15-618 Final Project Report

Parallelizing Betweenness Centrality on CPU and GPU

Yi-Ning Huang Isaiah Velez

April 30, 2026

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

1 Summary

We implemented Brandes' Betweenness Centrality on two parallel platforms, a multi-core CPU using OpenMP and an NVIDIA RTX 2080 using CUDA. The CPU implementation parallelizes the outer source loop and reaches up to 7.36× speedup at 8 threads through thread-local accumulation and reachable-vertex optimizations, and supports a sampled mode that handles large scale graphs (`web-Google` at 916,428 vertices) at predictable runtime. The GPU side worked through a profiling-driven sequence of three within-source optimizations that each addressed one bottleneck but worsened another, before pivoting to a multi-source design that runs many BFS computations side by side and dominates every within-source attempt. At 32 sources the multi-source GPU design is the fastest implementation on every graph we measured. At full source count the GPU wins on the larger and denser real graphs while OpenMP wins on the smaller graphs whose working set fits in L3 cache.

2 Background

Betweenness Centrality

For our final project we explored Betweenness Centrality (BC) of various graphs. BC deals with the importance of a node amongst a graph by measuring the ratio of the shortest paths that utilize that node. For a graph $G = (V, E)$:

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Where $C_B(v)$ is the BC score of v, σ_{st} is the total number of shortest paths from s to t, and $\sigma_{st}(v)$ is the count of all shortest paths from s to t where it passed through v. The BC formula is the summation of all the ratios of the shortest paths. When more shortest paths from s to t must pass through v, its shortest path ratio will be closer to 1, otherwise it will be closer to 0.

Its importance stems from network analysis that quantifies the influence or control a node has in a network. A node with a higher BC has more information flowing through it.[1] In domains such as social networks, roadways, and resources, nodes with high BC become key points of attention. Historically, computing BC naively required running all-pairs shortest paths, an $O(V^3)$ operation using algorithms like Floyd-Warshall, followed by counting how many of those paths pass through each vertex. Brandes' algorithm brought this down to $O(VE)$ by avoiding the explicit all-pairs computation entirely.

Brandes' Algorithm

[2] Brandes' algorithm takes an unweighted graph $G = (V, E)$ as its input and produces an array of BC scores $C_B(v)$ for every vertex $v \in V$ as its output. It measures BC by iterating over every vertex as a source and

computing each node's contribution to the global BC score relative to that source. The algorithm runs in $O(VE)$ time and $O(V+E)$ space, and the cost is dominated by the outer source loop. Each of the $|V|$ source iterations runs an $O(V+E)$ BFS plus dependency walk, and the source loop is the expensive part of the algorithm and the natural target for parallelism. For each source vertex s, it proceeds in two steps.

Single-Source Shortest Paths. A BFS is run from s, tracking three things for every vertex v: the shortest-path distance from s, the number of shortest paths $\sigma[v]$ from s to v, and the predecessor list $P[v]$ of vertices that lie on a shortest path to v. Each settled vertex is pushed onto a stack S in the order it is reached, so that deeper vertices furthest from s appear at the top of the stack.

Dependency Accumulation. Vertices are popped from the stack S in reverse BFS order (deepest and furthest from s first). For each predecessor u of the current vertex w, the pair dependency is propagated using Brandes' recursive formula

$$\delta[u] \mathrel{+}= \frac{\sigma[u]}{\sigma[w]} \cdot (1 + \delta[w])$$

Once all predecessors of w have been updated, $\delta[w]$ is added to the global BC score of w (provided $w \neq s$). Iterating this over all source vertices yields the exact BC scores for every node in the graph. The complete pseudocode as presented by Brandes is reproduced in Appendix A.

Parallelism The outer loop over source vertices is fully data-parallel because the iterations have no shared dependencies. Each source s produces its own BC contributions, and we can merge them into a single global array at the end. This gives us $|V|$ independent units of work that scale with thread count, which is what we initially exploited by splitting iterations across cores or threads and reducing the disjoint BC scores into one global array.

