15-618 S25
Parallelizing Push-Relabel Maximum Flow Algorithm using CUDA

Names: Alena Lu, Ankita Chatterjee
Andrew IDs: alenalu, ankitac
https://achstar.github.io/max-flow-gpu/

1 Summary

Our project focuses on the implementation and parallelization of Goldberg-Tarjan's push-relabel algorithm for solving the maximum flow problem in directed flow networks. We first developed a sequential baseline implementation of push-relabel, which served as a correctness reference and a performance baseline for further optimization efforts. We then implemented and optimized a parallel version of push-relabel in CUDA, and also incorporated the global relabeling heuristic. Our optimizations of the implementation are detailed in this paper, as well as the speedup achieved for each optimization made. We are able to achieve up to 22x speedup for large graphs, and we analyze the performance benefits of each experiment we performed in our results.

2 Background

2.1 Maximum Flow Problem

The maximum flow problem seeks to determine the greatest amount of flow that can be transported from a designated source node to a sink node in a flow network, subject to capacity constraints on the edges. This has many applications, such as network routing, scheduling, bipartite matching, and image segmentation in computer vision. The main inputs to the problem are the graph G, as well as the source (s) and sink (t) nodes. Each edge has some capacity $c_f(u, v)$, and the output is the max flow which satisfies 2 constraints:

1. **Capacity**: The flow through an edge must be between 0 and the edge's capacity, inclusive
$$0 \leq f(u, v) \leq c(u, v) \quad \forall (u, v) \in E$$

2. **Flow conservation**: For all vertices except the source and sink, the flow in must equal the flow out of the vertex:
$$\forall u \notin \{s, t\}, \quad \sum_{v \in V} f_{uv} = \sum_{v \in V} f_{vu}.$$

2.2 Push-Relabel Algorithm

There are several algorithms that solve the max flow problem, notably Ford-Fulkerson, Edmonds-Karp, Dinic's, and Goldberg's push-relabel algorithm. However, our project focuses mainly on the push-relabel algorithm, which has become a fundamental subroutine in fast graph algorithms research. For example, it has been used in faster algorithms for computing expander decompositions [6] and deterministic minimum cut [1].

Unlike path-augmenting methods that find complete paths from source to sink, push-relabel maintains a preflow, allowing excess flow at intermediate vertices. Flow is incrementally "pushed" from vertices with excess toward the sink, based on a labeling function that essentially approximates the distance to the sink.

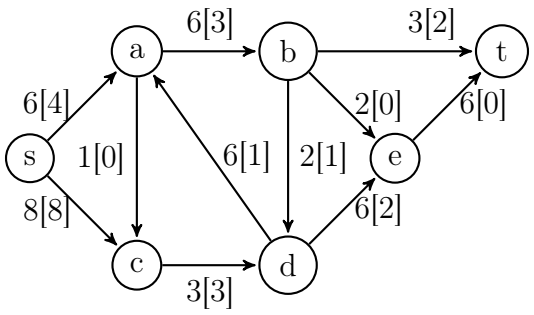

Figure 1: Example of a preflow network, taken from 15-451 course notes

The algorithm operates through two primary local operations:

1. **Push:** push excess flow from a higher-labeled vertex to a lower-labeled adjacent vertex if there is residual capacity along the edge.

2. **Relabel:** If no neighbor is available to push to, increment the vertex's label by 1.

Push-relabel terminates when no internal vertices have excess, which satisfies both conservation and capacity constraints. The excess value at a vertex is:

$$e_f(v) = \sum_{uv \in E} f(uv) - \sum_{vw \in E} f(vw).$$

The excess at the sink node is the value of the max flow when the algorithm terminates.

2.3 Parallelizing Push-Relabel

There is inherent data parallelism in push-relabel, since its two core operations are highly parallelizable across nodes.

1. **Push:** Active vertices with positive excess can attempt to push flow simultaneously to their neighbors. As long as two vertices are pushing to different destinations, their actions are completely independent. Even when pushing to the same neighbor, race conditions can be mitigated using atomic operations and synchronization methods.

2. **Relabel**: Vertices that are unable to push can independently relabel themselves. Since relabeling depends only on the local neighborhood of a vertex and does not require immediate synchronization with distant parts of the graph, it can be parallelized efficiently.

Each active vertex can handle these operations independently, but we run into potential issues with synchronization. Multiple pushes may simultaneously update the excess of a destination vertex, which means we need atomic operations to protect these critical regions. Updating the capacity of the edge, as well as the reverse edge for the residual graph, can also bring about data races. Furthermore, when we push, we must observe the *most recent* labels of neighboring vertices, which could also bring about issues if two vertices are trying to push to the same destination node.

The maximum flow is a graph algorithm, so the locality in this problem is inherent to graphs themselves. If we access a node u, we are likely to access its neighbors soon, or more specifically its outgoing edges; so we may want to store values specific to its outgoing edges (like residual flow values) in adjacent memory locations to optimize for that locality.

