Accelerating Push-Relabel Algorithm on GPU via Two-Level Parallelism Paradigm and Efficient CSR Designs

Chou-Ying Hsieh*, Po-Chieh Lin*, Sy-Yen Kuo*†
*Department of Electrical Engineering, National Taiwan University
{f07921043, r12921050, sykuo}@ntu.edu.tw
†Department of Computer Science and Information Engineering, Chang Gung University
kuosy@cgu.edu.tw

***Abstract*—The push-relabel algorithm is an efficient algorithm that solves the maximum flow/ minimum cut problems of its affinity to parallelization. As the size of graphs grows exponentially, researchers have used Graphics Processing Units (GPUs) to accelerate the computation of the push-relabel algorithm further. However, prior works suffer from a significant imbalanced workload distribution while computing with the GPU, which reduces the utilization of the GPU and increases execution time. Therefore, this paper first identifies the two challenges with the cost model we designed. Based on the observation of the model, we propose a novel two-level parallelism neighbor scanning (TLPNS) with two enhanced compressed sparse representations (CSR) to alleviate the workload imbalance. The enhanced CSR significantly reduces the time spent searching for minimum-height neighbors. Additionally, the two-level parallelism approach of our algorithm can effectively reduce the number of idle threads and improve the utilization of the GPU. In the experiment, the enhanced CSRs can significantly decrease the execution time, while the TLPNS can further achieve up to 23x and 6.9x runtime speedup compared to the state-of-the-art work on enhanced CSRs and real-world graphs in maximum flow and bipartite matching tasks, respectively. Our code is open-sourced for further research on accelerating the push-relabel algorithm.**

I. Introduction

The study and application of maximum flow algorithms have long been central to a myriad of computational problems across various disciplines within computer science, including VLSI design [1], [2], optimization [3], and computer vision [4]. The push-relabel (preflow-push) algorithm [5], in particular, represents a cornerstone in solving the maximum flow problem due to its efficiency and versatility. This algorithm iteratively improves the flow within a network by locally pushing excess flow at vertices until the algorithm achieves an optimal flow, leveraging relabel operations to dynamically adjust the heights of nodes to maintain a valid flow.

The emergence of general-purpose computing on graphics processing units (GPUs) has opened a new road to handle large-scale graphs. The GPU provides massively parallel processing capabilities, which has significantly reduced the computation time of the push-relabel algorithm in prior works [6], [7]. However, the prior work fails to fully leverage the computational capabilities of GPUs due to the imbalanced workload assignment.

To address these issues, we propose two enhanced CSR data structures: Reversed CSR (RCSR) and Bidirectional CSR (BCSR) to reduce the vertex search time. Upon these data structures, we devise a two-level parallelism neighbor scanning (TLPNS) approach for increase the utilization of the GPU. The code is open-sourced[1] We summarize our contributions as follows:

- We propose two enhanced data structures for the push-relabel algorithm: RCSR and BCSR. These structures significantly reduce neighbor searching time and offer different advantages for graphs with varying characteristics.
- We introduce a two-level parallelism approach for accelerating neighbor searching using the enhanced CSRs. In maximum flow tasks, this approach achieves a geometric average speedup of 1.41x with RCSR and 1.12x with BCSR. For bipartite matching tasks, we observe a geometric average execution time speedup of 2.56x with RCSR and 2.49x with BCSR.

The remainder of this paper is organized as follows: Section II provides background on the maximum flow problem, GPU architecture, and challenges of traditional approaches. Section III details our two-level parallelism method and the enhanced CSR structures. Section IV presents a performance evaluation, and Section VI concludes the paper and provides the future works.

II. Background

A. Generic Parallel Push-relabel Algorithm

The fundamental concept behind the push-relabel algorithm is to generate maximum flow from the source and progressively direct it to the sink. It introduces the concept of *excess flow*, allowing a vertex to receive multiple incoming flows, potentially accumulating more flow than it can discharge. Such vertices are termed *active*. The excess flow value $e(v)$ at a vertex v indicates the amount of surplus flow present. The push-relabel algorithm involves identifying active vertices and

[1]GitHub link: https://github.com/NTUDDSNLab/WBPR.git

Algorithm 1: The lock-free push-relabel algorithm

Input: $G(V, E)$: the directed graph, $G_f(V, E_f)$: the residual graph, $c_f(v, u)$: the residual flow on (u, v), $e(v)$: the excess flow of the vertex v, $h(v)$: the height of the vertex v, $Excess\_total$: the sum of excess flow

