

Engineering A Workload-balanced Push-Relabel Algorithm for Massive Graphs on GPUs

Chou-Ying Hsieh, Po-Chieh Lin, and Sy-Yen Kuo

National Taiwan University, Taipei, Taiwan
 {f07921043, r12921050, sykuo@ntu.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 need to handle the significant memory consumption to represent a massive residual graph. In addition, the nature of their algorithms has inherently imbalanced workload distribution on GPUs. This paper first identifies the two challenges with the memory and computational models. Based on the analysis of these models, we propose a workload-balanced push-relabel algorithm (WBPR) with two enhanced compressed sparse representations (CSR) and a vertex-centric approach. The enhanced CSR significantly reduces memory consumption, while the vertex-centric approach alleviates the workload imbalance and improves the utilization of the GPU. In the experiment, our approach reduces the memory consumption from $O(V^2)$ to $O(V + E)$. Moreover, we can achieve up to 7.31x and 2.29x runtime speedup compared to the state-of-the-art on real-world graphs in maximum flow and bipartite matching tasks, respectively. Our code will be open-sourced for further research on accelerating the push-relabel algorithm.

Keywords: Maximum flow algorithms · Push-relabel algorithm · Graphics processing units · Large-scale networks · Workload imbalance.

1 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 [18, 19], optimization [20], and computer vision [21]. The push-relabel (preflow-push) algorithm [7], 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.

As graphs grow to encompass billions of nodes and edges, traditional CPU-based solutions need help with the computational requirements necessary to

process such immense datasets efficiently. This limitation is particularly pronounced in the applications handling large-scale social networks, web graphs, and biological networks, where the ability to compute efficiently maximum flow or minimum cut is fundamental but essential for analyzing or understanding graph information. The emergence of general-purpose computing on graphics processing units (GPUs) has opened a new road to meet this requirement. The GPU provides massively parallel processing capabilities, which has significantly reduced the computation time of the push-relabel algorithm in prior works [9,14]. However, those works fail to address the impact of massive graphs. First, they use an adjacency matrix to represent the graph, which takes $O(V^2)$ memory space, where V is the set of vertices. The enormous memory consumption puts remarkable pressure on a single GPU. For instance, the most advanced GPU nowadays, H100 NVL with 188 GB VRAM, can only accommodate about 306,594 vertices if we use 2 bytes for a data point in the adjacency matrix. Second, the traditional parallel push-relabel algorithm on the GPU has severe workload imbalance on each thread, which fails to utilize the entire computing power of a GPU.

To address these challenges, we propose a novel workload-balanced push-relabel algorithm (WBPR)¹ designed for modern GPU architectures dealing with massive graphs. In WBPR, the two enhanced compressed sparse representation (CSR) data structures, RCSR (Reversed CSR) and BCSR (Bidirectional CSR), reduce the significant memory space of a large graph. Besides, these CSRs can also provide efficient memory access to neighbor vertices for graphs with different characteristics. Based on these CSRs, we then develop a novel vertex-centric approach of parallel push-relabel algorithms, which collects all the active vertices in a queue first, so that we can accelerate the local operations of an active vertex by assigning an arbitrary number of threads, rather than using a single thread iteratively. We summarize our contributions as follows:

- We derive a computation model to evaluate the execution time of the push-relabel on GPUs. Upon the model, we identify where the weakness and workload imbalance of the state-of-the-art design.
- To accommodate massive graphs in a GPU, we proposed RCSR and BCSR, which significantly reduce space complexity from $O(V^2)$ to $O(V + E)$ with trivial overhead on the process of the push-relabel algorithm. We also state that RCSR and BCSR have different advantages in graphs with varying characteristics.
- The novel vertex-centric approach can alleviate the workload imbalance among threads and improve the utilization of the GPU. It achieves an average 2.49x and 7.31x execution time speedup in maximum flow tasks with RCSR and BCSR, respectively; it also gains an average 2.29x and 1.89x execution time speedup in bipartite matching tasks with RCSR and BCSR.

We organize the rest of the paper as follows: Section 2 introduces the background of the maximum flow problem, the architecture of a GPU, and the weaknesses of the traditional approach. We discuss the BCSR and vertex-centric

¹ source: <https://github.com/NTUDDSNLab/WBPR>

approach in Section 3. Section 4 presents the evaluation of our design compared to the state-of-the-art one. Section 6 concludes.

