2048
wash...-wide_r2560_c2560	6,553,602	19,658,240	3.00	2560
wash...-wide_r3072_c3072	9,437,186	28,308,480	3.00	3072

TABLE II
UNDIRECTED INPUT GRAPHS

Graph Name	Vertices	Edges	d-avg	d-max
2d-2e20.sym	1,048,576	4,190,208	4.00	4
amazon0312	400,727	4,699,738	11.73	2747
as-skitter	1,694,616	22,188,418	13.09	35455
cit-Patents	3,764,117	33,023,480	8.77	793
coPapersDBLP	540,486	30,491,458	56.41	3299
delaunay_n24	16,777,216	100,663,202	6.00	26
in-2004	1,353,703	26,252,344	19.39	21869
kron_g500-logn21	1,543,901	182,081,678	117.94	213904
rgg_n_2_22_s0	4,194,299	60,718,394	14.48	36
rmat22.sym	3,744,385	65,618,254	17.52	3687
soc-LiveJournal1	4,843,953	85,691,368	17.69	20333
uk-2002	18,459,128	523,113,442	28.34	194955
USA-road-d.E	3,598,623	8,708,058	2.42	9
USA-road-d.NY	264,346	730,100	2.76	8
USA-road-d.W	6,262,104	15,119,284	2.41	9

VI. Results

Table III lists all runtimes gathered for all tested codes for all inputs on System 1. Fig. 1 and Fig. 2 present the speedup of our code over the comparison codes on System 1. The y axes list the input graphs and the x axes the speedups on a logarithmic scale. A speedup above 1 means our code runs faster than the comparison code, and below 1 means our code runs slower. This boundary of a speedup of 1 is visualized by the vertical dotted line. We use "TIMEOUT" labels to identify the inputs that the comparison codes could not complete in 60 minutes. The bottommost sets of bars reflect the geometric-mean speedup over the set of inputs that did not time out for each comparison code. The speedups on System 2 exhibit similar trends. For space reasons, we only describe those results in the text but do not show corresponding figures.

Fig. 1 presents the speedups for the DIMACS graphs listed in Table I. Each of these graphs has exactly one pair of source and sink vertices. Fig. 2 presents the speedups for the undirected graphs listed in Table II. The box-and-whisker plot shows the speedup distribution across the 5 pairs of source and sink vertices used for each input graph. The highest speedup for each graph is indicated by the rightmost whisker and the lowest by the leftmost whisker. The boxes range from the first to the third quartile. The line inside the box indicates the median speedup. We only use the "TIMEOUT" label in Fig. 2 if all 5 pairs timed out.

Note that the comparison code from Gunrock [30] outputs incorrect max-flow values on all tested Acyclic-Dense graphs and Washington-RLG graphs due to an unknown issue. Curiously, this issue does not seem to affect the Genrmf graphs. Since Gunrock's max-flow code is deprecated and not fully correct, its performance should be viewed with caution. We also found WBPR-RCSR [29] to contain an unknown bug that causes it to occasionally output wrong max-flow values, leading us to believe it might be a data race. However, it may not represent the final product as WBPR is from a preprint paper [10]. Since the current version of the code is not fully correct, its performance should also be viewed with caution. The remaining codes, including ours, produce identical maximum flow values for all tested inputs.

A. GPU Performance Comparison

In the mean and on most tested inputs, ECL-MaxFlow is substantially faster than the prior GPU max-flow works on both sets of inputs. Fig. 1 shows that our code is faster than the three baseline GPU codes on all but one DIMACS input. The Khatri code is faster on ac_n10000, which takes less than a second for our code to compute on both systems. Based on the geometric mean, our code is 54.6x faster than Gunrock, 5.97x faster than Khatri et al., and 25.6x faster than WBPR-RCSR across the DIMACS inputs on System 1. On System 2, our code is 63.8x faster than Gunrock, 7.53x faster than Khatri et al., and 29x faster than WBPR-RCSR. Note the many timeouts by the prior GPU works on the Genrmf graphs; WBPR-RCSR times out on all of them, Khatri et al.'s code times out on all except the two smallest long graphs, and Gunrock times out