Output: $e(t)$: the maximum flow value

Data: Initialize $c_f(v, u), e, h$, $Excess\_total \leftarrow 0$

```
/* Step 0: Preflow                                          */
1  foreach (s, v) ∈ E do
2      c_f(s, v) ← 0
3      c_f(v, s) ← c(s, v)
4      e(v) ← c(s, v)
5      Excess_total ← Excess_total + c(s, v)
6  while e(s) + e(t) < Excess_total do
       /* Step 1: Push-relabel kernel (GPU)                 */
7      cycle = |V|
8      while cycle > 0 do
9          foreach u ∈ V and e(u) > 0 and h(u) < |V| do
10             h' ← ∞
               /* Operation 1: MHNS                         */
11             foreach (u, v) ∈ E_f do
12                 if h(v) < h' then
13                     h' ← h(v)
14             if h(u) > h' then
                   /* Operation 2: BES                      */
15                 d ← MIN(e(u), c_f(u, v'))
16                 AtomicSub(c_f(u, v'), d)
17                 AtomicSub(e(u), d)
18                 AtomicAdd(c_f(v', u), d)
19                 AtomicAdd(e(v'), d)
20             else
21                 h(u) ← h' + 1
22         cycle ← cycle − 1
       /* Step 2: Heuristic Optimization (CPU)              */
23     GlobalRelabel()
```

applying *push* and *relabel* operations until no active vertices remain.

Initially, the algorithm pushes as much flow as possible from the source to its neighboring vertices, a process known as *preflow*. A push operation compels an active vertex to *discharge* its excess flow to neighboring vertices in the residual graph G_f. To prevent infinite pushing, a *height function* h is employed for each vertex. The source's initial height is $|V|$, while all other vertices start at zero. An active vertex u can only push to a neighboring vertex v if $h(u) = h(v) + 1$. If none of the neighbors meet this condition, the vertex increases its height via a relabel operation, setting $h(u) \leftarrow \min(h') + 1$, where h' is the minimum height of the neighboring vertices. This process is called *minimum-height neighbor searching*. It prevents endless pushing as the height of an unpushable active vertex rises until it eventually deactivates. The algorithm terminates once no active vertices remain.

The push-relabel algorithm is well-suited for parallelization due to the localized nature of its operations. He et al. [6] propose the lock-free GPU approach, which is considered the state-of-the-art GPU implementation (Algorithm 1). In this algorithm, each thread checks the activation of a specific vertex u (line 9). For convenience, the thread handling vertex u is referred to as *thread* u. If the vertex is active, thread u searches for its neighboring vertex v with the minimum height among all neighbors, called Minimum-Height Neighbor Scanning (MHNS) (lines 10-13). The thread then pushes flow from u to v if $h(u) > h(v)$ (lines 15-19), or otherwise relabels the active vertex u. The push operation have to locate backward edge v', u, which is called Backward Edge Searching (BES), to increase the backward flow. After a number of iterations, the algorithm applies a *global relabeling* heuristic [8] (line 6) to improve practical performance by updating each vertex's height via a backward breadth-first search (BFS) from the sink to the source. This ensures that the height of each vertex reflects the shortest distance from the sink in the residual graph G_f. Additionally, the excess flow of inactive vertices is adjusted to guarantee termination of the process (line 6).

B. Execution Model of GPUs

For clarity in our description, we adopt the terminology of NVIDIA's Compute Unified Device Architecture (CUDA). Modern GPUs expand on the traditional Single Instruction, Multiple Data (SIMD) architecture to a Single Instruction, Multiple Thread (SIMT) model, where a warp serves as the fundamental computing unit. A *warp* typically comprises 32 threads that share a program counter (PC) and execute instructions in a *lockstep* manner. All threads in a warp execute and access memory simultaneously, meaning conditional branching within a warp can cause *branch divergence*, serializing the warp's execution and incurring significant overhead. Multiple warps comprise a Cooperative Thread Array (CTA) or a thread block (TB) which is executed by the Streaming Multiprocessor (SM) concurrently. To avoid divergence, we use warp as a basic scheduling unit for creating second parallelism. [9], [10]