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  $(s, v) \in E$  do
2    $c_f(s, v) \leftarrow 0$ 
3    $c_f(v, s) \leftarrow c(s, v)$ 
4    $e(v) \leftarrow c(s, v)$ 
5    $Excess\_total \leftarrow Excess\_total + c(s, v)$ 
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  $u \in V$  and  $e(u) > 0$  and  $h(u) < |V|$  do
      10        $h' \leftarrow \infty$ 
      11       foreach  $(u, v) \in E_f$  do
      12         if  $h' < h(v)$  then
      13            $h' \leftarrow h(v)$ 
      14       if  $h(u) > h'$  then
      15          $d \leftarrow \text{MIN}(e(u), c_f(u, v'))$ 
      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          $h(u) \leftarrow h' + 1$ 
      22      $cycle \leftarrow cycle - 1$ 
  /* Step 2: Heuristic Optimization (CPU) */
  23   GlobalRelabel()

```

2 Background

2.1 The Maximum Flow/ Minimum Cut Problems

Given a directed graph $G(V, E)$, where V and E are the vertex and weighted edge set, respectively, each edge (u, v) has a weight, *capacity* $c(u, v)$, representing the

maximum amount of flow that can be passed through this edge. The maximum flow problem tries to find the maximum flow value from the source vertex s to the sink vertex t . Generally, algorithms solve the maximum flow problem operating on the *residual graph* $G_f(V, E_f)$, which has the same vertex set as the given G with *residual edges* E_f . The weight $c_f(u, v)$ of a residual edge (u, v) represents how much-remaining flow can be passed from u to v . Note that when there is a flow from u to v , there will be a residual edge from v to u , even if there is no (v, u) in E .

The foundational algorithm solving the maximum flow problem is the Ford-Fulkerson algorithm [6], which finds augmenting paths from the source to the sink iteratively. It pushes the additional flow until no such path remains. Edmonds and Karp improved the efficiency of finding augmenting paths using breadth-first search (BFS). The Edmonds-Karp algorithm [5] reduces the runtime complexity of Ford-Fulkerson from $O(Ef)$ to $O(VE^2)$, where f is the maximum flow value. Dinic et al. [4] further improved upon the basic augmenting path approach by employing a level graph and blocking flows, which significantly reduced the number of augmentations needed; hence, reducing the complexity to $O(V^2E)$ and $O(V^{3/2}E)$ on general and unit-capacity graph respectively. Goldberg et al. [7] proposed the push-relabel algorithm (also known as the preflow-push algorithm), which focuses on local operations to adjust the flows. Some optimizations are based on it to improve the complexity to $O(V^2E)$.

2.2 Generic Parallel Push-relabel Algorithm

The basic idea of the push-relabel algorithm is to generate as much flow as possible at the source, and gradually push it to the sink. It introduces the *excess flow* concept that allows a vertex to have multiple flows coming into it; namely, it allows a vertex to have more flow coming in than passing out during the push-relabel procedure, this vertex is called *active*. We denote $e(v)$ as the excess flow value on vertex v . The procedure of the push-relabel algorithm is to find active vertices and apply *push* and *relabel* operations on them until there are no remaining active vertices. To find the initial active vertices, the algorithm pushes flow from the source to all its neighbor vertices as much as possible, called *preflow*. A push operation then forces an active vertex to *discharge* its excess flow and pushes to its neighbors in G_f . To avoid the endless pushing, the push-relabel algorithm also introduces the *height function* h on each vertex. At first, the source height is $|V|$, and the height of vertices except the source is 0. It forces an active vertex u to push to the vertex v iff $h(u) = h(v) + 1$. If no neighbor vertex satisfies the constraint, the active vertex will *relabel* its height by finding the minimum height h' of its neighbor vertices and setting $h(u) \leftarrow h' + 1$. With the relabel operation, the height of an active vertex, which cannot push its excess flow at the end, will be increased. The active vertex will be deactivated once its height exceeds $|V|$. The whole procedure ends when there are no active vertices.

The push-relabel algorithm has not only the best theoretical and practical complexity, but it is well-suitable to parallelization due to its inherent structure