Push-relabel is not particularly amenable to SIMD execution; especially for the CUDA programming model, there is bound to be divergence between the conditional

paths taken by threads within warps. If each thread processes a different node, each node likely has different labels, excess, outgoing edges, etc.; whether a thread pushes, relabels, or does nothing is completely dependent on all of these attributes, and even the execution time for multiple pushing threads can vary widely depending on the contention at the memory locations that are being modified. As a result, it was challenging to improve the warp occupancy of our program, especially on sparse graphs.

3 Approach

3.1 Initial CUDA Implementation and Optimizations

We implemented the CUDA version of push-relabel by referencing the asynchronous lock-free algorithm described in [7]. All our sequential and input processing code is in C++. Our implementation runs on the NVIDIA RTX 2080 GPUs on the GHC machines.

3.1.1 Implementation and Mapping of Structures

In terms of representing the graph, we started with an adjacency matrix of capacities. In this implementation, the capcities are stored as an $n \times n$ matrix. The benefits of the adjacency matrix are that there is constant time access and simple indexing. However, there is a large memory limitation since most of the matrix will be empty if there doesn't exist an edge between a row and a column. Furthermore, it is quite infeasible to store an $n \times n$ matrix for much larger graphs, so the adjacency matrix implementation scales very poorly. In fact, for our larger test cases (100K nodes, 500K edges), we are unable to initialize the adjacency matrix because there is not enough device memory on the GPU.

As a result, we used a compressed sparse row (CSR) representation of the graph, which has an array that stores the starting index of each vertex's adjacency list, and a flattened adjacency list array that stores the destination vertices starting at each index.The CSR representation is much more memory efficient and allows our implementation to run on very large graphs. To mitigate the longer access times, we created a reverse edge array that holds the indices of the reverse edge that we use in the residual graph, preserving the $O(1)$ access time but still using much less memory than the original $n \times n$ matrix. Thus, we decided to use the CSR representation in our final implementation.

Due to concerns regarding warp divergence and low occupancy, since as previously mentioned, the push relabel algorithm when parallelized per node introduces varying and unpredictable conditional paths, we considered other axes of parallelization. One approach we considered, inspired by [2] was to assign each warp to a single node, and parallelize the processing of each outgoing edge. When each thread takes care of a single node, it must find the neighboring node with the lowest label, and does so with a simple linear search. The warp-per-vertex approach would instead use a parallel reduction algorithm to find the minimum, and then a designated thread would do the pushing and relabeling computations. In theory, this would improve workload balance across individual warps, since each thread in the warp would be performing similar computations. However, when implementing this approach, we were unable to achieve a fully correct implementation, so we did not end up following through with it. Additionally, for most of the real-world graphs we tested on, the average degree (i.e. number of outgoing edges for each node) was less than 32; indicating that for many kernels, there would be threads within a warp that would not be doing any work. Due to this observation and our difficulty in achieving correctness for this approach, we decided to stick with our initial axis of parallelization, which was to assign each thread to a node. With graphs that have higher average degrees, however, we can probably expect that GPU utilization would be better and so would overall performance.

3.1.2 Overview of Algorithm

The algorithm maps each vertex to an individual CUDA thread, which allows multiple vertices to perform push and relabel operations concurrently. Global synchronization across the entire graph is minimized through the lock-free implementation, and the algorithm uses CUDA's native atomic operations to manage concurrent updates safely. The general algorithm is detailed below.

Algorithm 1 Parallel push-relabel algorithm using CUDA

1: Initialize e, h, cf, and $ExcessTotal$
2: Copy e and cf from the CPU's main memory to the CUDA device global memory
3: **while** $e(s) + e(t) < ExcessTotal$ **do**
4: copy h from the CPU main memory to the CUDA device global memory
5: call `push-relabel()` kernel on CUDA device
6: copy cf, h, and e from CUDA device global memory to CPU main memory
7: call `global-relabel()` on CPU
8: **end while**

Capacities for each edge (cf), excesses for each vertex (e), labels for each vertex (h), and $ExcessTotal$ are initialized, as well as the initial preflow from the source node. $ExcessTotal$ tracks the initial total amount of excess flow pushed out from the source. Our termination condition does not keep track of the number of active nodes, but rather the value of $ExcessTotal$ and the excess at the source and sink nodes. This termination condition preserves the conservation of flow constraint, since the sum of the excess at the sink and the excess that gets pushed back to the source is equal to the $ExcessTotal$ value.

In terms of the `push-relabel()` kernel, each vertex is assigned a CUDA thread, through which it scans its outgoing edges to find the neighbor with the *lowest* label. This is different from the general push-relabel algorithm, which finds an arbitrary neighbor with a lower label than itself, so not necessarily the neighbor with the lowest label. If such a neighbor exists and there is excess to be pushed, the vertex will push the minimum of its excess and the edge capacity through the edge and to the destination node. The edge in the residual graph is also updated, in case we need to reverse the operation and push flow back (since it is unable to move any further towards the sink). If the vertex is unable to push, then it will relabel itself to one more than the minimum neighbor's label. Details about our push-relabel() kernel are below.