Within a single source iteration, things are more constrained. The forward BFS must run in level order because each depth depends on the previous one being fully expanded, and the dependency accumulation walks the stack in reverse, so it cannot start until the BFS finishes. These ordering constraints prevent simple parallel decomposition at the per-source level, but each individual BFS level is parallel within itself because every vertex at the current depth can be expanded at the same time. Per-iteration parallelism proved to be much more challenging to optimize than the outer loop.

Locality is uneven. CSR stores each vertex's neighbor list contiguously, which gives good access patterns when expanding any single vertex. However, the BFS jumps between different rows of the neighbor array as it pulls vertices off the queue, which weakens spatial locality across the traversal. Random access patterns dominate as graphs grow larger and more irregular.

SIMD execution is a poor fit for the inner work. Neighbor list lengths vary widely across vertices, so vector lanes end up either idle or processing unrelated data. Branching on visited and distance checks adds further divergence. This is why we focused on thread-level parallelism with OpenMP and CUDA rather than relying on short-vector SIMD.

Graph Representation

Both CPU and GPU implementations use Compressed Sparse Row (CSR) format. Each input edge is treated as undirected and stored as two directed CSR entries (one for each direction). This doubles the edge count compared to the raw input file but keeps traversal consistent across all datasets, including those that the SNAP collection lists as directed.

SNAP Graphs

Stanford provides publicly available graphs via their *Stanford Network Analysis Platform* (SNAP). [3] We evaluate on six graphs of varying structure and one additional large scale graph used only for the sampled approximation experiments. CSR edge counts below are after-loading entries (each undirected input edge contributes two directed entries).

Name	Type	Vertices	CSR Edges	Description
`line_10000`	Synthetic chain	10,000	19,998	Regular baseline; best-case scaling
`ca-GrQc`	Collaboration	5,242	28,980	Arxiv General Relativity
`p2p-Gnutella04`	P2P	10,876	79,988	Gnutella network from 8-4-2002
`wiki-Vote`	Wikipedia votes	7,115	207,378	who-votes-on-whom network
`ego-facebook`	Social	4,039	176,468	Social circles (anonymized)
`soc-Slashdot0811`	Social	77,360	905,468	Slashdot social network
`web-Google`	Web	916,428	8,644,102	Large scale, sampled BC only

Table 1: Datasets used for evaluation.

The first six graphs are used for exact BC measurements. The web-Google graph is too large for all-source Brandes within a reasonable time budget, so we use it only to evaluate the sampled approximation described in Section 3.1.

The figure below shows the top 100 highest BC nodes for each dataset we used. The node size and color are proportional to the BC score that they have. The first six images use the CPU implementation BC scores, and the `web-Google` graph uses the sampled approximation.