and the local nature of its operations. Numerous works have developed the parallel version on the different hardware platforms, such as multiprocessor [2, 10], and graphics processing unit (GPU) [9, 14, 24]. Since there are different trade-offs and optimization used in different algorithms, we target the lock-free algorithm proposed by He et al. [9], which is the state-of-the-art algorithm operating on the GPU shown in 1. It parallelly checks whether a vertex u is active by assigning a thread in a GPU (Line: 9). For convenience, we will use thread u to represent the thread, which checks the vertex u . If the vertex is active, the thread u will find its neighbor vertex v whose height is minimum among other neighbor vertices (line:10 - 13). The thread then push flow from u to v (line:15 - 19) when $h(u) > h(v)$; otherwise, the thread will relabel the active vertex u . After *cycles* times of iteration, the algorithm applies *global-relabeling* heuristic, one of the push-relabel optimizations, to improve the practical performance [9]. The global relabeling updates the height of each vertex by performing a backward breadth-first search (BFS) from the sink to the source in the residual graph G_f . The height will be reassigned to the shortest distance from the sink. After global relabeling, the procedure subtracts *Excess.total* from the excess flow of those inactive vertices to guarantee the termination of the procedure (line: 6 in Algorithm 1). Since the lock-free algorithm assigns a thread to process a vertex, we also call this a *thread-centric* approach.

The most significant difference between the original and lock-free push-relabel algorithms is that the lock-free one relaxes the push constraint of the height function $h(v) = h(u) + 1$ to $h(v) > h(u)$ (line: 14). By finding the minimum-height neighbor and using atomic operations, the correctness of the lock-free algorithm has been proven in [10].

2.3 Execution Model of GPUs

For the convenience of description, we use the CUDA (Compute Unified Device Architecture, proposed by Nvidia Corporation) terminologies. The modern GPU extends SIMD (Single-Instruction-Multiple-Data) to SIMT (Single-Instruction-Multiple-Thread), where a warp is the primary computing unit. A *warp* usually consists of 32 threads, which share a programming counter (PC) and execute in a lockstep manner; namely, all threads in a warp will execute and access the memory simultaneously; hence, the if-else condition inside a warp will cause warp divergence and serialize the execution of a warp, leading to tremendous overhead. On the top of warps, the thread block (TB) executes multiple warps concurrently to hide the latency of memory access on each warp. Note, all threads in a TB share two fast on-chip memory, *shared memory* and *cache*. The former memory is programmable, while the latter is automatic caching. In addition to this cached memory, the *memory coalescing* technique can also improve memory access efficiency. If threads inside warp access the memory continuously, the hardware will combine these accesses into one request to increase the memory bandwidth. Multiple TBs comprise an entire grid, which maps to a physical GPU hardware and equips up with *global memory* for data used in a GPU.

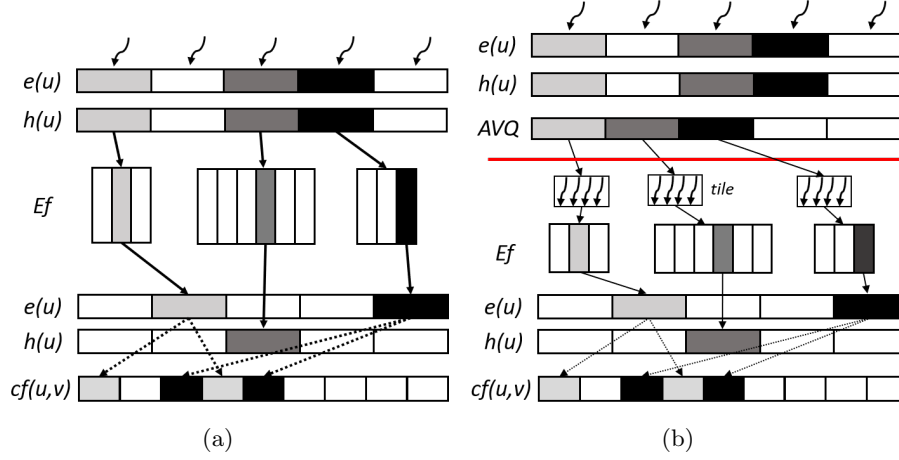


Fig. 1. The illustration of the push-relabel algorithm in both (a) thread-centric manner and (b) vertex-centric manner in an iteration. Both two manner approach check the active vertex first ($e(u), h(u)$), and then find the its minimum-height neighbor for push (updating $e(u)$ and $cf(u, v)$) or relabel (updating $h(u)$). The vertex-centric approach uses the AVQ to collect all active vertices, so that it can assign more threads (a tile) for finding a minimum-height neighbor.

2.4 Workload Imbalance of Push-relabel Algorithm for Large Graph on GPUs

Despite the parallel lock-free algorithm mentioned in Section 2.2 has gained excellent performance improvement on GPU, the thread-centric approach still faces significant workload imbalance, especially when the underlying graph becomes larger and larger. Specifically, the prior approach fails to fully utilize the parallelism of the GPU. Before diving into the imbalance problem, we want to derive the cost model of the thread-centric approach on GPU to help us explain where the workload imbalance happens. The execution time model can be roughly represented by the following equation:

$$time = \max_{\{t \in T\}} \left(\sum_v^{V_t} \underbrace{(k * d(v) + (\lambda_v P(v) + (1 - \lambda_v) R(v)))}_{\text{execution time of a local operation}} \right) \quad (1)$$

In this equation, T represents the set of the workers (threads) evolving in this computation, and we assume the total active vertex set that the thread t should perform the push and relabel operations on is V_t . $P(v)$ and $R(v)$ are the time of performing push and relabel on vertex v , while the λ_v , which determines which operation the vertex needs to perform, is 0 or 1. The $d(v)$ stands for the out-degree of v in the residual graph G_f , while $k * d(v)$ represents the total time the vertex v finds its minimum-height neighbor, where k is the constant. Since the GPU computes in the SIMT manner mentioned in the last section, we use the