C. Motivation and Challenges

The state-of-the-art lock-free approach faces significant GPU underutilization, particularly as graph size increases, due to its inability to fully exploit the parallelism of GPUs. To identify the inefficiencies, we develop a cost model for the lock-free push-relabel algorithm on GPUs using the Bulk-Synchronous Parallel (BSP) model. Starting from the superstep model C_s derived from [11], we have:

$$C_s = \max(w_t) + \max(h_t \times g) + l \tag{1}$$

where w_t is the local computation time for thread t, h_t is the number of messages sent or received by thread t, g is the message cost, and the cost of barrier synchronization is captured by l. The *max* operator reflects the SIMT execution

model of GPUs, where threads wait for the slowest one to finish. In the lock-free push-relabel algorithm, a superstep is represented by lines 8 to 22, with no message passing between threads, allowing us to ignore h_t.

We further break down the local computation w_t into two components: minimum-height neighbor searching (MHNS) and backward-edge searching (BES), resulting in the refined model:

$$C_s = \max(\text{MHNS}_t + \text{BES}_t) + l \tag{2}$$

Since MHNS and BES workloads depend on the number of vertices scanned by each thread, we derive the final equation as:

$$C_s = \max_{t \in T} \left(\sum_{v}^{V_t} (k \cdot d(v) + \lambda_v P(v) + (1 - \lambda_v) R(v)) \right) + l \tag{3}$$

Here, V_t represents the set of active vertices for thread t, $P(v)$ and $R(v)$ are the push and relabel times for vertex v, λ_v indicates the operation type, and $d(v)$ is the out-degree of vertex v. Reducing the cost of MHNS and BES, or balancing vertex workload across threads, can reduce the superstep cost. To achieve this, we propose enhanced compressed sparse row (CSR) representations and a two-level parallelism approach targeting these improvements.

III. Workload-balanced Push-relabel Algorithm

A. Overview

Basically, our new algorithm is the enhanced lock-free push-relabel algorithm with efficient CSR design and thread management, shown in Algorithm 2. The two enhanced Compressed Sparse Row (CSR) formats substantially minimize the time required to scan neighboring vertices or identify backward edges in the residual graph. Upon these enhanced CSRs, we design *two-level parallelism neighbor scanning (TLPNS)* to make workload more balanced among threads, and thus accelerate the neighbor searching process.

B. Enhanced Compressed Sparse Representation

Using a traditional CSR for the residual graph in the push-relabel algorithm is inefficient. As mentioned in Section II-A, an active vertex must find the minimum-height neighbor v in the residual graph, a process known as *outgoing neighbors scanning*. After identifying the minimum-height neighbor, the backward edge of the pair (v, u) needs to be located so that the flow from u to v can be decreased, and the flow from v to u can be increased—this process is referred to as *backward edges finding*. The structural design of the residual graph's data affects the efficiency of these operations.

If backward edges are placed directly below forward edges, as depicted in Figure 1 (b), they can be accessed in constant time. However, finding an active vertex's neighbors then takes $O(|E|)$. For instance, locating the neighbors of vertex 2 requires scanning all orange blocks to find the edge $(2, 0)$.

Algorithm 2: Two-level Parallelism in An Iteration of Push-relabel Kernel

```
   Data: avq: active vertex queue
   /* Scan the active vertices                    */
1  foreach u ∈ V do
2      if e(u) > 0 and h(u) < |V| then
3          pos ← atomic_add(avq, 1) ;
4          avq[pos] ← u ;
5  grid_sync() ;
   /* First level parallelism                     */
6  foreach u ∈ avq do
       /* Second level parallelism                */
7      foreach v ∈ D(u) do
8          min = ParallelReduction() ;
9      tile.sync() ;
10     if localIdx == 0 then
11         if h(v) < h(min) then
12             Push() ;
13         else
14             Relabel() ;
```

To improve the efficiency of finding incoming neighbors of a given vertex, we developed the Reversed CSR (RCSR) and Bidirectional CSR (BCSR) for residual graphs, shown in Figures 1 (b) and (c), respectively. RCSR uses a separate CSR to store backward edge locations, with *flow_idx* indexing backward flows rather than storing their values directly. This allows us to identify all neighbors of vertex 2 by scanning both the original and reversed CSR (represented by the orange portion of Figure 1 (b)). This design reduces the time complexity of outgoing neighbor finding from $O(|E|)$ to $O(d(v))$, where $d(v)$ denotes the degree of the given vertex v. However, we found that accessing RCSR can place considerable pressure on memory bandwidth because neighbors are stored at non-contiguous addresses, resulting in uncoalesced memory access.