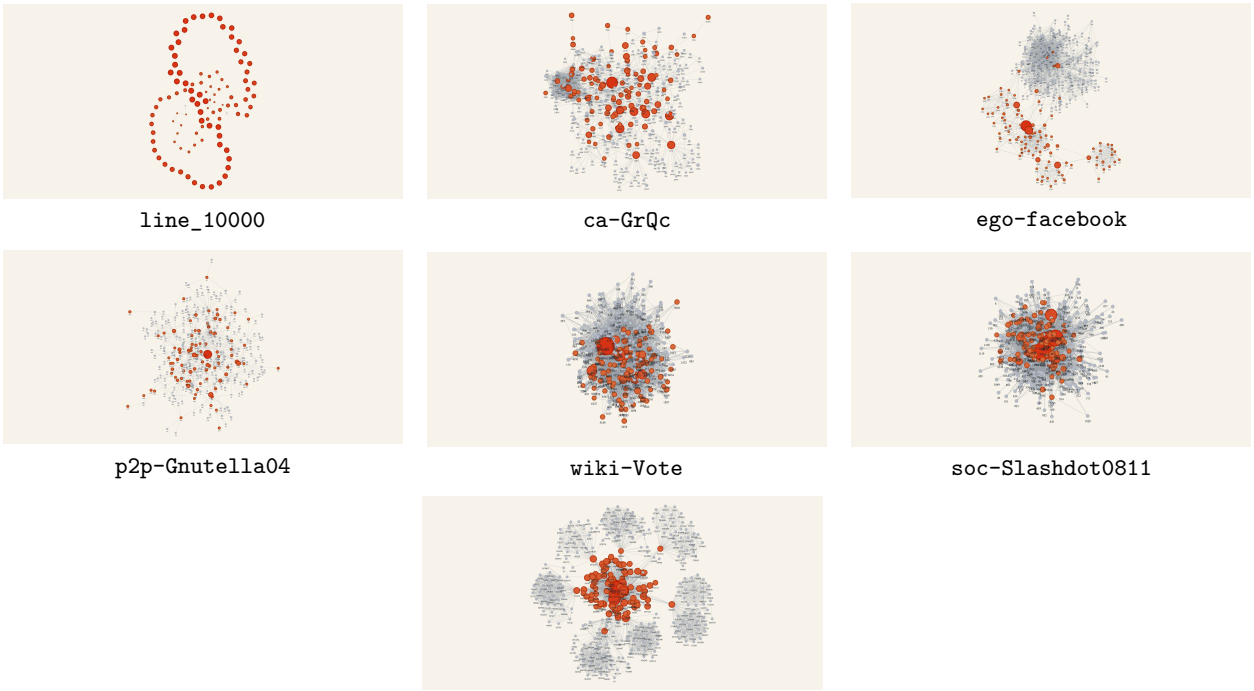

Figure 1: Top-100 BC vertices for each dataset.

3 Approach

There are two main parallel tracks: CPU with OpenMP and GPU with CUDA.

3.1 CPU Track

The CPU track was benchmarked on the GHC 28 machine using OpenMP. Throughout the CPU results below, we use up to 8 OpenMP threads and report the 8-thread configuration as the main strong-scaling endpoint.

Initial implementation: serial Brandes. The serial implementation is a direct C++ realization of Brandes' algorithm and is the correctness oracle that every other implementation in this project is checked against. For each source vertex it runs the forward BFS using a standard queue, computes the distance and shortest-path counts, then walks the visited vertices in reverse depth order to accumulate dependencies and finally to update the global BC scores. The implementation does no parallel work, but its correctness is what makes the rest of the project possible.

First OpenMP attempt (Shared thread-array reduction). The first OpenMP version did the obvious thing: parallelize the outer loop over source vertices. Each thread processed a subset of sources independently and stored its BC contributions in a per-thread array. In principle this should have scaled well, since different sources do not depend on each other. In practice, it did not. The bottleneck was the **reduction step** at the end. After all threads finished, the program still had to merge every thread's BC array back into the global result. That merge scanned across all threads for every vertex, creating a large shared reduction pass. Because the array was laid out thread-by-vertex, this merge effectively walked it column-wise, jumping across large per-thread rows with poor cache locality. On large graphs, this merge was expensive enough to erase almost all of the benefit from parallelizing the source loop.

Thread-local accumulation redesign (Per-thread final merge). We then simplified the reduction path. Instead of keeping a shared collection of thread-indexed BC arrays and performing one global vertex-by-thread reduction, each thread now accumulates into a single local BC array while it works through its assigned sources, and merges that local array into the global BC result only once at the end. This keeps synchronization out of the hot path and replaces the awkward shared reduction structure with a simpler per-thread final merge. The effect is visible in the slashdot comparison below:

```
Old reduction:
for v in vertices:
    for t in threads:
        bc[v] += thread_bc_arrays[t][v]
```

```
Redesigned reduction:
for each thread t:
    for v in vertices:
        bc[v] += local_bc_t[v]
```

Figure 2: Old vs. redesigned reduction patterns for the OpenMP BC.

Implementation path	1T time (s)	4T time (s)	8T time (s)	8T speedup
Shared thread-array reduction	564.84	564.84	564.84	1.00×
Per-thread final merge	580.16	161.78	108.41	5.35×

Table 2: Effect of the reduction redesign on `soc-Slashdot0811` at full source count.

The original reduction path showed essentially no scaling, while the redesigned version recovered a 5.35× speedup at 8 threads on the same graph.

Reachable-vertex accumulation. A second refinement targeted the per-source BC accumulation. The earlier code added the dependency value of every vertex into the per-source BC array regardless of whether the vertex was reachable from the source. On disconnected or partially connected graphs, this produced an extra full-vertex traversal per source, contributing an unnecessary $O(V)$ term per source on top of the productive BFS work. We changed the implementation to update BC only for vertices that the BFS actually reached, which the recorded visit order already gave us for free. On the larger graphs this measurably reduced both runtime and memory traffic.