\max operation to represent the overall execution time, which is determined by the last finished thread. If we want to achieve the optimal overall time, we need to divide the workload equally on each thread and reduce the execution time of a local operation in the last finished thread; namely, let sum of out-degree $d(v)$ in each V_t equally and reduce the time on searching minimum-height neighbor vertex; However, the prior approach fails to achieve either of the two goals. Figure 2.4 (a) illustrates the execution of the thread-centric push-relabel algorithm and where the workload imbalance happens. Since it assigns a thread to check if a vertex is active, the V_t is not equal. Besides, the active vertex's out-degree is various, so the execution time of the local operation on each active vertex is also various. Each thread has to iteratively search the incoming and outgoing neighbors in E_f without the help of other threads. Furthermore, the large graph forces us to use the compressed format, such as compressed sparse row (CSR), rather than the adjacency matrix to represent the residual graph. Finding all outgoing neighbors in the adjacency matrix only takes $O(d(v))$ time complexity. However, it takes up to $O(V * d(v))$ in CSR format, exacerbating the impact of workload imbalance. We will discuss the details in the next section.

3 Workload-balanced Push-relabel Algorithm

3.1 Overview

Figure 2.4 (b) shows the architecture for our workload-balanced push-relabel implementation. As mentioned in the Section 2, we aim to alleviate the two workload imbalances using the thread-centric approach. Fundamentally, we first assign all threads to scan all vertices to find the active ones and add the active vertices to the active vertex queue (AVQ). With the AVQ, each thread has an equal workload when finding active vertices. Besides, the active vertices can be coalesced in the AVQ; hence, we can assign a tile (a group of threads) to find the minimum-height neighbor vertex of an active vertex, which reduces the time of the searching time (Algorithm 1 line: 11-13) from $O(d(v))$ to $O(\log_2 d(v))$. Since we can process multiple vertices and use multiple threads in an active vertex simultaneously, we call this approach *two-level parallelism*. Furthermore, we designed two enhanced CSRs, *reversed CSR* and *bidirectional CSR*, to reduce significant time in scanning all neighbors or finding backward edges on the residual graph.

3.2 Enhanced Compressed Sparse Representation

The prior approach used an adjacency matrix to represent a residual graph, so it takes $O(d(u))$ time to find all in- and out-neighbor vertices with a given vertex u . Using an adjacency matrix is convenient and less time-consuming in finding the minimum-height neighbor vertex, but it has $O(V^2)$ memory complexity, which puts considerable pressure on the GPU. More and more people have embraced the compressed sparse representation (CSR) to reduce the memory consumption

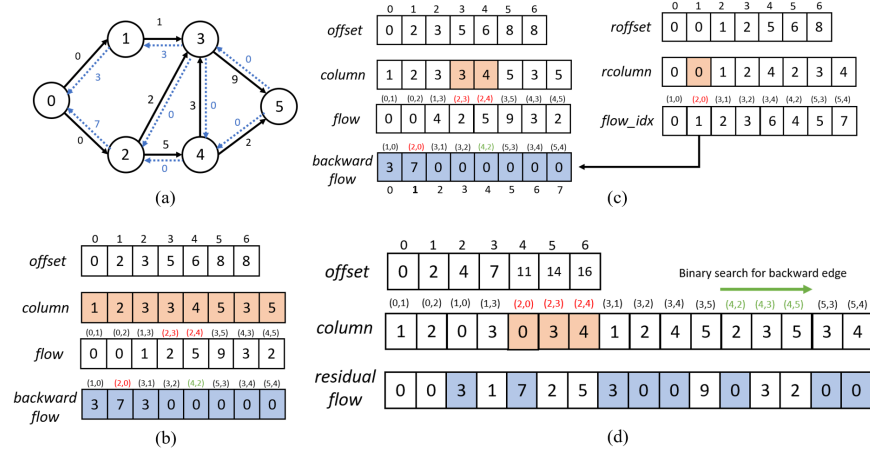


Fig. 2. (a) The example residual graph. (b) The original CSR. (c) The reversed CSR (RCSR). (d) The bidirectional CSR (BCSR). The blue block is the flow of backward edges; while the red color stands for all neighbors of vertex 2 in the residual graph. The orange ones represent the edges to scan when finding the minimum-height neighbor of a given vertex 2. The green part is the cost to find the backward flow with the given edge (2, 4).

of a massive graph, especially the sparse graph. However, using a conventional CSR as the residual graph is inefficient for the push-relabel algorithm. As mentioned in Section 2.2, an active vertex must find the minimum-height neighbor v in the residual graph. After finding the minimum-height neighbor, we need to find the backward edge of the (v, u) , so that we can decrease $flow(u, v)$ and increase $flow(v, u)$. If we directly put the backward edge below the forward edge shown in Figure 2 (b), it can access the backward edge in a constant time, but it takes $O(|E|)$ to find an active vertex's neighbors. For instance, finding neighbors of vertex 2 requires scanning all orange blocks to find (2, 0).

To alleviate the inefficiency of finding incoming neighbor vertices of a given vertex, we designed reversed CSR (RCSR) and bidirectional CSR (BCSR) for the residual graph, shown in Figure 2 (b), (c), respectively. The RCSR uses another CSR for backward edges. The *flow_idx* records the index of backward flow rather than the value. In this way, we can find all vertex 2's neighbors by scanning the original CSR and reversed CSR (the orange part of Figure 2 (b)). Unfortunately, we found that accessing RCSR puts tremendous pressure on the memory bandwidth since neighbors of a vertex are stored in discontinuous addresses, causing uncoalesced memory access.

We aggregated incoming and outgoing neighbors to achieve better locality and proposed BCSR shown in Figure 2 (d). Since the neighbor is aggregated, we cannot access the backward edge in a constant amount of time. For instance, the edge (4, 2) belongs to vertex 4's neighbor; Hence, we need to do an additional search to find the backward edge of (2, 4). If we sort the column list in ascending

order by vertex ID, we can reduce the searching time from $O(d(v))$ to $O(\log_2 d(v))$ with binary search, where $d(v)$ is the degree of a given vertex v . We evaluate the performance of both RCSR and BCSR in Section 4.

3.3 Two-level Parallelism with Vertex-centric Approach

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