To improve data locality, we combined incoming and outgoing neighbors and proposed BCSR, as illustrated in Figure 1 (d). Although the aggregated neighbors cannot be accessed in constant time, sorting the column list by vertex ID in ascending order allows us to reduce the search time for backward edges from $O(d(v))$ to $O(\log_2 d(v))$ using a binary search. We will evaluate the performance of both RCSR and BCSR in Section IV.

C. Two-level Parallelism Neighbor Scanning

Inspired by the vertex-centric programming model [12], we present the two-level parallelism neighbor scanning (TLPNS) to accelerate the process of MHNS. Algorithm 2 show an iteration of Algorithm 1 line:6. We initially use the `atomic_add()` operation to gather active vertices to the Active Vertex Queue (AVQ). Since a global synchronization is placed at line 5 (grid_sync()), we can reorganize thread

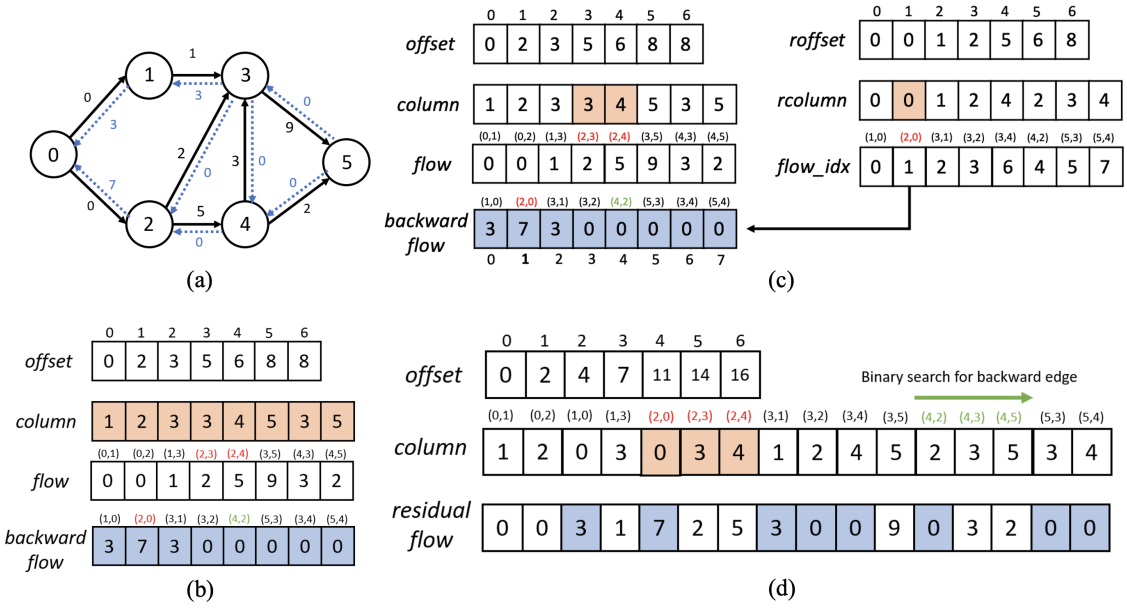

Fig. 1. (a) The example residual graph. (b) The original CSR. (c) The reversed CSR (RCSR). (d) The bidirectional CSR (BCSR). The blue block represents the flow of backward edges, while the red blocks indicate all neighbors of vertex 2 in the residual graph. The orange blocks represent the edges scanned when identifying the minimum-height neighbor of a given vertex 2. The green section indicates the cost of finding the backward flow associated with the edge $(2, 4)$.

assignments for the search of minimum-height neighbors. Additionally, we can exit the *while* loop early (Algorithm 1, line 8) when no active vertices remain in the AVQ, preventing redundant iterations.

To parallelize the search for a minimum-height neighbor, we use a tile (a group of threads) for each active vertex. We employ the parallel reduction technique proposed by [13] to identify the neighbor with the minimum height. The Kernel 7 implementation from [13] is used due to its superior bandwidth speedup among the seven kernels presented.

Since the search for the minimum value has low arithmetic intensity, achieving peak bandwidth gain is crucial for optimizing our implementation. Before the reduction process, each warp fetches a fixed number of neighboring vertices into individual buffers stored in shared memory, reducing global memory access. After determining the minimum-height neighbor, the designated warp thread with *localIdx* set to 0 performs the push or relabel operation.