Sampled BC for large-scale graphs. For large-scale inputs like web-Google (916,428 vertices), exact all-source Brandes is not practical: at the implementation's measured 70 ms per source on 8 threads, the full computation would take about 18 hours. To address this, we added a sampled mode. Given a desired sample size k, the implementation selects k source vertices using a deterministic random seed, runs the same Brandes computation only from those sources, and scales the accumulated scores by $|V|/k$. The result is an unbiased estimator of the exact BC scores rather than the exact values themselves, but the high-BC ranking that

most downstream applications care about is preserved at modest sample sizes. The runtime and accuracy characteristics of this mode are reported in Section 4.3.

Scheduling check. We also ran a brief OpenMP scheduling check on `soc-Slashdot0811` (8 threads, 3 runs): `static` averaged 133,770 ms, `guided,4` 134,003 ms, `dynamic,4` 134,824 ms, and `dynamic,1` 139,768 ms. In this workload, finer-grain dynamic scheduling increased overhead without improving throughput, so we use `static` as the default CPU schedule.

3.2 GPU Track

The GPU track was developed on GHC 28 with an NVIDIA RTX 2080. We wrote all kernels from scratch in CUDA C++ and used Nsight Compute and Nsight Systems for profiling. The Nsight profiling script was reused from Assignment 2, and the project's main entry-point file was based on Assignment 2's structure but heavily modified for this work.

Problem Mapping A *source* is one vertex chosen as the starting point of a BFS traversal. Brandes' algorithm runs the same BFS plus dependency computation $|V|$ times, once per source, and adds each per-source contribution into a global BC array. This gives parallelism in two places. The **outer loop** runs across sources, where the $|V|$ independent computations share no state. The **inner BFS level** runs within one source, where every vertex at the current depth can be explored at the same time. Earlier versions of our code used only the inner level, which left most of the GPU's 46 streaming multiprocessors idle whenever the set of active vertices was small.

Our final implementation uses both at once with a 2D grid (Figure 3). Each thread is a (vertex, source) pair. The per-source state for distance, σ, and δ is stored as one big array of size $S \times V$, organized so that all of source 0's vertex slots come first, then all of source 1's, and so on. We call this layout source-major because the source index is the slowest-varying coordinate. The point of this layout is memory coalescing. Consecutive threads in a warp share a source and differ only in their vertex index, so they read consecutive memory locations and the GPU can fetch them in one transaction.

The CSR graph (the row offsets and the neighbor list) is read-only and shared across all sources. We also keep a second distance buffer for the BFS updates. The reason is that many parents at the same depth often try to claim the same child. Imagine two vertices A and B both at depth 5 that share a neighbor C. Both want to mark C as depth 6 and add their path count to it. If they wrote into the same array, one parent's write could overwrite the other's read, and we would lose path counts. Writing the new depth into a separate buffer and copying it back at the end of the level lets every parent contribute correctly. The global BC array is the only structure all sources write to, and we accumulate into it once at the end of each chunk through atomic adds. Running all sources at once costs about $S \times V \times 24$ bytes, which on `soc-Slashdot0811` would exceed 142 GB, so we process sources in chunks of 256. Each chunk reuses the same state arrays, and the global BC array persists across chunks.

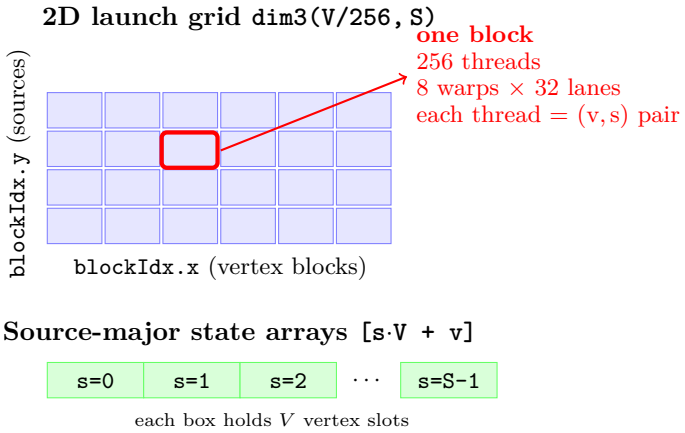

Figure 3: Multi-source launch geometry and data layout. The 2D grid couples vertex blocks (x) with source slots (y), so each thread is a unique (vertex, source) pair. Per-source state is laid out source-major so consecutive threads within a warp read consecutive vertex slots.