```

Data: avq: active vertex queue
/* Scan the active vertices */
1 foreach  $u \in V$  do
2   if  $e(u) > 0$  and  $h(u) < |V|$  then
3      $pos \leftarrow \text{atomic\_add}(avq, 1)$  ;
4      $avq[pos] \leftarrow u$  ;
5  $\text{grid\_sync}()$  ;
/* First level parallelism */
6 foreach  $u \in avq$  do
  /* Second level parallelism */
  7 foreach  $v \in D(u)$  do
  8    $min = \text{ParallelReduction}()$  ;
  9    $\text{tile.sync}()$  ;
  10 if  $localIdx == 0$  then
  11   if  $h(v) < h(min)$  then
  12      $\text{Push}()$  ;
  13   else
  14      $\text{Relabel}()$  ;

```

Algorithm 2 shows the two-level parallelism algorithm for an iteration in Algorithm.1 (line: 9-21). In each iteration, we first use the `atomic_add()` operation to add the active vertex to the AVQ. Since we put a global synchronization in Line:5 (`grid_sync()`), we can re-organize the thread assignment for the search of minimum-height neighbor. Besides, we can early break the *while* loop (Algorithm 1, line: 8) when there is no active vertex remaining in the AVQ, which avoids the redundant iteration. We use a warp as a tile for an active vertex to parallelize the search of a minimum-height neighbor vertex. Since the neighbor of an active vertex stored in the CSR is continuous, using a warp can accelerate the process and reduce the memory request with inherent memory coalescing mentioned in Section 2.3. We use the parallel reduction proposed in [8] to find the vertex with minimum height. We chose the Kernel 7 implementation in [8] since it has the best bandwidth speedup among the seven kernels. Because the finding minimum operation has a very low arithmetic intensity, the peak bandwidth grain can benefit our implementation the most. After finding the minimum-height neighbor, we will use the delegated thread in a warp, whose *localIdx* is 0, to execute the push or relabel operations.

Table 1. The execution time and speedup of different algorithms across 13 graphs. The R0-R10 graphs are the real-world network from SNAP [16], while the S0-S1 are the synthesis network generated from 1st DIMACS Challenge [13]. The edge capacity of graphs in SNAP is set to 1. The bold font time represents the best execution time among these four algorithms.

Graph	V	E	Execution time (ms)				Speedup (TC/VC)	
			TC+RCSR	TC+BCSR	VC+RCSR	VC+BCSR	RCSR	BCSR
Amazon0302 (R0)	262,111	1,234,877	5,728	2,307	8,477	5,191	0.67x	0.44x
roadNet-CA (R1)	1,965,206	2,766,607	57,966	70,468	74,707	32,842	0.78x	2.15x
roadNet-PA (R2)	1,088,092	1,541,898	27,667	14,822	43,283	18,078	0.64x	0.82x
web-BerkStan (R3)	685,230	7,600,595	82,984	23,959	35,129	23,596	2.36x	1.02x
web-Google (R4)	875,713	5,105,039	28,053	12,165	17,664	7,927	1.58x	1.53x
cit-Patents (R5)	3,774,768	16,518,948	80,223	237,968	4,879	2,992	16.44x	79.53x
cit-HepPh (R6)	34,546	421,578	651	214	312	141	2.09x	1.52x
soc-LiveJournal1 (R7)	4,847,571	68,993,773	833,189	703,077	572,705	385,912	1.45x	1.82x
soc-Pokec (R8)	81,306	1,768,149	82,525	66,319	73,060	34,214	1.13x	1.94x
com-YouTube (R9)	1,134,890	2,987,624	398,794	91,859	177,555	114,003	2.25x	0.81x
com-Orkut (R10)	3,072,441	117,185,083	5,001,263	326,534	3,141,124	325,351	1.59x	1.00x
Washington-RLG (S0)	262,146	785,920	162,792	132,482	287,390	99,410	0.56x	1.33x
Genrmf (S1)	2,097,152	10,403,840	2,138	2,900	2,685	2,503	0.79x	1.15x

4 Evaluation

4.1 Experiment Setup