Algorithm 2 push-relabel() kernel implementation

```
1: cycleNum = N
2: while cycle > 0 do
3:     if e(u) > 0 and label(u) ≤ N + 1 then
4:         e' ← e(u)
5:         min_label ← ∞
6:         for (u, v) ∈ E_f do
7:             if label(v) < min_label then
8:                 v' ← v
9:                 min_label ← label(v)
10:            end if
11:        end for
12:        if label(u) > min_label then
13:            d ← min(e', cf(u, v'))
14:            AtomicAdd(cf(v', u), d)
15:            AtomicSub(cf(u, v'), d)
16:            AtomicAdd(e(v'), d)
17:            AtomicSub(e(u), d)
18:        else
19:            label(u) ← min_label + 1
20:        end if
21:    end if
22:    cycle ← cycle − 1
23: end while
```

3.1.3 Synchronization

Race conditions naturally arise when multiple vertices simultaneously push flow to the same neighbor. The algorithm addresses these issues by using atomic operations to update the capacity of the edge, the capacity of the reverse edge in the residual graph, the excess of the source node, and the excess of the destination node.

Beyond simply resolving race conditions, the use of atomic operations enables our implementation to be lock-free. Lock-free synchronization eliminates the possibility of thread deadlock and livelock, which could happen when a push/relabel happens on both edge directions of two nodes. Threads avoid being serialized on locks and can continue progressing independently with the atomic operations. However, one of our optimizations later on makes it so that our implementation of this algorithm

is not truly lock-free; we sacrifice that in order to preserve correctness and reduce staleness of shared values, which ultimately allowed the algorithm to converge faster.

3.2 Global Relabel Heuristic

As recommended by several papers [3] we looked at when determining how to implement our CUDA algorithm, we used the global relabeling heuristic, detailed in the pseudocode below.

Algorithm 3 Global relabeling heuristic

```
1: Initialize empty queue Q
2: for each u ∈ V do
3:     u visited ← false
4: end for
5: Q.enqueue(t), where t is the sink node
6: t visited ← true
7: while Q not empty do
8:     u ← Q.dequeue
9:     current ← label(x)
10:    for (v, u) ∈ E_f : v not visited do
11:        label(v) ← label(u) + 1
12:        v visited ← true
13:        Q.enqueue(v)
14:    end for
15: end while
```

This heuristic is run every N cycles of the push-relabel kernel, and is intended to prevent the node labels from growing out of control; it relabels all nodes to be their smallest possible labels (such that there is still a possible path to push to the sink.) In other words, the nodes are relabeled to their shortest "distance" from the sink. It essentially consists of a BFS searching for all *incoming* nodes to a particular node, where if the incoming node is not visited, that node is relabeled to be one higher than the current node. We ran this heuristic sequentially on the CPU. It vastly improved the time needed for the algorithm to converge on a max flow value; in fact, without this heuristic, most of our large test cases did not complete within a reasonable time frame, so it's possible that the heuristic is necessary for the algorithm to be complete and correct.

3.3 Further Optimization

3.3.1 Graph Precoloring

One optimization technique we tried early on was the graph precoloring mentioned in [7]. This method addresses synchronization overhead and atomic operation bottlenecks by structuring the computation such that we eliminate race conditions altogether. In the precoloring-based synchronized algorithm, the edges of the residual graph are precolored before any pushing begins. The coloring ensures that no two edges sharing a common vertex are assigned the same color. As a result, during execution, all edges of a given color can be processed in parallel without any possibility of race conditions because no two threads will simultaneously modify the same vertex's excess. Because race conditions are eliminated, we don't need the atomic operations anymore.

However, in our experimentation with graph precoloring, the overhead for actually coloring the edges seems to outweigh the benefit of the optimization. The initial coloring cost, which is $O(VE)$ is expensive for smaller graphs where the computation of the max flow is relatively fast. Furthermore, if the graph contains vertices with high degrees, like if the source or sink is connected to many nodes, many colors are needed to avoid conflicts. This reduces parallelism within each color group, as fewer edges remain active at a time. The precoloring algorithm is outlined below:

Algorithm 4 Graph coloring pseudocode

```
1: precolor_graph()
2: cycleNum = N
3: while cycle > 0 do
4:     l_flag = 0
5:     if e(u) > 0 and label(u) <= N + 1 then
6:         l_flag = 1
7:         e' ← e(u)
8:         min_label ← ∞
9:         for (u, v) ∈ E_f do
10:            min_label ← label(v)
11:            if label(v) < min_label then
12:                l_flag = 0
13:                v' ← v
14:                break
15:            end if
16:        end for
17:    end if
18:    barrier()
19:    for all cr ∈ COLORS do
20:        if color(v') == cr && l_flag = 0 then
21:            d ← min(e', cf(u, v'))
22:            cf(v', u)+ = d
23:            cf(u, v')− = d)
24:            e(v')+ = d)
25:            e(u)− = d)
26:        end if
27:        barrier()
28:    end for
29:    if l_flag == 1 then
30:        label(u) ← min_label + 1
31:    end if
32:    cycle ← cycle − 1
33: end while
```