Optimization Iterations The optimization journey was driven by Nsight profiling. We started with a naive edge-parallel baseline that assigned one thread per directed edge. Every edge thread checked whether its source vertex was at the current BFS depth and, if so, tried to mark the destination as one level deeper if it had not been seen yet and add to its path count. We then profiled this baseline with Nsight Compute across the dataset suite, and three patterns showed up consistently.

Three Bottlenecks Identified by Nsight

(1) Hub vertex contention. Memory stalls dominated the warp-cycle breakdown of the BFS kernel. When many edges point to the same high-degree hub vertex (a celebrity in a social graph or a popular page in a web graph), every thread processing one of those edges has to atomically update the same memory location. The GPU serializes those updates internally, and the rest of each affected warp sits idle waiting its turn.

(2) Idle threads. The GPU's instruction scheduler was eligible to issue a warp on only ∼3% of cycles, meaning the SMs sat unable to do work for nearly the entire run. We launched a huge number of threads, but at any given BFS level, most of them belonged to edges whose source vertex was not at the current depth. Those threads did a quick read-and-discard and exited, occupying scheduler slots they could not productively use.

(3) Host-device sync overhead. After every BFS level the host had to read back a "continue or stop" flag from the device, which forced one device synchronization plus one PCIe transfer per level. On `line_10000` (diameter 9,999), 32 sources times 9,999 levels was 319,968 such transfers, each pausing the GPU for tens of microseconds.

Three Within-Source Attempts

Attempt 1. Vertex-parallel BFS (targets bottleneck 2). We switched from one thread per edge to one thread per vertex. Each thread now checks whether its assigned vertex is at the current depth, and if so walks its own neighbor list inside the kernel. The kernel only does productive work for threads that actually own an active vertex. The speedup was good on the sparse collaboration graph (12.9× faster than the baseline on `ca-GrQc`) because almost every level has only a few active vertices. The regression was bad on dense social graphs where assigning one thread to a hub vertex meant a single thread sequentially walked thousands of edges. On the densest graph we measured a 10.2× slowdown.

Attempt 2. Batched level execution (targets bottleneck 3). Instead of waiting after every BFS level, the host launches 32 levels in a row in the same CUDA stream and then waits once at the end of the batch. CUDA executes operations on the same stream in the order they were issued, so correctness is preserved. The same trick is applied to the entire backward dependency pass, which always traverses a known depth range and

so does not need a termination check at all. This recovered a large fraction of the lost time on the chain graph (4.3× over the previous attempt) but had no effect on bottlenecks 1 or 2 and gave only modest gains on social graphs.

Attempt 3. Active-vertex queue (targets bottleneck 2 more aggressively). We maintained an explicit list of currently-active vertices each level and launched exactly enough threads to cover that list, eliminating the wasted threads from the vertex-parallel design. The result was mixed. The queue requires the host to read the list size every level (otherwise it does not know how many threads to launch), which re-introduced the same synchronization pattern attempt 2 had eliminated. On graphs with few active vertices in their early or late BFS levels, the kernel launched only a handful of thread blocks and Nsight measured 12.8% achieved occupancy.

Lesson learned. After three within-source designs, no single one was best on every graph. Each addressed one of the three bottlenecks but worsened or revealed another. The fundamental constraint is that a single source's BFS at a given depth only has so much parallelism in it. No amount of clever scheduling could create more parallel work where the algorithm did not have it.

Multi-Source Optimization

We returned to the axis we had named in the background but never exploited, the outer loop over sources. Until this point we had been running each source one at a time and trying to make each one faster. The final design runs many sources at the same time on the GPU, with each source given its own private copies of the per-source state arrays so that no two sources can interfere with each other. The launching geometry becomes two-dimensional, with one dimension over vertices and one dimension over source slots, and the BFS forward pass keeps the batched-level pattern from attempt 2 so we do not regress on deep graphs.