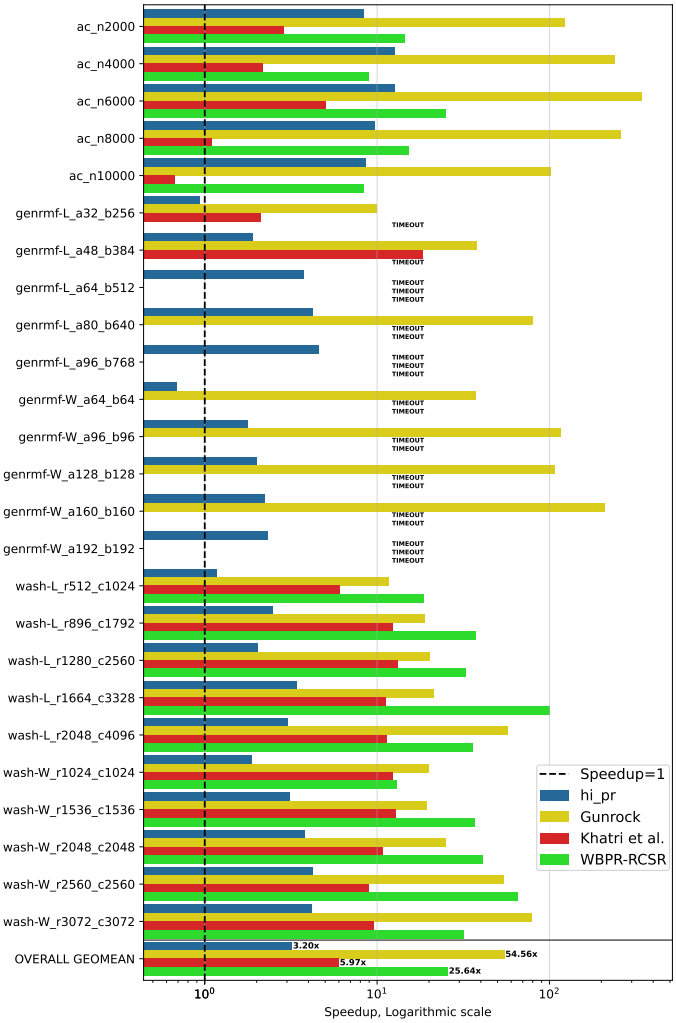

Fig. 1. Speedup over prior works on DIMACS graphs on System 1

on three of the larger graphs. In contrast, our code finishes the slowest Genrmf input in under 39 seconds on our test systems.

Fig. 2 shows that ECL-MaxFlow is faster than the three baseline GPU codes on all tested undirected inputs. According to the geometric mean, it is 615x faster than Gunrock, 5.48x faster than Khatri et al., and 430x faster than WBPR-RCSR on System 1. On System 2, it is 638x faster than Gunrock, 6x faster than Khatri et al., and 482x faster than WBPR-RCSR. The three prior GPU codes time out on all five source/sink pairs for uk-2002, the largest tested graph, while our code finishes its slowest pair in under 3 minutes on our systems.

Across both sets of inputs, our code is 5.75x faster than the fastest prior GPU code on System 1 and 6.78x faster on System 2. Note that these speedups are underestimates as they exclude the inputs on which the comparison codes timed out.

B. CPU Performance Comparison

Fig. 1 shows that our GPU code outperforms the serial hi_pr code on all DIMACS inputs except the smallest Genrmf long and wide graphs. Importantly, the speedup over hi_pr increases with the size of the Genrmf and Washington-RLG-long graphs. Compared to hi_pr, which we always run on System 3, our