Additionally, there is a bottleneck with the barriers that are needed for global synchronization between the push, color, and relabel phases. After processing all edges of one color class, all memory updates must be globally visible and no thread should proceed to the next color prematurely. On CUDA architectures, however, there

is no native global barrier available within a single kernel execution. CUDA threads can synchronize at the block level using `__syncthreads()`, but not across different blocks. Thus, we implemented the global barrier through kernel relaunches. Because a kernel must be relaunched after processing each color, the total number of kernel launches scales with the number of colors assigned during precoloring. In graphs with high-degree vertices or very irregular structure, the number of colors can become extremely large, and thus the number of relaunches will scale poorly. We observed that each iteration took extremely long compared to the version without precoloring (to the point where it was slower than our sequential implementation), because the global thread barrier is implemented by iteratively launching the kernels, introducing far more overhead than performance benefit. Ultimately, our final implementation did not make use of graph precoloring.

3.3.2 Cooperative Groups

Another optimization we tried, which may have resolved our synchronization issues for the precoloring approach had we decided to pursue it further, was the use of cooperative groups, an extension to the CUDA programming model that organizes groups of communicating threads. Before switching to cooperative groups, our code was structured such that the relabeling step needed to be an atomic instruction, as occasionally other threads would read the label before relabeling and push to the node after it was relabeled, which resulted in race conditions and the algorithm often converging to the wrong value or never converging at all. Using an atomic instruction here to make our algorithm complete and correct resulted in a significant increase in our computation time due to contention at the label's memory location, so the natural progression was to look for an alternative way to synchronize that aspect of the computation.

Using the cooperative groups API, we were able to define grid groups that represented all threads launched in a single grid, with the ability to synchronize across the grid. This was ideal for us because we were able to synchronize the grid at the end of each cycle, ensuring that all threads finish processing nodes for a given cycle before any thread moves on. Before, this synchronization was not possible, resulting in nodes being updated with stale information and unnecessary relabeling (with the algorithm thus taking longer to converge.) Now of course, this made it so that our implementation of this algorithm was not truly lock-free, as it is possible that threads will not do work while waiting for other threads to finish their computation at each barrier. However, the reduction in staleness of shared values actually allowed the algorithm to converge faster, demonstrating that a fully lock-free solution is not

necessarily the best choice for all parallelization problems.

We observed the best performance using a grid size equivalent to the number of SMs on the GPU, and 1024 threads per block; when varying threads per block (and adjusting grid size proportionally such that the full SM is still used), we didn't see much of a performance difference. Our warp occupancy before using cooperative groups was actually about the same as after using cooperative groups (around 12 average active threads per warp) but the algorithm took far less time to converge, because many active threads were doing unnecessary work. We expected that this synchronization would lead to less warp divergence, since threads within a warp may be more likely to do similar work during each cycle; however, our active threads per warp staying the same showed otherwise. Still, the benefit from the algorithm converging faster led to almost 10 times speedup compared to our previous implementation on certain test cases; additionally, we no longer needed to make the relabel atomic, as the synchronization at the end of each cycle eliminated the race condition we observed previously.

4 Results

4.1 Testing and Traces

Our testing consisted of two parts, with the first part observing trends in varying the problem size, and the second with special application graphs. We measured our performance using speedup compared to the sequential implementation (essentially dividing wall-clock time for the CUDA implementation by wall-clock time for the sequential implementation.) We also looked at warp occupancy, but observed that this did not change significantly across our optimizations and test cases (the average consistently stayed between 10 and 15 threads per warp for our large test cases.)

The problem size analysis consisted of experiments across 4 different features of the graph. Specifically, we varied the number of nodes while holding the number of edges and the capacities of each edge constant, varied the number of edges while keeping the number of nodes and edge capacities constant, and also varied the variance of the capacities across each edge while keeping the graph size consistent. Our last experiment tested how the execution times would vary with increasing graph sizes with a consistent ratio of nodes to edges. For instance, 1000 nodes to 10000 edges (10x more edges than nodes), then 10000 nodes to 100000 edges, etc. These results are detailed in the speedup analysis section below.

These graphs were generated using a simple Python script that generated random edges, and also specified random source and sink nodes for each generation attempt. The edge capacities were also initialized randomly, at varying ranges. However, we found that it was difficult to control specific aspects of the graph with pure randomization, so we also manually generated graphs with specific structures, such as cyclic graphs where each node is connected to another.