On every graph we measured at 32 sources, the multi-source design is the fastest implementation. Where the within-source designs each won on some graphs and lost on others, the multi-source design dominates uniformly (Figure 4). On the chain graph it is 14.8× faster than the best within-source design, and on the densest social graph it is 18× faster.

Graph	Edge-par.	Vertex-par.	Batched lvl.	Active queue	Multi-source
`line_10000`	8,623.4	8,847.8	2,041.1	6,809.1	**137.7**
`ca-GrQc`	68.4	5.3	12.1	5.9	**3.49**
`p2p-Gnutella04`	9.1	18.2	19.0	16.6	**2.82**
`wiki-Vote`	7.7	59.2	54.6	57.4	**6.74**
`ego-facebook`	9.3	94.9	79.1	104.6	**4.43**

Table 3: BC compute time (ms) at 32 sources across the five GPU implementations. Each within-source design wins on some graphs and loses on others; the multi-source design dominates every graph.

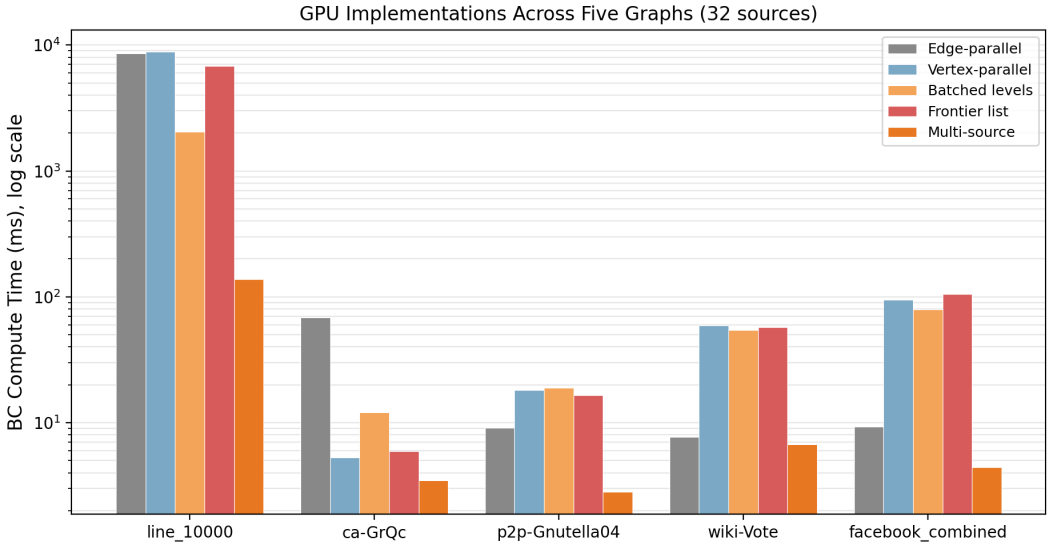

Figure 4: GPU implementation times across five graphs at 32 sources, log scale. The multi-source design (rightmost in each cluster) is the fastest implementation on every graph. The within-source designs each win on some graphs and lose on others.

Multi-Source Insight Nsight Compute confirms the reason for the big speedup. The numbers below compare the BFS kernel of the previous best implementation (batched levels) against the BFS kernel of the multi-source design.

Metric	Previous best	Multi-source	Direction
Achieved occupancy (how full the SMs are)	50%	**72–82%**	up 22–32 points
Memory throughput in BFS kernel	36–38 GB/s	**92–108 GB/s**	roughly 3× higher
Scheduler-eligible cycles	∼3%	**7–9%**	up 4–6 points

Table 4: Nsight Compute metrics for the BFS kernel before and after the multi-source redesign.

Serial Change Brandes' algorithm uses two data structures that we changed to enable better mapping to the GPU. The first is the predecessor list $P[w]$. Brandes' algorithm explicitly maintains a list for each node during the BFS, recording which vertices lie on a shortest path leading to it. Maintaining a variable-length per-vertex list across thousands of GPU threads is memory-heavy, so our backward pass instead re-derives the predecessor relationship by checking whether u is exactly one level shallower than w before accumulating the dependency.