IV. Evaluation

A. Experiment Setup

Applications and datasets. We evaluated our workload-balanced push-relabel algorithm on both the *maximum flow/minimum cut* and *bipartite matching* problems. For the maximum flow/minimum cut problem, we utilized the Washington and Genrmf synthetic networks from the DIMACS 1st Implementation Challenge [15] alongside 10 real-world networks from SNAP [14]. For the bipartite matching problem, we selected 12 real-world bipartite graphs from KONECT [16].

Since the real-world networks from SNAP do not have specified source and sink vertices, we conducted a breadth-first search to identify 20 pairs of distinct source and sink vertices, each with the top 25% longest diameters. A *super-source* and *super-sink* were then connected to all 20 sources and sinks, respectively, allowing us to transform real-world datasets to residual networks. In bipartite matching problem, the super-source connects to all vertices in L set, and all the vertices in R set are connected to super-sink node. Detailed pairing information is available in our open-source repository.

Implemented algorithms. We implemented the CUDA-based push-relabel algorithm proposed by He et al. [6], referred to as the thread-centric (TC) approach. Since the dynamic switching between GPU and CPU execution in TC is orthogonal to our focus, we implemented only its GPU kernel component. For fairness, we adopt the CUDA-based global relabeling technique proposed by Khatri et al. [7] across all algorithms. Both TC and TLPNS are integrated with three different CSR-based graph representations, as shown in Table II:

Measuring machine. We ran all the experiments on the Intel i9-14900 24-core processor @ 5.4GHz with 128GB DDR5 RAM and an Nvidia RTX 4090 GPU. The number of blocks and block size of the kernel configuration are 1024 and 128, respectively.

B. Performance

We measured the execution time of the push-relabel kernel (Algorithm 1, lines 6–22), excluding global relabeling. Tables I and III report results for maximum flow and bipartite matching tasks, respectively.

TABLE I

THE EXECUTION TIME AND SPEEDUP OF DIFFERENT ALGORITHMS ACROSS 13 GRAPHS. THE R0-R10 GRAPHS ARE THE REAL-WORLD NETWORK FROM SNAP [14], WHILE THE S0-S1 ARE THE SYNTHESIS NETWORK GENERATED FROM 1ST DIMACS CHALLENGE [15]. THE EDGE CAPACITY OF GRAPHS IN SNAP IS SET TO 1. THE **BOLD** FONT TIME REPRESENTS THE BEST EXECUTION TIME AMONG THESE FOUR ALGORITHMS. (**TO**: EXECUTION TIME EXCEEDS ONE HOUR.)

Graph	$\|V\|$	$\|E\|$	Execution time (ms)						TLPNS Speedup	
			CSR	CSR+	RCSR	RCSR+	BCSR	BCSR+	on RCSR	on BCSR
Amazon0302 (R0)	262,111	1,234,877	TO	TO	1,600	2,738	**1,354**	2,681	0.58x	0.51x
roadNet-CA (R1)	1,965,206	2,766,607	TO	TO	**8,779**	14,147	10,375	16,404	0.62x	0.63x
roadNet-PA (R2)	1,088,092	1,541,898	TO	TO	**7,713**	13,668	8,226	14,332	0.56x	0.57x
web-BerkStan (R3)	685,230	7,600,595	TO	TO	18,965	**8,905**	16,776	13,199	2.13x	1.27x
web-Google (R4)	875,713	5,105,039	TO	TO	6,705	**5,215**	7,417	6,327	1.29x	1.17x
cit-Patents (R5)	3,774,768	16,518,948	TO	TO	21,380	1,743	30,737	**1,335**	12.27x	23.02x
cit-HepPh (R6)	34,546	421,578	182,908	TO	149	96	**31**	61	1.55x	0.51x
soc-LiveJounal1 (R7)	4,847,571	68,993,773	TO	TO	138,240	**61,979**	241,805	136,686	2.23x	1.77x
soc-Pokec (R8)	81,306	1,768,149	TO	TO	35,665	**14,669**	23,938	15,182	2.43x	1.58x
com-YouTube (R9)	1,134,890	2,987,624	TO	TO	242,880	89,483	124,534	**64,436**	2.71x	1.95x
com-Orkut (R10)	3,072,441	117,185,083	TO	TO	819,254	806,244	**349,095**	2,155,083	1.23x	0.16x
Washington-RLG (S0)	262,146	785,920	TO	TO	162,792	287,390	132,482	**99,410**	0.57x	1.33x
Genrmf (S1)	2,097,152	10,403,840	TO	TO	**2,138**	2,685	2,900	2,503	0.8x	1.16x