Our baseline implementation is a single-threaded CPU implementation of push-relabel, which goes through each active node and sequentially computes the max flow. For the application graphs, the main metric we are observing is the speedup, so we compared the sequential baseline with the initial CUDA implementation, as well as with the final CUDA implementation that had the cooperative groups and global relabel optimization.

4.2 Speedup Analysis

4.2.1 Problem Size Analysis

As mentioned earlier, the first part of our testing was to understand how varying different aspects of the graph would affect the execution time. Varying the problem size and density of the graphs is important to understanding how the algorithm scales in practice. For instance, denser graphs can exhibit better performance compared to sparser, smaller graphs in the parallel implementation due to better workload distribution. The results of our experiments are shown below.

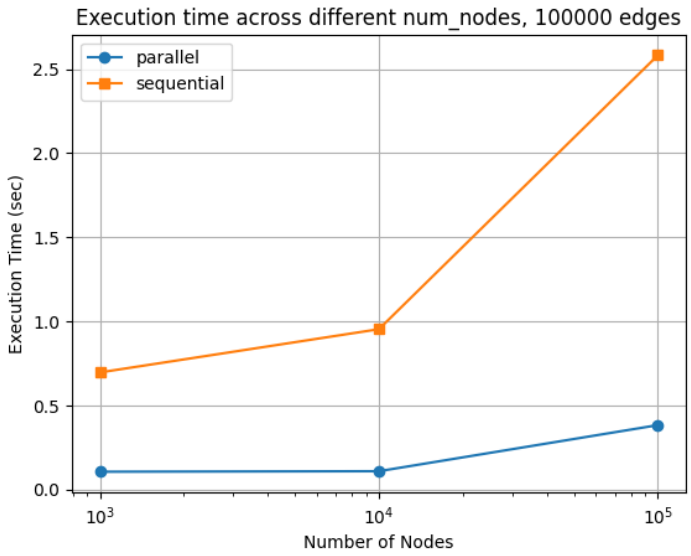

Figure 2: Varying across Nodes

Figure 2 shows the execution time of both the sequential and parallel CUDA implementations of push-relabel as the number of nodes is increased, with the number of edges fixed at 100,000 and capacities fixed at 5000. The execution time of the sequential algorithm increases significantly as the number of nodes grows. This is expected, and we observe that the sequential version scales very poorly when jumping between 10,000 and 100,000 nodes. We are able to achieve 6.5x-8x speedup with the CUDA implementation, but we also observe similar trends in the execution time increasing with the number of nodes.

The scalability of the parallel algorithm is also expected, but across all three experiments, we notice that the execution time for the parallel implementation is the worst for varying the number of nodes. This is expected, as since the number of edges remains constant, the total number of push operations required does not increase as dramatically as it would if the graph density grew with the node count. Thus, the algorithm benefits less from parallelization since there is inherently less work to do for each node.

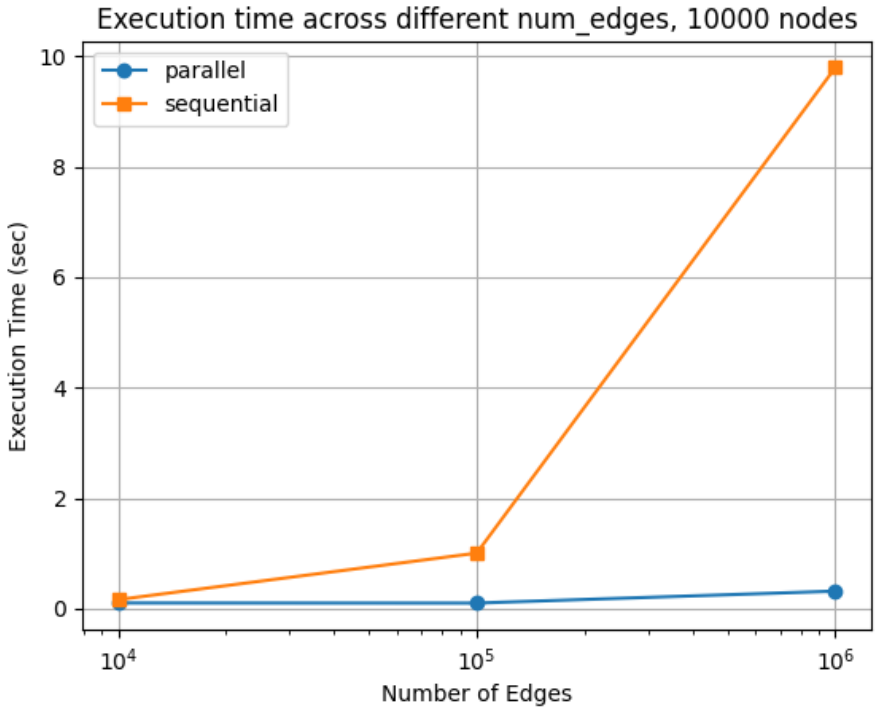

Figure 3: Varying across Edges

We observe similar trends for varying the number of edges, but our speedup is much better on denser graphs, compared to just increasing the number of nodes. The parallel advantage becomes much more pronounced because each thread processing a vertex has more work to do. Another interesting observation is the dip in execution time around 10,000 edges. This aligns with what is expected because the workload is more balanced with more edges, since each CUDA thread/node does more work in parallel, so time is not spent idling. We'd later see that with even larger graphs, however, this workload balance degrades due to larger variation in work done by each thread.