The second is the vertex-order stack S. Brandes' algorithm pops vertices from this stack in non-increasing-distance order and processes them one at a time during the backward pass. We replaced this with an explicit level-by-level walk where the host launches one kernel per depth, and the kernel processes all (vertex, source) threads at that depth in parallel. This swaps a serial pop loop for a parallel level walk and lets every dependency update at the current depth happen at the same time across all sources in the chunk.

4 Results

Setup. We measure performance as wall-clock BC compute time in milliseconds, averaged over 3 timed runs after one warmup run, and exclude graph load and CSR build time. Both tracks were benchmarked on GHC 28. The CPU OpenMP strong-scaling and cross-platform full-source comparisons all use the same six SNAP graphs introduced in the Background. The web-Google graph is too large for exact all-source Brandes within

a reasonable time budget, so it appears only in the sampled BC experiments in Section 4.3. Speedups are reported relative to the serial CPU implementation.

4.1 CPU Serial Baseline and OpenMP Strong Scaling

The OpenMP implementation scales near-linearly on regular and sparse graphs and plateaus at 5–5.5× on the larger or more skewed ones. The chain graph and the small collaboration graph reach over 7× at 8 threads, while the dense social graphs (`wiki-Vote`, `soc-Slashdot0811`) cap out closer to 5.3×. `ego-facebook` is dense but small enough to fit comfortably in L3 cache, so it scales as well as the sparse graphs.

Graph	Serial (ms)	1T (ms)	4T (ms)	8T (ms)	Speedup at 8T
`line_10000`	3,090.7	3,139.7	815.3	426.3	**7.36**×
`ca-GrQc`	2,569.3	2,644.3	691.3	366.3	**7.22**×
`ego-facebook`	1,654.0	1,507.0	417.0	221.0	**6.82**×
`p2p-Gnutella04`	8,971.0	9,091.7	2,447.3	1,520.0	5.98×
`wiki-Vote`	6,877.7	6,562.0	1,805.7	1,233.3	5.32×
`soc-Slashdot0811`	570,448.0	580,160.0	161,781.0	108,406.7	5.35×

Table 5: OpenMP strong scaling at full source count on the GHC 28 machine, 8 physical cores. Serial times serve as the speedup denominator.