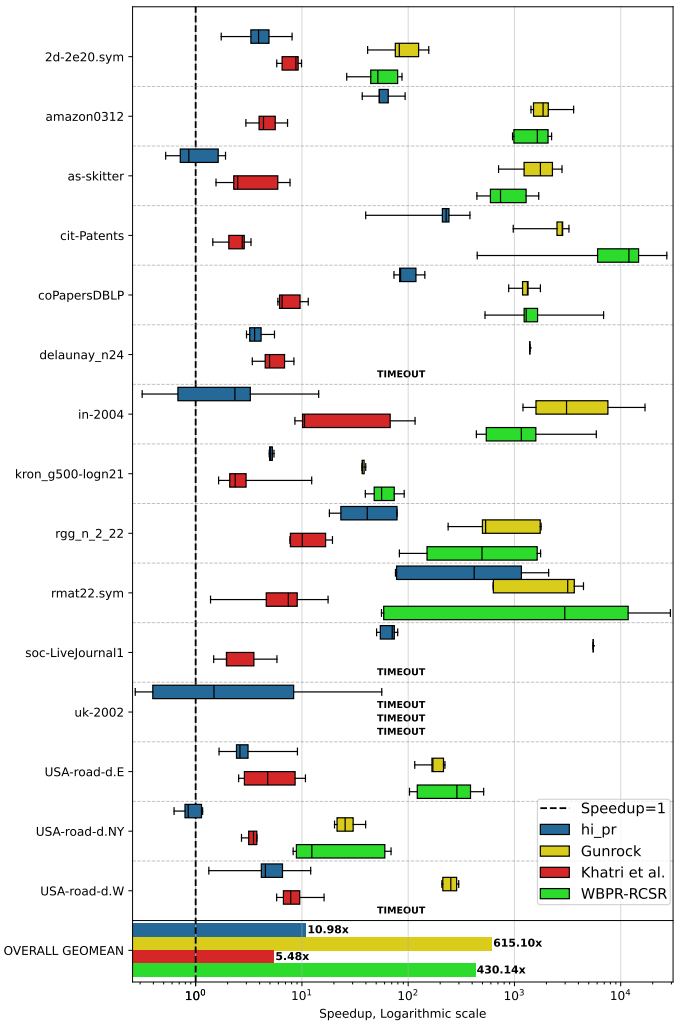

Fig. 2. Speedup over prior works on undirected graphs on System 1

code is 3.2x faster on System 1 and 2.35x faster on System 2 across the DIMACS inputs according to the geometric mean.

Fig. 2 shows that our code is also faster than hi_pr on most undirected inputs. It is only slower on a few source/sink pairs for as-skitter, in-2004, uk-2002, and USA-road-d.NY. However, the median speedups for in-2004 and uk-2002 are above 1. Across the undirected inputs, our code is 11x faster than hi_pr on System 1 and 6.98x faster on System 2. Across both sets of inputs, it is 5.08x faster on System 1 and 3.54x faster on System 2. Of course, the speedup over hi_pr depends on the GPU/CPU pairing and is different on other systems.

VII. Summary and Conclusions

Determining the maximum possible flow (max-flow) between a source and a sink vertex in a network is a fundamental graph problem with many applications, such as transportation planning, bipartite matching, and segmentation. Existing implementations of the max-flow push-relabel (PR) algorithm for GPUs are work-inefficient and, in fact, struggle to compete with the fastest serial CPU code called hi_pr. This paper presents ECL-MaxFlow, a new GPU implementation of PR and the key optimizations used to make it efficient. For example, we use worklists to minimize the amount of wasted work. We employ a 2-level parallelization scheme to reduce workload imbalance and to better exploit the GPU hardware. We also introduce a new frequency for the global relabeling heuristic that yields good performance across multiple classes of inputs.

