15-618 S25
Parallelizing Maximum Flow using CUDA Project Proposal

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

1 Summary

We plan to parallelize a maximum flow algorithm using CUDA, on the GHC machine GPUs (NVIDIA RTX 2080). Specifically, we plan to use the push-relabel max flow algorithm and compare the performance between sequential and GPU-parallel implementations.

2 Background

The max flow problem is a graph theory problem, where given a directed graph with two distinct source and sink nodes, as well as capacity constraints on each edge, we calculate the maximum amount of flow that can be sent from the source to the sink. The flow of the graph is an assignment of values to each edge that matches the following constraints: flow values for each edge are between 0 and the edge's capacity, and for every node other than the source and sink, the total incoming flow must equal the total outgoing flow. Max flow has many applications in different areas, such as segmentation in image processing, routing in VLSI design, bipartite matching, and resource allocation.

Goldberg's push-relabel algorithm, which solves the problem in $O(V^2E)$ complexity, works by maintaining a preflow in the network and gradually converting it into a valid max flow by locally pushing excess flow from active nodes to neighboring nodes and adjusting node labels (which represent estimates of distance to the sink) to guide flow in the correct direction. We selected this max flow algorithm as the push-relabel works on one node at a time, and depends only on the current node and its neighboring nodes. Thus, this operation can be parallelized for each node in the graph. Using CUDA, we plan to solve the max flow problem by using atomic operations to perform the push and relabel operations asynchronously.

Algorithm 1 Parallel push-relabel algorithm using CUDA pseudocode

```
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 the CPU's main memory
7:     call global-relabel() on CPU
8: end while
```

The above algorithm is outlined in [1], and we plan on implementing and optimizing it.

3 The Challenge

Parallelizing this algorithm involves assigning one thread for each vertex and letting the threads do push-relabel operations simultaneously, but race conditions can occur if push and relabel operations are applied to the same vertex simultaneously. As a result, two push operations may simultaneously and incorrectly modify the excessive flow of a single node, or a push operation may send flow to a vertex that is currently being relabeled by another thread, which would violate the constraint of the algorithm that flow is always pushed to neighbors with a lower label. We will need to handle these race conditions with atomic operations, and strategies to separate two edges that share a vertex. For instance, performing push-relabel operations on sets of independent nodes.

In terms of workload, each node needs to keep track of its excess value and label, and depends on its neighboring nodes. Accesses to flow values, labels, and residual edge capacities often involve random accesses across adjacency lists. Locality is limited due to irregular graph structures. However, optimizations that we can look into include precoloring the graph to group nodes initially and process them to maximize locality. The communication to computation ratio is relatively high, since both push and relabel operations are computationally light, while more time is spent on memory movement across nodes. Each push operation affects the source and the target nodes, and excess flow/residual capacities must be updated atomically to prevent race conditions. All of the above factors make this algorithm difficult to parallelize.

4 Resources

We will be implementing the parallel push-relabel algorithm outlined in "Efficient CUDA Algorithms for the Maximum Network Flow Problem" [1] in CUDA from scratch. We will be using the GHC machines, which have NVIDIA RTX 2080 GPUs and are sufficient for the goals of our project, as we are measuring the speedup of the algorithm compared to

a sequential implementation on the CPU in C++. Aside from this paper, we will also be referencing notes on max flow and the push-relabel algorithm from 15-451 lecture notes, and any additional papers will be cited.

5 Goals and Deliverables

5.1 Plan to Achieve

We expect to achieve the following:

1. A working (correct) sequential push-relabel max flow algorithm.
2. A working CUDA (correct, with non-negligible speedup) parallelization of the push-relabel max flow algorithm, using atomic operations to protect shared resources (residual and excess flow values.)
3. Compare execution time and compute speedup across implementations. Evaluate the performance of both implementations on different graph inputs (sparse vs. dense) and evaluate scalability across different graph sizes and varying number of threads per block.
4. Explain all findings from the above evaluation using concepts discussed in course material or other factors determined from additional research. We want to determine what kinds of workloads most benefit from GPU parallelization and how well we are able to scale with larger inputs or more threads per block.

5.2 Hope to Achieve

We hope to achieve the following:

1. Optimizations to the algorithm, including a synchronous approach with precolored graphs to reduce atomic operations and increase overall speedup
2. A working (correct) CPU-parallel implementation of the push-relabel max flow algorithm in OpenMP to provide another benchmark for our CUDA implementation.

6 Platform Choice

We have chosen CUDA for this project because the maximum flow algorithm (specifically the push-relabel implementation) involves multiple independent operations that can benefit from GPU parallelism. For example, the push and relabel operations can be done in parallel as long as atomic update methods are used. Especially for large graphs, we think the parallelization benefit on GPUs will be more compelling than it would be on multi-core CPUs; however, we hope to also compare against a CPU-parallel implementation of the same algorithm to verify this assumption.

7 Schedule

- Week 1: Read research papers to understand sequential and parallel versions of push-relabel algorithm in their entirety. Brainstorm ways to approach parallelization and outline a solution through pseudocode.
- Week 2: Write and test sequential implementation. Develop test cases, which will be varying sized input graphs, and define expected results.
- Week 3: Continue to develop test cases, and write first iteration of CUDA implementation. Evaluate performance and debug. Outline findings and roadblocks in milestone report.
- Week 4: Continue to debug and improve parallel implementation. Begin deeper analysis of results to prepare for final report. If time permits, prototype OpenMP implementation or alternative max flow algorithm.
- Week 5: Finalize parallel implementations (including any nice-to-haves that were started in Week 4). Collect profiling results for finalized implementations and analyze/explain findings for final report.

References

[1] Jiadong Wu, Zhengyu He, Bo Hong, Chapter 5 - Efficient CUDA Algorithms for the Maximum Network Flow Problem, Editor(s): Wen-mei W. Hwu, In Applications of GPU Computing Series, GPU Computing Gems Jade Edition, Morgan Kaufmann, 2012, Pages 55-66, ISBN 9780123859631, https://doi.org/10.1016/B978-0-12-385963-1.00005-8.

[2] CMU 15-451/651: Design & Analysis of Algorithms Lecture 14. Fall 2013.