Applications and datasets. We evaluated our workload-balanced push-relabel algorithm on both the *maximum flow/ minimum cut* and the *bipartite matching problems*. We used Washington and Genrmf synthesized networks from the DIMACS 1st Implementation Challenge [13] and 10 real-world networks from SNAP [16] for the maximum flow/ minimum cut problem, while we selected 12 real-world bipartite graphs from KONECT [15] for bipartite matching problem. Since the real-world networks from SNAP have no specified source and sink vertices, we previously used breadth-first-search to find 20 pairs of distinct source and sink vertices with the top 25% longest diameters. We add a *super-source* and a *super-sink* connecting to all the 20 sources and sinks, respectively, to compute the multi-source multi-sink maximum flow problem. In the bipartite matching, the super-source and super-sink connect to two groups of vertices, respectively. The detailed pair information can be found in our open-sourced repository.

Implemented algorithms. Since the prior work [9] did not specify which graph representation it used, we only measured our workload-balanced algorithm with RCSR and BCSR. We implemented the below algorithms:

- TC+RCSR. The thread-centric approach with the RCSR.
- TC+BCSR. The thread-centric approach with the BCSR.
- VC+RCSR. Our vertex-centric approach with the RCSR.
- VC+BCSR. Our vertex-centric approach with the BCSR.

Measuring machine. We ran all the experiments on the Intel i9-10900k 20-core processor @ 3.7GHz with 128GB DDR4 RAM and an Nvidia RTX 3090 GPU. The number of blocks and block size of the kernel configuration are 1024 and 82, respectively.

Table 2. The execution time of different push-relabel algorithms across 13 graphs for bipartite matching. The bold font represents the best execution time among the three designs.

Graph	$ L $	$ R $	$ E $	Maximum Flow	Execution Time (ms)				Speedup (TC/VC)	
					TC+RCSR	TC+BCSR	VC+RCSR	VC+BCSR	on RCSR	on BCSR
corporate-leadership (B0)	24	20	99	20	0.15	1.10	0.18	1.00	0.83x	1.1x
Unicode (B1)	614	254	1,255	188	14.31	13.92	9.12	10.84	1.57x	1.28x
UCforum (B2)	899	522	7,089	516	28.87	28	17.04	19	1.69x	1.47x
movielens-u-i (B3)	7,601	4,009	55,484	2,836	289	248	139	169	2.08x	1.47x
Marvel (B4)	12,942	6,486	96,662	5,057	299	331	197	240	1.52x	1.38x
movielens-u-t (B5)	16,528	4,009	43,760	3,258	602	567	305	381	1.97x	1.49x
movielens-t-i (B6)	16,528	7,601	71,154	5,882	621	570	400	422	1.55x	1.35x
YouTube (B7)	94,238	30,087	293,360	25,624	157,941	110,626	35,003	36,136	4.51x	3.06x
DBpedia_locations (B8)	172,079	53,407	293,697	50,595	488,939	417,497	102,373	109,098	4.78x	3.83x
BookCrossing (B9)	340,523	105,278	1,149,739	75,444	184,985	128,079	64,337	70,531	2.88x	1.82x
stackoverflow (B10)	545,195	96,678	1,301,942	90,537	864,168	631,918	309,294	267,487	2.79x	2.36x
IMDB-actor (B11)	896,302	303,617	3,782,463	250,516	427,357	335,800	162,909	160,420	2.63x	2.09x
DBLP-author (B12)	5,624,219	1,953,085	12,282,059	1,952,883	337,366	1,121,432	371,953	609,259	0.91x	1.84x

4.2 Performance

We measured the kernel execution time of different algorithms. Table 1 and Table 2 show the performance of the algorithms in maximum flow and bipartite matching tasks, respectively. At first glance, our vertex-centric approach can improve the execution of both RCSR and BCSR in the two tasks. It gains an average 2.49x and 7.31x execution time speedup in maximum flow tasks on RCSR and BCSR, respectively. In bipartite matching, our VC approach achieves 2.29x and 1.89x execution time speedup on RCSR and BCSR. We observed two critical points in these experiments. First, The VC method is suitable for graphs with a high standard deviation of degree, and these graphs should be a manageable size, or the overhead of synchronization can offset the workload balance benefits of VC. For example, the B0-B2 graphs are too small, so the speedup is marginal. The performance degradation of graphs R0-R2, S0, S1, and B12 comes from the characteristics of the graph itself. The Amazon0302 (R0) is the co-purchased product network on Amazon.com. If a product v is frequently co-purchased with product u , the graph contains a directed edge from