TABLE III
System 1 computation times in seconds

Graph Name	Source / Sink	hi_pr	ECL-MaxFlow	Gunrock	Khatri	WBPR-RCSR
ac_n2000	0 / 1999	0.200	0.024	2.943	0.068	0.345
ac_n4000	0 / 3999	0.995	0.078	18.677	0.170	0.696
ac_n6000	0 / 5999	1.563	0.124	42.135	0.618	3.098
ac_n8000	0 / 7999	3.526	0.364	93.622	0.401	5.567
ac_n10000	0 / 9999	5.730	0.665	67.138	0.446	5.583
genrmf-L_a32_b256	0 / 262143	0.769	0.822	8.148	1.739	TIMEOUT
genrmf-L_a48_b384	0 / 884735	3.516	1.868	70.350	34.062	TIMEOUT
genrmf-L_a64_b512	0 / 2097151	10.934	2.936	TIMEOUT	TIMEOUT	TIMEOUT
genrmf-L_a80_b640	0 / 4095999	30.830	7.326	581.519	TIMEOUT	TIMEOUT
genrmf-L_a96_b768	0 / 7077887	75.401	16.537	TIMEOUT	TIMEOUT	TIMEOUT
genrmf-W_a128_b128	0 / 2097151	17.073	8.564	912.045	TIMEOUT	TIMEOUT
genrmf-W_a160_b160	0 / 4095999	32.897	14.898	3113.394	TIMEOUT	TIMEOUT
genrmf-W_a192_b192	0 / 7077887	88.406	38.139	TIMEOUT	TIMEOUT	TIMEOUT
genrmf-W_a64_b64	0 / 262143	0.874	1.275	47.342	TIMEOUT	TIMEOUT
genrmf-W_a96_b96	0 / 884735	5.851	3.283	378.303	TIMEOUT	TIMEOUT
wash...-L_r1280_c2560	0 / 3276801	5.605	2.753	55.248	36.388	90.315
wash...-L_r1664_c3328	0 / 5537793	20.959	6.151	131.345	69.046	619.811
wash...-L_r2048_c4096	0 / 8388609	36.080	11.987	683.758	135.893	430.449
wash...-L_r512_c1024	0 / 524289	0.469	0.399	4.649	2.427	7.448
wash...-L_r896_c1792	0 / 1605633	2.749	1.113	21.000	13.644	41.321
wash...-W_r1024_c1024	0 / 1048577	1.459	0.779	15.398	9.587	10.089
wash...-W_r1536_c1536	0 / 2359297	4.887	1.570	30.521	20.118	57.449
wash...-W_r2048_c2048	0 / 4194305	14.656	3.871	96.985	41.529	158.705
wash...-W_r2560_c2560	0 / 6553601	31.278	7.368	400.272	65.326	482.585
wash...-W_r3072_c3072	0 / 9437185	57.276	13.724	1087.039	130.656	434.690
2d-2e20.sym	0 / 400000	0.303	0.038	5.852	0.244	3.275
2d-2e20.sym	100000 / 500000	0.522	0.158	13.021	1.454	8.174
2d-2e20.sym	200000 / 600000	0.302	0.077	5.845	0.448	3.432
2d-2e20.sym	300000 / 700000	0.511	0.104	13.076	1.034	8.300
2d-2e20.sym	400000 / 800000	0.544	0.313	12.980	2.743	8.255
amazon0312	32 / 335	2.312	0.043	80.536	0.188	42.783
amazon0312	8 / 2887	1.934	0.052	74.898	0.156	50.215
amazon0312	12588 / 2989	2.038	0.022	78.559	0.160	48.728
amazon0312	44038 / 9363	1.447	0.027	56.244	0.152	44.354
amazon0312	20107 / 21020	1.436	0.022	33.439	0.087	45.824
as-skitter	7046 / 1039	2.666	1.636	TIMEOUT	4.067	2106.074
as-skitter	7579 / 7588	2.647	5.068	TIMEOUT	11.568	2246.823
as-skitter	7040 / 5569	2.620	3.657	TIMEOUT	5.685	2172.512
as-skitter	811 / 7581	2.397	1.256	3517.286	9.697	2124.444
as-skitter	7582 / 7589	2.584	2.996	2123.278	17.761	2215.509
cit-Patents	2506522 / 3559277	25.210	0.066	166.237	0.096	790.304
cit-Patents	3646084 / 3616615	3.185	0.080	77.928	0.220	35.693
cit-Patents	2248384 / 2524085	8.100	0.039	108.717	0.079	1048.666
cit-Patents	2131075 / 2342725	7.552	0.033	108.042	0.110	200.915
cit-Patents	2099155 / 2466844	15.744	0.065	184.975	0.186	958.565
coPapersDBLP	27942 / 27836	2.336	0.016	28.582	0.097	112.416
coPapersDBLP	27967 / 28839	2.278	0.027	24.174	0.178	45.203
coPapersDBLP	3226 / 27818	2.217	0.019	25.153	0.215	24.161
coPapersDBLP	27077 / 11422	1.339	0.018	24.120	0.112	22.452
coPapersDBLP	78442 / 22607	2.072	0.024	29.105	0.235	12.907