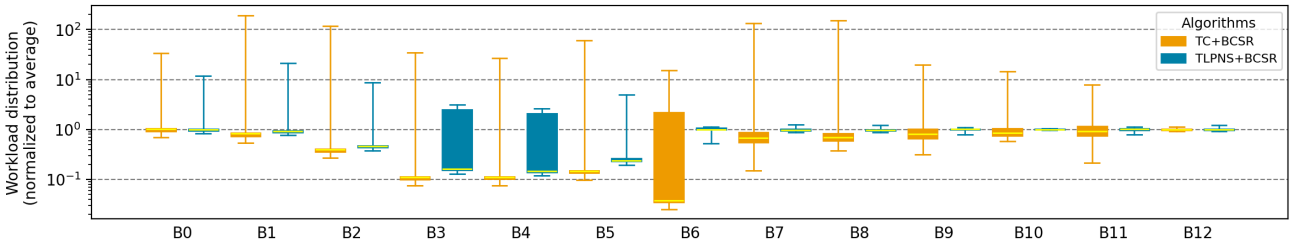

Fig. 2. The workload distribution of the bipartite matching problem across 13 bipartite graphs.

TABLE II

ALGORITHM CONFIGURATIONS ORGANIZED BY LOAD-BALANCING STRATEGIES (**LB**) AND GRAPH REPRESENTATIONS (**GR**).

LB / GR	*CSR*	*RCSR*	*BCSR*
TC	CSR	RCSR	BCSR
TLPNS	CSR+	RCSR+	BCSR+

Overall, the original CSR format suffers from overhead in minimum-height neighbor search, while our TLPNS method improves performance for both RCSR and BCSR. Specifically, TLPNS achieves geometric mean speedups of 1.41× (RCSR) and 1.12× (BCSR) in maximum flow, and 2.56× (RCSR) and 2.43× (BCSR) in bipartite matching.

Two key observations arise. First, TLPNS is most effective on graphs with high degree variance, provided the graph is large enough to amortize synchronization overhead. For small graphs like B0–B2, the benefit is limited. In graphs such as R0–R2, S0, S1, and B12, limited speedup is due to their balanced degree distribution or low maximum degree. For instance, R0 (Amazon0302) has most nodes in a single SCC (Strongest Connect Component) with uniform degree, and R1/R2 (road networks) have low degrees, leading to underutilized tiles. In contrast, TLPNS performs well on high-degree graphs like R5, B7, and B8.

Second, BCSR outperforms RCSR in maximum flow but underperforms in bipartite matching. This aligns with Section III: RCSR enables constant-time backward edge access, benefiting high-degree bipartite graphs, while BCSR improves locality and memory coalescing, aiding maximum flow tasks.

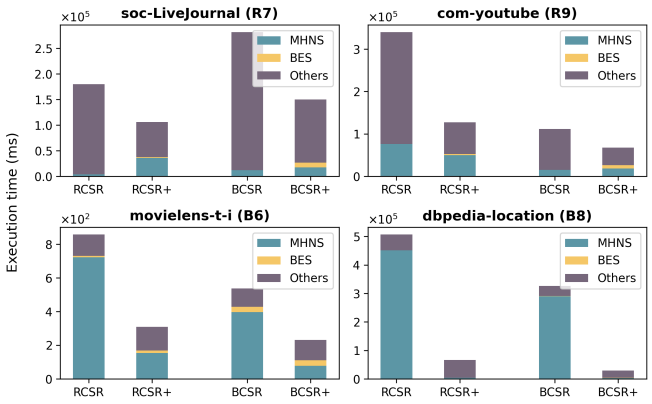

Fig. 3. The execution time breakdown of the four selected datasets among maximum flow and bipartite matching problems, respectively. The *MHNS* is the time of minimum-height neighbor searching; *BES* stands for backward-edge searching time; the *others* comprises the synchronization overhead and thread idle time, exclude from the thread waiting time in MHNS and BES parts.