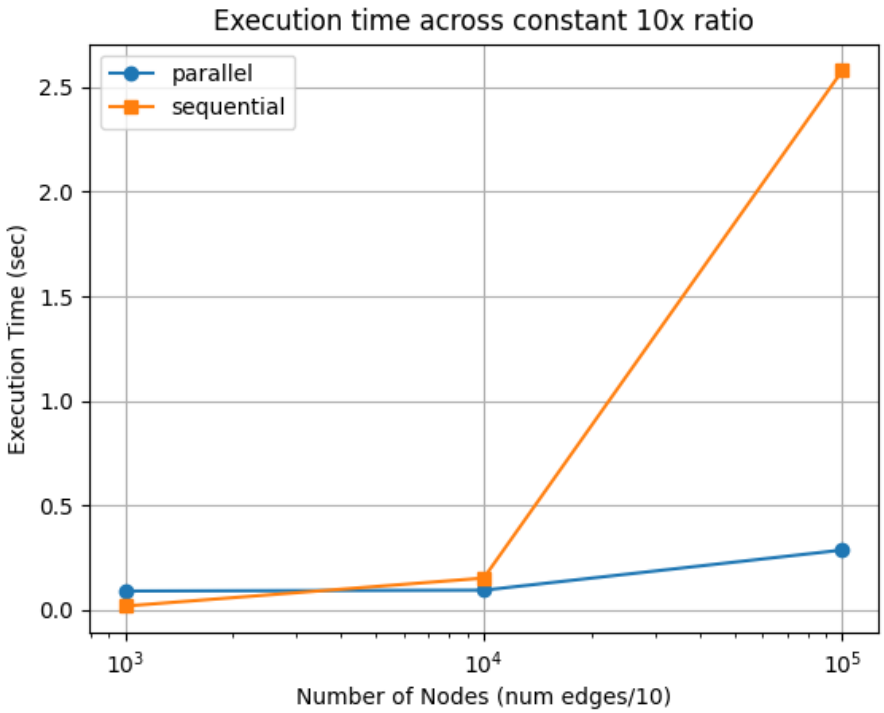

Figure 4: Increasing Graph Size

Figure 4 presents the execution time of both algorithms as the general graph size increases, while maintaining a constant edge-to-node ratio of 10. The capacities of the edges are also kept at a constant 5000. In this experiment, we mainly wanted to see how the algorithm would scale as the graph itself scales, not particular to nodes or edges. The results show that the sequential execution time grows rapidly with graph size and scales poorly, while the parallel CUDA implementation exhibits a much slower growth in execution time. The results demonstrate that the parallel version maintains effective scaling even as both nodes and edges grow together, but is outperformed by the sequential algorithm for smaller graphs. This makes sense, since the benefit we get from parallelization is more prominent in larger graphs as the CUDA implementation scales much better with increasing graph size. The slight upward trend in parallel execution time at larger graph sizes is expected, as higher total work is still distributed across the GPU. Nevertheless, the parallel algorithm's performance remains stable relative to the worsening performance of the sequential implementation.

4.2.2 Application Graphs

We also tested our implementation on a variety of real-world application graphs, obtained from the University of Waterloo's "Max-flow problem instances in vision" [4]

and SNAP (Stanford Network Analysis Project)'s Large Network Dataset Collection [5]. Since the SNAP graphs weren't configured for max flow problems, we just set the capacity for every edge to be 1; so convergence was pretty fast on these compared to other graphs that may have had varying flow values. We compared these against our baseline sequential single-threaded CPU implementation, as well as our CUDA implementation without synchronization and cooperative groups.

	CUDA-With Sync		CUDA-No Sync				
Test	Time (s)	Speedup	Time (s)	Speedup	Sequential Time (s)	Nodes	Edges
Stereo: BVZ-Tsukuba	28.274	21.66x	129.816	4.72x	612.416	100,000	500,000
Stereo: KZ2-Tsukuba	161.73	12.62x	432.159	4.72x	2040.4	200,000	1,200,0000
Surface Fitting: Bunny	60.54	2.773x	115.32	1.1x	123.68	805,802	5,040,834
SNAP: wiki-topcats	18.99	5.46x	94.32	1.1x	103.67	974,667	8,669,373
SNAP: web-Google	3.3	5.41x	15.43	1.16x	17.86	428,329	1,866,319

Table 1: Execution Times (s) On Application Graphs and Corresponding Sequential Speedup

Below are some graphs to visualize the speedups in the above table, separated by test case for better readability.