delaunay_n24	3142593 / 4456146	5.257	1.465	TIMEOUT	12.310	TIMEOUT
delaunay_n24	135118 / 5180418	7.778	1.884	TIMEOUT	9.343	TIMEOUT
delaunay_n24	1063984 / 3395389	4.934	1.530	TIMEOUT	10.497	TIMEOUT
delaunay_n24	4378370 / 8062206	6.216	1.127	TIMEOUT	5.083	TIMEOUT
delaunay_n24	8122623 / 8589070	4.849	1.611	2247.888	5.497	TIMEOUT
in-2004	854554 / 246034	0.618	1.969	2369.207	19.974	1068.963
in-2004	854566 / 1198077	1.842	2.707	TIMEOUT	23.266	1186.366
in-2004	1094952 / 1198078	2.723	0.189	3207.309	21.987	1116.153
in-2004	1295144 / 1231855	2.370	0.725	3232.707	7.672	1152.504
in-2004	687758 / 863247	0.598	0.255	440.313	17.288	296.485
kron_g500-logn21	1421105 / 1531673	36.448	6.917	TIMEOUT	85.475	TIMEOUT
kron_g500-logn21	317615 / 787489	12.450	2.527	93.831	7.537	99.853
kron_g500-logn21	350500 / 447517	15.009	2.750	TIMEOUT	4.522	TIMEOUT
kron_g500-logn21	773179 / 1058878	12.617	2.540	93.310	5.315	142.776
kron_g500-logn21	777712 / 711343	12.243	2.426	96.670	5.716	222.432
rgg_n_2_22_s0	1010837 / 1777383	2.781	0.120	59.800	2.320	59.242
rgg_n_2_22_s0	1777466 / 1777484	2.672	0.034	59.490	0.265	55.705
rgg_n_2_22_s0	601227 / 800176	2.250	0.055	29.252	0.551	8.215
rgg_n_2_22_s0	1777076 / 1777681	2.624	0.033	59.523	0.258	58.860
rgg_n_2_22_s0	2346659 / 3295893	2.271	0.125	29.754	2.091	10.349
rmat22.sym	1837250 / 2244496	121.093	0.104	464.781	0.144	TIMEOUT
rmat22.sym	492520 / 1539774	179.145	0.085	270.743	1.505	503.125
rmat22.sym	2374507 / 3094149	5.370	0.069	43.467	0.514	3.869
rmat22.sym	1181640 / 3055470	5.376	0.071	44.845	0.640	4.232
rmat22.sym	3414666 / 1395919	38.130	0.091	333.400	0.421	2674.928
soc-LiveJournal1	10009 / 37344	59.104	1.172	TIMEOUT	2.288	TIMEOUT
soc-LiveJournal1	10029 / 8737	53.072	0.717	TIMEOUT	1.397	TIMEOUT
soc-LiveJournal1	87 / 18962	36.425	0.666	TIMEOUT	0.985	TIMEOUT
soc-LiveJournal1	2914 / 18961	39.362	0.556	3065.798	1.965	TIMEOUT
soc-LiveJournal1	39283 / 91282	31.126	0.391	TIMEOUT	2.273	TIMEOUT
uk-2002	17159799 / 13118355	15.020	38.024	TIMEOUT	TIMEOUT	TIMEOUT
uk-2002	15237349 / 8504954	11.571	0.205	TIMEOUT	TIMEOUT	TIMEOUT
uk-2002	6748291 / 3870967	91.756	61.724	TIMEOUT	TIMEOUT	TIMEOUT
uk-2002	6750968 / 15421709	15.790	58.459	TIMEOUT	TIMEOUT	TIMEOUT
uk-2002	17364050 / 11678704	66.209	7.921	TIMEOUT	TIMEOUT	TIMEOUT
USA-road-d.E	3300125 / 574622	0.639	0.385	44.351	1.831	39.567
USA-road-d.E	1344070 / 1842610	0.416	0.172	28.708	1.852	66.214
USA-road-d.E	3228454 / 219319	3.056	0.337	74.659	2.906	41.011
USA-road-d.E	574620 / 673831	0.700	0.268	45.809	0.771	77.073
USA-road-d.E	793551 / 794244	0.433	0.139	29.894	0.353	71.369
USA-road-d.NY	140960 / 134677	0.151	0.191	4.863	0.718	1.691
USA-road-d.NY	136299 / 139787	0.122	0.143	5.697	0.501	1.775
USA-road-d.NY	141455 / 145196	0.028	0.025	0.496	0.092	1.475
USA-road-d.NY	187960 / 190405	0.027	0.023	0.495	0.062	1.594
USA-road-d.NY	194677 / 47619	0.112	0.179	5.423	0.564	1.478
USA-road-d.W	5901424 / 1026346	4.261	0.940	280.980	15.188	TIMEOUT
USA-road-d.W	1160080 / 5261496	2.253	1.693	358.506	13.271	TIMEOUT
USA-road-d.W	6237650 / 721826	17.191	2.619	543.486	17.534	TIMEOUT
USA-road-d.W	1378713 / 1407631	1.104	0.266	76.089	1.540	TIMEOUT
USA-road-d.W	1712888 / 1993834	12.416	1.027	258.515	9.791	TIMEOUT

