15-618 Final Project Proposal

Betweenness Centrality with CPU and GPU

Isaiah Velez Yi-Ning Huang

March 25, 2026

Project Website: https://www.isaiahvelez.com/15618-Final-Project/

1 Summary

Our project is about graph processing on CUDA. We are implementing Betweenness Centrality (BC) and benchmarking an optimized, conflict-aware edge-parallel GPU implementation against both serial and multicore CPU baselines. As part of our analysis, we will document the optimization journey from a naive GPU kernel to our final optimized version, highlighting how we reduce or avoid conflicting writes to shared vertex states.

2 Background

We are evaluating parallel graph processing using the Betweenness Centrality algorithm. BC is computationally heavy and highly irregular, requiring parallel Breadth-First Searches (BFS) from multiple source nodes.

The core idea we want to explore is edge-parallel graph computation. In this style of execution, one thread is assigned to one edge, which makes the GPU mapping feel natural because large sparse graphs usually have many more edges than vertices. The catch is that edges are not independent. As the BFS frontier expands, multiple threads processing different edges will attempt to update the shortest-path count for the same destination node simultaneously. That is where contention shows up, and that is what makes the project interesting.

3 The Challenge

The main challenge is that graph workloads are irregular in ways that GPUs do not always like. Different vertices have different degrees, which means some edges are associated with much more work or much more contention than others. Memory accesses are also less regular than in dense numerical code, so performance can become limited by data movement and locality instead of pure arithmetic throughput.

The specific challenge we want to study is write conflict. In an edge-parallel kernel computing BC, many threads need to update the same vertex state. The simplest solution is to protect those updates with atomic operations. We will build this naive version first to establish a baseline for GPU bottlenecks. A more careful strategy is to become conflict-aware: instead of letting all edges run together, we organize or schedule them so that conflicting updates are reduced or separated. We

will explain clearly why the improved version performs better and where it still struggles compared to our CPU baselines.

4 Resources

Hardware. We plan to use the GHC machines with NVIDIA RTX 2080 GPUs for the CUDA implementations. We will also use the same environment for our CPU baselines so that the comparison is as fair as possible.

Dataset. We will use public graph datasets from the Stanford Network Analysis Project (SNAP). This provides standard, real-world sparse graphs.

Software. The main implementation will be in C++ and CUDA. The multicore CPU version will use OpenMP.

Code Base. We may reuse parts of earlier course infrastructure such as Assignment 2 and 3 for build convenience on the CUDA and OpenMP side, but the graph kernels, benchmarking code, and evaluation pipeline will be implemented by us.

5 Goals and Deliverables

Plan to Achieve

Our main goal is to build and evaluate three primary implementations of Betweenness Centrality:

- a serial CPU baseline,
- a multicore CPU implementation, and
- an optimized CUDA implementation.

We will also build a naive CUDA implementation that resolves shared updates with atomics. This will not be our primary benchmark target, but will serve as a stepping stone to profile bottlenecks and motivate our optimized GPU design. We will discuess design decisions backed by benchmarking to show how we got to our final optimized CUDA implementation.

The primary output for all versions will be an array of centrality scores for all vertices, operating on the exact same CSR graph representations. We will also include a parameter to extract the top K highest-betweenness nodes.

Hope to Achieve

If the main implementation goes well, we would like to compare different conflict-aware strategies rather than just one. We also hope to test on dense or uniform graphs to see whether the same trends hold across completely different graph structures.

Evaluation and Deliverables

Our final deliverables will include:

- a serial and multicore CPU baseline for BC
- a documented optimization path from a naive CUDA version to an improved conflict-aware CUDA version

- runtime and speedup graphs comparing the serial, multicore, and optimized GPU implementations
- a discussion of bottlenecks such as contention, synchronization overhead, memory behavior, and workload imbalance
- a short analysis of how graph structure affects the results.

6 Platform Choice

CUDA is a good fit here because the workload exposes a lot of potential parallelism across graph edges. If we assign one edge to one thread, the GPU can process a large number of edges at once. At the same time, this is not an easy GPU problem because shared vertex updates create conflicts, and sparse graph structure tends to lead to irregular memory access and uneven work distribution.

That is exactly why this is a worthwhile project. We are not choosing CUDA just because it is fast hardware. We are choosing it because it forces us to deal with the real problems that show up in irregular parallel workloads.

The CPU baselines give us a grounded comparison and help us understand whether the GPU is winning because of real throughput advantages or losing because contention and irregularity dominate the computation.

7 Benchmarking Plan

We plan to benchmark the project in a way that makes the comparison fair and easy to explain.

First, we will compare the three final implementations directly:

- serial CPU,
- multicore CPU, and
- optimized conflict-aware GPU.

For each version, we will measure wall-clock runtime and speedup against the serial baseline. Second, we will vary the input workload using different SNAP datasets to test varying graph sizes and structures.

Third, we will profile the GPU versions. We will use Nsight tools to identify the exact cost of atomics and memory stalls in the naive version, and use that data to demonstrate the effectiveness of the improved version.

8 Schedule

Mar 30–Apr 5: Baselines and Naive GPU

- **Isaiah:** Set up the CUDA project skeleton and implement the Naive GPU BC kernel using atomics.
- **Yi-Ning:** Implement the Serial BC baseline and the Multicore CPU (OpenMP) BC implementation.
- **Shared deliverable:** Serial CPU, Multicore CPU, and Naive GPU execute successfully on a small test graph and return matching arrays of centrality scores.

Apr 6–Apr 12: Profiling and Milestone Prep

- **Isaiah:** Profile the Naive GPU version using Nsight to document bottlenecks. Begin progressive implemenation towards optimized GPU version.
- **Yi-Ning:** Build the automated benchmarking harness. Run initial scaling tests comparing the CPU implementations against the Naive GPU to gather baseline constraint data.
- **Shared deliverable:** Working prototype of the Conflict-Aware GPU kernel and raw performance data for the milestone.

Apr 13–Apr 19: Milestone and Optimization

- **Isaiah:** Debug and optimize GPU kernel (memory coalescing, warp divergence). Add the top K parameter.
- **Yi-Ning:** Run the full suite of comparative benchmarks (Serial vs. Multicore vs. Optimized GPU) across massive SNAP datasets.
- **Shared deliverable:** Milestone submitted. Full final benchmarking data collected.

Apr 20–Apr 26: Deep Analysis and Edge Cases

- **Isaiah:** Conduct ablation study mapping the specific performance gains from the naive to the optimized GPU version.
- **Yi-Ning:** Test implementations on dense/uniform graphs. Generate final runtime and speedup charts.
- **Shared deliverable:** Code finalized. Drafts of visual charts and bottleneck analysis completed.

Apr 27–May 1: Final Report and Poster

- **Shared:** Split writing for the final report. Design and print the poster.
- **Shared deliverable:** Final PDF uploaded to Gradescope, code to Autolab, poster prepared for Friday's session.