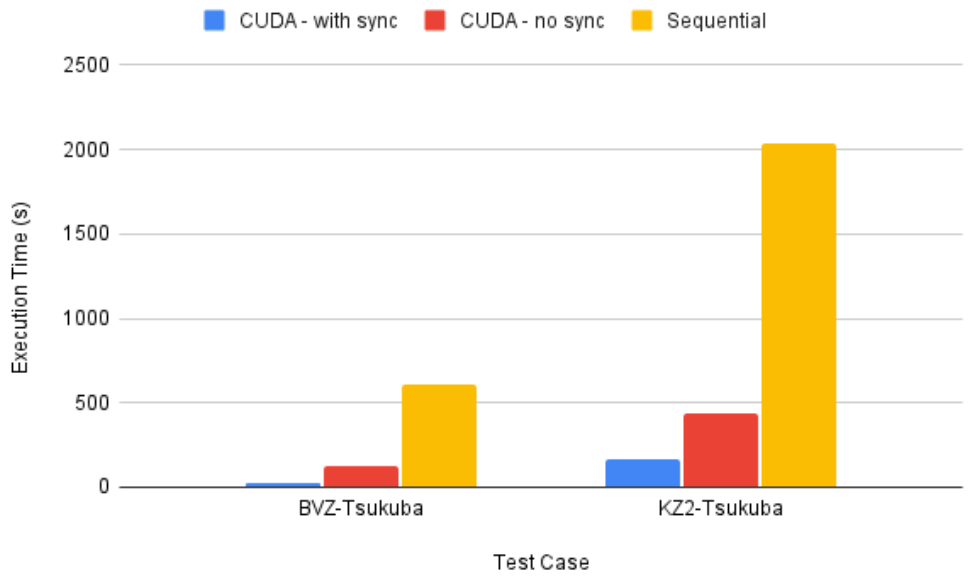

Figure 5: Comparing Execution Times Across Implementations for Tsukuba graphs

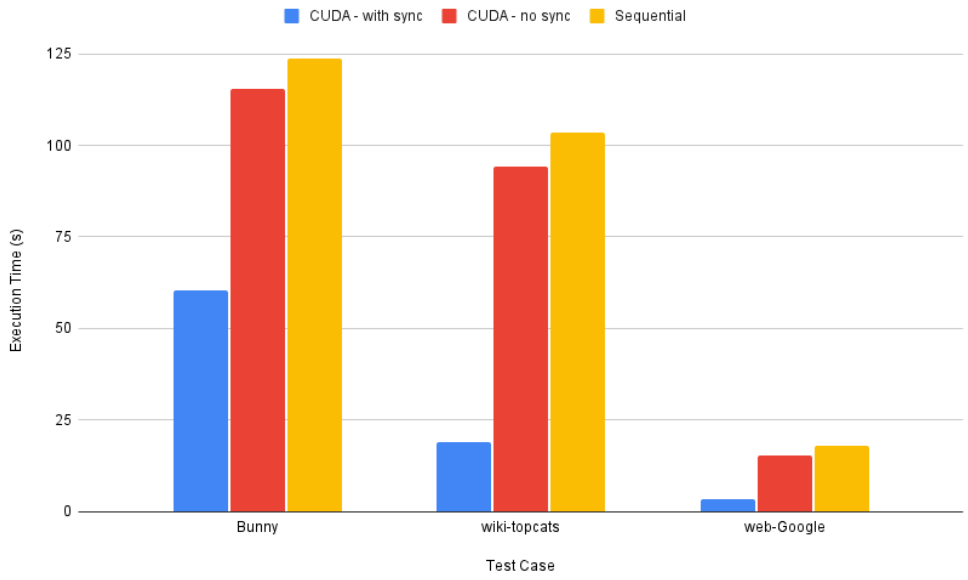

Figure 6: Comparing Execution Times Across Implementations for Bunny, SNAP graphs

We tested our implementation on 5 different real-world graph problems. The first two are stereo vision problems, represented by sequences of 16 graphs that represent the process of finding corresponding points between two images to estimate depth. BVZ-Tsukuba consists 4-connected grids with dense terminal edges from the source or to the sink, and KZ2-Tsukuba consists of graphs that contain two 4-connected grids with sparse connections between them (each vertex is connected to at most two vertices in the other grid) and dense terminal edges from the source or to the sink. To get the total time, we ran our code on all 16 graphs and summed the execution time. We took an approximate average of nodes and edges for each of the graphs to represent in the table (i.e. each of the 16 graphs had around 100,000 nodes and 500,000 edges and 200,000 nodes and 1,200,000 edges for BVZ and KZ2, respectively.) On the individual BVZ-Tsukuba graphs, we obtained a speedup of 30.1x at maximum and 15.6x at minimum, compared to the sequential. On the individual KZ2-Tsukuba graphs, we obtained a speedup of 17x at maximum and 11x at minimum.