We implemented ECL-MaxFlow in CUDA. The source code is available at https://github.com/burtscher/ECL-MaxFlow. We evaluated our code on two distinct sets of input graphs to demonstrate the robustness and generality of our approach. The first set consists of synthetic directed graphs from DIMACS that were specifically designed for evaluating max-flow algorithms and are widely used in the literature. The second set includes real-world and synthetic undirected graphs from various domains and reflects a broader range of input characteristics. On an RTX 4090-based system, ECL-MaxFlow outperforms leading CPU and GPU codes from the literature on these inputs by 5.08x and 5.75x, respectively.

REFERENCES

[1] R. Rockafellar, *Network Flows and Monotropic Optimization*, ser. Series on optimization, computation, and control. Athena Scientific, 1999. [Online]. Available: https://books.google.com/books?id=7_85EAAAQBAJ

[2] E. L. Lawler, *Combinatorial optimization: networks and matroids*. Holt, Rinehart and Winston, New York, 1976.

[3] V. Vineet and P. J. Narayanan, "Cuda cuts: Fast graph cuts on the GPU," in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2008, pp. 1–8.

[4] C.-G. Lyuh and T. Kim, "High-level synthesis for low power based on network flow method," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 3, pp. 364–375, 2003.

[5] J. Qian, Z. Zhou, T. Gu, L. Zhao, and L. Chang, "Optimal reconfiguration of high-performance VLSI subarrays with network flow," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3575–3587, 2016.

[6] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *Journal of the ACM*, vol. 35, no. 4, pp. 921–940, 1988. [Online]. Available: https://dl.acm.org/doi/10.1145/48014.61051

[7] B. V. Cherkassky and A. V. Goldberg, "On implementing the push—relabel method for the maximum flow problem," *Algorithmica*, vol. 19, no. 4, pp. 390–410, 1997. [Online]. Available: http://link.springer.com/10.1007/PL00009180

[8] Z. He and B. Hong, "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)*, 2010, pp. 1–10.

[9] U. Derigs and W. Meier, "Implementing Goldberg's max-flow-algorithm — a computational investigation," *Zeitschrift für Operations Research*, vol. 33, no. 6, pp. 383–403, 1989. [Online]. Available: https://doi.org/10.1007/BF01415937

[10] C.-Y. Hsieh, P.-C. Lin, and S.-Y. Kuo, "Engineering a workload-balanced push-relabel algorithm for massive graphs on GPUs," 2024. [Online]. Available: https://arxiv.org/abs/2404.00270

[11] B. Hong, "A lock-free multi-threaded algorithm for the maximum flow problem," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–8.

[12] L. R. Ford and D. Fulkerson, "Flows in networks," 1963.

[13] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, no. 2, p. 248–264, Apr. 1972. [Online]. Available: https://doi.org/10.1145/321694.321699

[14] Y. Dinitz, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Math. Dokl.*, vol. 11, pp. 1277–1280, 01 1970.

[15] A. Karzanov, "Determining the maximal flow in a network by the method of preflows," *Doklady Mathematics*, vol. 15, p. 434–437, 02 1974.