C. Execution Time Breakdown

Figure 3 provides an execution time breakdown for four representative datasets from Tables I and II. To analyze the

TABLE III
THE EXECUTION TIME OF VARIOUS PUSH-RELABEL ALGORITHMS WAS MEASURED ACROSS 13 GRAPHS FOR THE BIPARTITE MATCHING PROBLEM. THE BEST EXECUTION TIME AMONG THE THREE DESIGNS IS HIGHLIGHTED IN **BOLD**. (**TO**: EXECUTION TIME EXCEEDS ONE HOUR.)

Graph	$\|L\|$	$\|R\|$	$\|E\|$	Maximum Flow	Execution Time (ms)						TLPNS Speedup	
					CSR	CSR+	RCSR	RCSR+	BCSR	BCSR+	on RCSR	on BCSR
corporate-leadership (B0)	24	20	99	20	4.96	5.08	**4.73**	4.74	5.8	4.78	1x	1.21x
Unicode (B1)	614	254	1,255	188	129.32	128.72	16.19	10.43	15.19	**10.30**	1.55x	1.47x
UCforum (B2)	899	522	7,089	516	628.14	387.73	25.44	13.21	22.13	**13.17**	1.93x	1.68x
movielens-u-i (B3)	7,601	4,009	55,484	2,836	43,321	23,056	202.16	**79.03**	156.70	79.10	2.56x	1.98x
Marvel (B4)	12,942	6,486	96,662	5,057	123,274	65,384	232.36	**119.24**	210.93	123.62	1.95x	1.71x
movielens-u-t (B5)	16,528	4,009	43,760	3,258	91,488	70,465	444.41	159.57	371.11	**155.76**	2.79x	2.38x
movielens-t-i (B6)	16,528	7,601	71,154	5,882	150,253	100,593	417.76	**215.18**	393.99	218.96	1.94x	1.8x
YouTube (B7)	94,238	30,087	293,360	25,624	TO	TO	91,477	12,428	71,008	**9,841**	7.36x	7.22x
DBpedia_locations (B8)	172,079	53,407	293,697	50,595	TO	TO	244,893	35,774	191,775	**27,912**	6.85x	6.87x
BookcCrossing (B9)	340,523	105,278	1,149,739	75,444	TO	TO	92,563	19,622	69,080	**19,185**	4.72x	3.6x
stackoverflow (B10)	545,195	96,678	1,301,942	90,537	TO	TO	340,975	83,454	260,519	**82,086**	4.09x	3.17x
IMDB-actor (B11)	896,302	303,617	3,782,463	250,516	TO	TO	151,605	**47,122**	153,295	47,365	3.22x	3.24x
DBLP-author (B12)	5,624,219	1,953,085	12,282,059	1,952,883	TO	TO	**92,703**	113,449	229,375	178,023	0.82x	1.29x

impact of our TLPNS approach, we profiled the algorithm based on Equation 3 (Section II), focusing on performance across both RCSR and BCSR.

The TLPNS method effectively redistributes idle threads to work on Minimum-Height Neighbor Scanning (MHNS), resulting in a significant reduction in time spent on both MHNS and other overhead tasks, particularly in BCSR and RCSR implementations. For the maximum flow task, performance improvements are largely attributed to reduced idle time in non-MHNS portions. In contrast, for the bipartite matching task, the reduction in MHNS time is the primary source of speedup.

Our analysis shows that, for sparse datasets in maximum flow tasks, the primary bottleneck occurs when threads become idle while determining active vertices (line 9 in Algorithm 1). Conversely, for bipartite matching, the bottleneck lies in the time spent on MHNS itself. By leveraging the TLPNS approach, which centrally processes active vertices (Algorithm 2, lines 1-4) and applies multiple threads to MHNS, we effectively alleviate these bottlenecks in both maximum flow and bipartite matching tasks.

D. Workload Analysis

To validate that the performance gains from TLPNS are derived from improved workload distribution, we utilized the workload analysis method from [17] to measure warp execution times. Specifically, the delegated first thread in each warp was tasked with recording execution timestamps, allowing us to visualize the workload distribution across 13 bipartite graphs for both the TC and TLPNS approaches (Figure 2). Both configurations used RCSR for consistency in comparison.

Two primary insights emerged from this analysis: (1) *Reduction in variability*: While the graph does not explicitly show reduced mean execution times for TLPNS, it demonstrates a notable reduction in the standard deviation of execution times across warps. This result indicates that TLPNS achieves more balanced workload distribution, leading to more consistent GPU thread performance. (2) *Impact of synchronization on small graphs*: For smaller graphs, even with balanced workload distribution, overall performance may degrade due to synchronization overhead. This is evident in graphs B0, B1, and B2, where the benefits of workload balancing are offset by the costs of thread synchronization.