i to j . In this network, almost all nodes are within the same SCC (Strongly Connected Component), and the degrees of these nodes are very close to each other; hence, the workload of this network is naturally balanced. The R1 and R2 are two real-world road networks, so their maximum degree is less than 10. Since we use a tile (usually more than 32 threads per tile) to find neighbors of an active vertex, the active node with a small degree will idle most of the threads within a tile and decrease the utilization of the GPU. On the contrary, if there are some nodes whose degrees are enormous, our vertex-centric approach can effectively alleviate the imbalanced workload, such as R5, B7, and B8.

The second observation is that the BCSR outperforms the RCSR in the maximum flow problem. Nevertheless, there is a slight degradation in the bipartite matching problem. This result recalls the notion in Section 3 that the RCSR has constant accessing time of backward edges, while the BCSR has a better locality than the RCSR. In the bipartite matching task, since the average degree of a graph is higher than that in the maximum flow task, the RCSR performs well

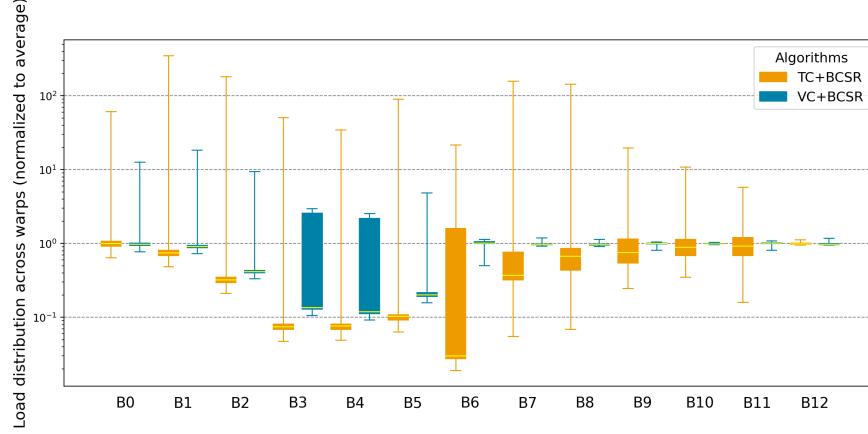


Fig. 3. The workload distribution of the bipartite matching problem across 13 bipartite graphs.

because of its fast access to backward edges without searching. On the other hand, the better locality of BCSR can reduce the memory request number by memory coalescing, leading to better performance.

4.3 Workload Analysis

To prove that the performance improvement of our vertex-centric approach comes from the workload balance during execution, we used the method proposed in [1] and measured each warp’s execution time. By assigning the delegated (first) thread within a warp for recording the execution timestamp in the kernel function, we can draw the workload distribution for the TC and VC approach across 13 bipartite graphs in Figure 3. Note that both configurations operate on RCSR. Based on the result, we can make two observations.

First, even though the chart does not directly indicate a shorter execution time for VC (after being normalized by the mean), the vertex-centric approach does indeed reduce the standard deviation of execution times across warps, thus achieving a more even distribution of work.

Second, in smaller graphs, even if we equalize the uneven distribution of work among warps, the overall performance may still decrease due to excessive synchronization. The B0, B1, and B2 are the cases.

5 Related Works

Load balancing is the key factor to achieve high performance when computing graph algorithms on GPUs. Several fundamental graph algorithms have developed their load balancing heuristics, such as SSSP [3] (single-source-shortest-

path), BFS(breadth-first-search) [11], triangle counting [12], clique enumeration [1], and graph neural network [23]. On the other hand, some works simplified the load balancing rule by proposing a unified abstraction and programming model [17, 22].

6 Conclusion

In the paper, we first identify the challenges and workload imbalance issue of the traditional parallel push-relabel algorithm with the derived computation model. The introduction of enhanced compressed sparse representation data structures, namely RCSR and BCSR, mitigates the memory space challenges posed by large graphs and optimizes memory access patterns for diverse graph characteristics. Our vertex-centric approach to the parallel push-relabel algorithm further rectifies the issues of workload imbalance and achieves an average 3.45x runtime speedup in maximum flow and bipartite matching tasks.

References

1. Almasri, M., Chang, Y.H., El Hajj, I., Nagi, R., Xiong, J., Hwu, W.m.: Parallelizing maximal clique enumeration on gpus. In: 2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 162–175. IEEE (2023)