The third one is a surface fitting vision problem, a 3D volumetric surface fit of the Stanford bunny. This comprised of a 6-connected grid with 800,000 nodes and 5,000,000 edges. We observed our poorest speedup on this graph, potentially because this graph has large variability in each node's degree (number of outgoing edges) and thus more workload imbalance between threads.

The other two graphs are SNAP graphs; one is a Google web graph with nodes representing web pages and edges representing hyperlinks between them, and the other is similar but with Wikipedia hyperlinks instead. We observe as the number of edges increases, the scaling of our program worsens, likely due to the workload imbalance issues mentioned above.

Our speedup was likely limited the most by poor SIMD utilization due to divergence, as well as workload imbalance and synchronization. Examining profiler results using Nsight compute revealed that our program was compute-bound, only 25% SM throughput. As mentioned earlier, our average number of active warps per thread ranged from 10 to 15, which is less than half occupancy. Furthermore, we timed the computation time for each warp per iteration and observed that the minimum warp execution time varied from 4-20 times lower than the maximum warp execution time, indicating that the workload is not balanced across warps. This would not be a huge concern if it weren't for the fact that we synchronize the grid at the end of each push-relabel cycle; as a result, workload imbalance across warps leads to many warps remaining idle and the GPU being underutilized. The profiler reported that much of our GPU underutilization was due to the usage of barriers, as expected.

4.3 Further Analysis

The execution time of our algorithm can be broken into three main components: initialization, including the preflow initialization of each graph and the population of the CSR data structures; the GPU kernel, which includes the push and relabel operations; and the global relabel time. Initialization time is typically around 0.5 seconds when the number of edges is greater than 100,000, which ranges from around 5 to 20% of the execution time. Global relabel time is miniscule in comparison, at maximum taking up 0.1% of the execution time. The kernel region takes the most amount of time, ranging from 80-95% of the execution time. This makes sense, since the kernel is where most of the computation should be taking place; however, this time could definitely be minimized by improving workload imbalance.

We do ultimately believe that our choice of machine target was sound. While our program suffered greatly from divergence and workload imbalance, it still allowed us to make use of the massive parallelism provided by the GPU architecture. Especially on larger graphs, despite the use of cooperative groups for synchronization, the GPU is still able to hide latency by switching between warps. CPU parallelism

is also ultimately bound by number of cores, and is unable to hide the latency of stalls caused by, for example, memory accesses. The fine-grained nature of GPU parallelism, since each warp is scheduled to execute on instruction at a time, allows for more efficient workload distribution; synchronization would still be required with CPU parallelism, and if the same model is followed where each thread handles computation for one node at a time, the coarse-grained nature of each thread's task would likely result in even more severe workload imbalance. Additionally, the sheer number of threads that can be launched on the GPU make up for the loss in throughput cause by warp divergence and poor GPU utilization.

We were able to implement a correct OpenMP CPU parallel version of push-relabel, but because it was not optimized, it is not a point of comparison in our experiments. It was a naive implementation that used locks, and while it far outperformed the CUDA and sequential implementation on small test cases, the synchronization overhead associated with locks deteriorated its performance for much larger graphs, to the point where we felt the comparison was not meaningful. A better comparison would have been an asynchronous lock-free OpenMP implementation, which can be used in future experiments as a baseline.

5 Work Distribution

The work distribution for this project was a 50/50 split among both partners. Alena worked primarily on implementing the sequential baseline, getting the initial asynchronous lock-free CUDA implementation to work correctly, generating tests for problem size analysis, and the OpenMP implementation. Ankita also worked on implementing the CUDA implementation, wrote the initial attempt at the graph precoloring optimization, the cooperative groups optimization, and data collection on the stereo, surface fitting, and SNAP traces. Both team members worked on the report and poster together, and spent time debugging the whole implementation together.

References

[1] Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. *SIAM J. Comput.*, 49(1):1–36, 2020.

[2] Chou-Ying Hsieh, Po-Chieh Lin, and Sy-Yen Kuo. Engineering a workload-balanced push-relabel algorithm for massive graphs on gpus. *arXiv preprint arXiv:2404.00270*, 2024.

[3] Agnieszka Lupinska. Parallel implematation of flow and matching algorithms. *CoRR*, abs/1110.6231, 2011.

[4] University of Waterloo Department of Computer Science. Max flow problem instances in vision. https://vision.cs.uwaterloo.ca/data/maxflow.

[5] Stanford Network Analysis Project. Stanford large network dataset collection. https://snap.stanford.edu/data/index.html.

[6] Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2616–2635. SIAM, 2019.

[7] Jiadong Wu, Zhengyu He, and Bo Hong. Chapter 5 - efficient cuda algorithms for the maximum network flow problem. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 55–66. Morgan Kaufmann, Boston, 2012.