[16] J. Cheriyan and S. N. Maheshwari, "Analysis of preflow push algorithms for maximum network flow," *SIAM Journal on Computing*, vol. 18, no. 6, pp. 1057–1086, 1989. [Online]. Available: http://epubs.siam.org/doi/10.1137/0218072

[17] A. V. Goldberg, "hi_pr maximum flow solver version 3.6," accessed: 2025-06-12. [Online]. Available: https://web.archive.org/web/20061104200416/http://www.avglab.com/andrew/soft.html

[18] D. Bader and V. Sachdeva, "A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic," in *18th ISCA International Conference on Parallel and Distributed Computing Systems 2005, PDCS 2005*. United States: International Society for Computers and Their Applications (ISCA), 2005, pp. 41–48.

[19] B. Hong and Z. He, "An asynchronous multithreaded algorithm for the maximum network flow problem with nonblocking global relabeling heuristic," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 1025–1033, 2011. [Online]. Available: http://ieeexplore.ieee.org/document/5557863/

[20] S. Soner and C. Ozturan, "Experiences with parallel multi-threaded network maximum flow algorithm," *Partnership for Advanced Computing in Europe*, vol. 2013, no. 1, pp. 1–10, 2013.

[21] N. Baumstark, G. Blelloch, and J. Shun, "Efficient implementation of a synchronous parallel push-relabel algorithm," in *Algorithms - ESA 2015*, N. Bansal and I. Finocchi, Eds., vol. 9294. Springer Berlin Heidelberg, 2015, pp. 106–117. [Online]. Available: http://link.springer.com/10.1007/978-3-662-48350-3_10

[22] M. A. Stefanes and L. F. Alvino, "A hybrid parallel implementation for the maximum flow problem," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, 2018, pp. 229–233. [Online]. Available: https://ieeexplore.ieee.org/document/8374461/

[23] R. J. Anderson and J. C. Setubal, "On the parallel implementation of goldberg's maximum flow algorithm," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1992, pp. 168–177. [Online]. Available: https://dl.acm.org/doi/10.1145/140901.140919

[24] M. E. Hussein, A. Varshney, and L. Davis, "On implementing graph cuts on CUDA," *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:9357297

[25] V. Vineet and P. J. Narayanan, "CUDA cuts: Fast graph cuts on the GPU," in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2008, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/4563095

[26] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: GPU graph analytics," *ACM Trans. Parallel Comput.*, vol. 4, no. 1, Aug. 2017. [Online]. Available: https://doi.org/10.1145/3108140

[27] J. Khatri, A. Samar, B. Behera, and R. Nasre, "Scaling the maximum flow computation on GPUs," *International Journal of Parallel Programming*, vol. 50, no. 5, pp. 515–561, 2022. [Online]. Available: https://link.springer.com/10.1007/s10766-022-00740-7

[28] J. Wu, Z. He, and B. Hong, "Chapter 5 - efficient CUDA algorithms for the maximum network flow problem," in *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series, W.-m. W. Hwu, Ed. Morgan Kaufmann, 2012, pp. 55–66. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123859631000058

[29] "GitHub - NTUDDSNLab/WBPR: Engineering A Workload-balanced Push-Relabel Algorithm for Massive Graphs on GPUs (2024 arxiv) — github.com," https://github.com/NTUDDSNLab/WBPR/, [Accessed 05-07-2025].

[30] "GitHub - gunrock/gunrock at dev — github.com," https://github.com/gunrock/gunrock/tree/dev, [Accessed 05-07-2025].

[31] "GitHub - Jash-Khatri/IJPP — github.com," https://github.com/Jash-Khatri/IJPP/, [Accessed 05-07-2025].

[32] D. Johnson and C. McGeoch, Eds., *Network Flows and Matching: First DIMACS Implementation Challenge*, ser. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1993, vol. 12. [Online]. Available: http://www.ams.org/dimacs/012

[33] C. for Discrete Mathematics and T. C. Science, "DIMACS," http://www.diag.uniroma1.it//challenge9/download.shtml, 2010, accessed: 2022-10-21.

[34] I. T. U. of Texas at Austin, "Galois," https://iss.oden.utexas.edu/?p=projects/galois, 2010, accessed: 2022-10-21.

[35] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[36] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663