2. Anderson, R., Setubal, J.C.: A parallel implementation of the push-relabel algorithm for the maximum flow problem. *Journal of parallel and distributed computing* **29**(1), 17–26 (1995)
3. Davidson, A., Baxter, S., Garland, M., Owens, J.D.: Work-efficient parallel gpu methods for single-source shortest paths. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium. pp. 349–359. IEEE (2014)
4. Dinitz, Y.: Dinitz’s algorithm: The original version and even’s version. In: *Theoretical Computer Science: Essays in Memory of Shimon Even*, pp. 218–240. Springer (2006)
5. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)* **19**(2), 248–264 (1972)
6. Ford, L.R., Fulkerson, D.R.: Maximal flow through a network. *Canadian journal of Mathematics* **8**, 399–404 (1956)
7. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum-flow problem. *Journal of the ACM (JACM)* **35**(4), 921–940 (1988)
8. Harris, M., et al.: Optimizing parallel reduction in cuda. *Nvidia developer technology* **2**(4), 70 (2007)
9. He, Z., Hong, B.: Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). pp. 1–10. IEEE (2010)
10. Hong, B.: A lock-free multi-threaded algorithm for the maximum flow problem. In: 2008 IEEE International Symposium on Parallel and Distributed Processing. pp. 1–8. IEEE (2008)
11. Hsieh, C.Y., Cheng, P.H., Chang, C.M., Kuo, S.Y.: A decentralized frontier queue for improving scalability of breadth-first-search on gpus. In: 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1–6. IEEE (2023)

12. Hu, L., Guan, N., Zou, L.: Triangle counting on gpu using fine-grained task distribution. In: 2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW). pp. 225–232. IEEE (2019)
13. Johnson, D.S., McGeoch, C.C., et al.: Network flows and matching: first DIMACS implementation challenge, vol. 12. American Mathematical Soc. (1993)
14. Khatiri, J., Samar, A., Behera, B., Nasre, R.: Scaling the maximum flow computation on gpus. *International Journal of Parallel Programming* **50**(5-6), 515–561 (2022)
15. Kunegis, J.: Konect: the koblenz network collection. In: Proceedings of the 22nd international conference on world wide web. pp. 1343–1350 (2013)
16. Leskovec, J., Sosič, R.: Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* **8**(1), 1–20 (2016)
17. Liu, H., Huang, H.H.: {SIMD-X}: Programming and processing of graph algorithms on {GPUs}. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). pp. 411–428 (2019)
18. Lyuh, C.G., Kim, T.: High-level synthesis for low power based on network flow method. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **11**(3), 364–375 (2003)
19. Qian, J., Zhou, Z., Gu, T., Zhao, L., Chang, L.: Optimal reconfiguration of high-performance vlsi subarrays with network flow. *IEEE Transactions on Parallel and Distributed Systems* **27**(12), 3575–3587 (2016)
20. Rockafellar, R.T.: Network flows and monotropic optimization, vol. 9. Athena scientific (1999)
21. Vineet, V., Narayanan, P.: Cuda cuts: Fast graph cuts on the gpu. In: 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops. pp. 1–8. IEEE (2008)
22. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: A high-performance graph processing library on the gpu. In: Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming. pp. 1–12 (2016)
23. Wang, Y., Feng, B., Li, G., Li, S., Deng, L., Xie, Y., Ding, Y.: {GNNAdvisor}: An adaptive and efficient runtime system for {GNN} acceleration on {GPUs}. In: 15th USENIX symposium on operating systems design and implementation (OSDI 21). pp. 515–531 (2021)
24. Wu, J., He, Z., Hong, B.: Efficient cuda algorithms for the maximum network flow problem. In: GPU Computing Gems Jade Edition, pp. 55–66. Elsevier (2012)