V. RELATED WORKS

The Ford-Fulkerson algorithm [18] is the foundational method for solving the maximum flow problem by iteratively finding augmenting paths from source to sink. Edmonds and Karp [19] improved this by using breadth-first search (BFS) to find augmenting paths, reducing the runtime complexity from $O(Ef)$ to $O(VE^2)$, where f is the maximum flow. Dinic [20] further optimized this with level graphs and blocking flows, achieving $O(V^2E)$. The push-relabel algorithm by Goldberg et al. [5] refines flow locally, also with $O(V^2E)$ complexity.

Building on this, gap relabeling [5] and global relabeling [8] are heuristics designed to skip inefficient relabel operations. Anderson et al. [21] introduced the first parallel implementation on shared-memory multiprocessors, while He et al. [6] proposed a CUDA-based version that dynamically switches between CPU and GPU. Khatri et al. [7] later implement a GPU-version global relabel kernel, which is adopted in our work.

VI. CONCLUSION AND FUTURE WORKS

In this paper, we address inefficiencies in the traditional parallel push-relabel algorithm by introducing enhanced compressed sparse representation data structures, RCSR and BCSR. These improvements reduce the runtime complexity of minimum-height neighbor scanning and backward edge searching. Our two-level parallelism neighbor scanning (TLPNS) approach further enhances GPU utilization. TLPNS achieves a geometric average speedup of 1.41x for RCSR and 1.12x for BCSR in maximum flow tasks. For bipartite matching, it delivers geometric average speedups of 2.56x and 2.49x, respectively. Future work can explore dynamic tile size adjustment and further optimization of data structures and strategies.

REFERENCES

[1] 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.

[2] 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.

[3] R. T. Rockafellar, *Network flows and monotropic optimization*. Athena scientific, 1999, vol. 9.

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

[5] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *Journal of the ACM (JACM)*, vol. 35, no. 4, pp. 921–940, 1988.

[6] 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)*. IEEE, 2010, pp. 1–10.

[7] J. Khatri, A. Samar, B. Behera, and R. Nasre, "Scaling the maximum flow computation on gpus," *International Journal of Parallel Programming*, vol. 50, no. 5-6, pp. 515–561, 2022.

[8] U. Derigs and W. Meier, "Implementing goldberg's max-flow-algorithm—a computational investigation," *Zeitschrift für Operations Research*, vol. 33, pp. 383–403, 1989.

[9] J. Fox, A. Tripathy, and O. Green, "Improving scheduling for irregular applications with logarithmic radix binning," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–7.

[10] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 308–317.

[11] D. B. Skillicorn, J. M. Hill, and W. F. McColl, "Questions and answers about bsp," *Scientific Programming*, vol. 6, no. 3, pp. 249–274, 1997.

[12] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: vertex-centric graph processing on gpus," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, 2014, pp. 239–252.

[13] M. Harris *et al.*, "Optimizing parallel reduction in cuda," *Nvidia developer technology*, vol. 2, no. 4, p. 70, 2007.

[14] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, pp. 1–20, 2016.

[15] D. S. Johnson, C. C. McGeoch *et al.*, *Network flows and matching: first DIMACS implementation challenge*. American Mathematical Soc., 1993, vol. 12.

[16] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd international conference on world wide web*, 2013, pp. 1343–1350.

[17] M. Almasri, Y.-H. Chang, I. El Hajj, R. Nagi, J. Xiong, and W.-m. Hwu, "Parallelizing maximal clique enumeration on gpus," in *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2023, pp. 162–175.

[18] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Canadian journal of Mathematics*, vol. 8, pp. 399–404, 1956.

[19] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *Journal of the ACM (JACM)*, vol. 19, no. 2, pp. 248–264, 1972.

[20] Y. Dinitz, "Dinitz'algorithm: The original version and even's version," in *Theoretical Computer Science: Essays in Memory of Shimon Even*. Springer, 2006, pp. 218–240.

[21] R. Anderson and J. C. Setubal, "A parallel implementation of the push-relabel algorithm for the maximum flow problem," *Journal of parallel and distributed computing*, vol. 29, no. 1, pp. 17–26, 1995.