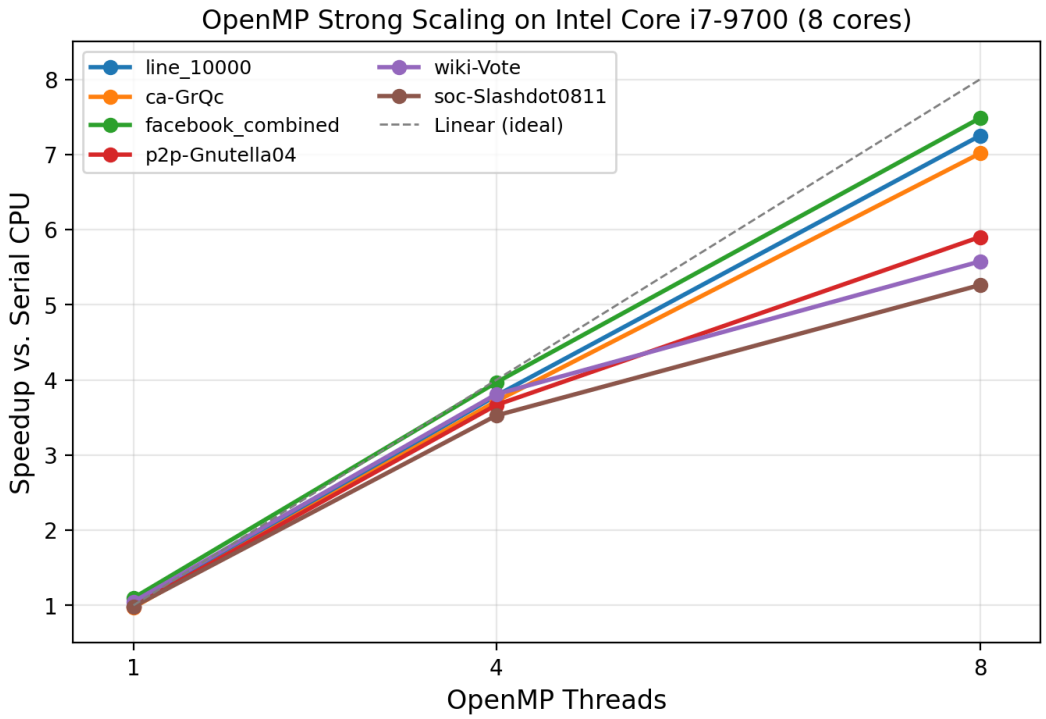

Figure 5: OpenMP strong scaling, 1 to 8 threads, normalized to serial.

4.2 Cross-Platform Comparison at Full Source Count

At full source count the two parallel tracks tell different stories on different graphs. The GPU multi-source design wins on graphs with enough work per BFS to amortize its per-chunk setup cost, while OpenMP wins on graphs small enough to fit in L3 cache. The crossover lies roughly at the size of `p2p-Gnutella04`.

Graph	Serial CPU	OpenMP-8T	GPU multi-source	Winner
`line_10000`	3,090.7	426.3	37,810.5	OpenMP*
`ca-GrQc`	2,569.3	366.3	1,824.7	OpenMP
`ego-facebook`	1,654.0	221.0	242.5	OpenMP (very close)
`p2p-Gnutella04`	8,971.0	1,520.0	835.2	GPU
`wiki-Vote`	6,877.7	1,233.3	1,163.2	GPU (close)
`soc-Slashdot0811`	570,448.0	108,406.7	68,491.6	**GPU**

Table 6: BC compute time in ms at full source count. *`line_10000` is a special case, because the chain graph has only one active vertex per BFS level, so the GPU has nothing to parallelize within a source, and even with 256 sources running together the per-chunk setup overhead dominates.

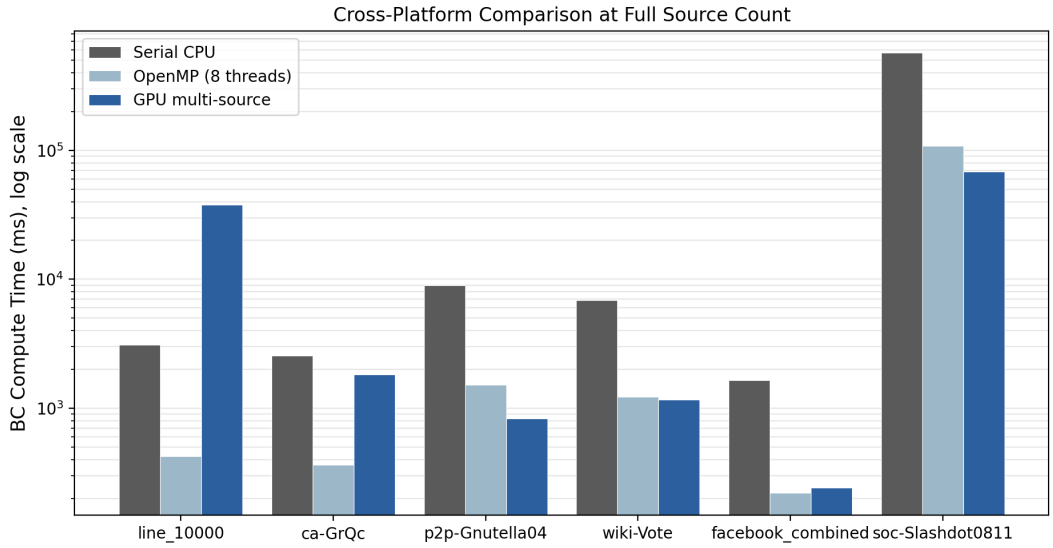

Figure 6: Serial CPU vs. 8-thread OpenMP vs. GPU multi-source at full source count, log scale. The GPU wins on the larger and denser real graphs, while the CPU wins on the smaller graphs whose working set fits in L3.

The biggest GPU win is `soc-Slashdot0811`, where the GPU finishes in 68 seconds versus 108 seconds at 8 threads on the CPU and nearly 10 minutes on the serial baseline. This is the largest graph in our exact-BC suite, and it is exactly the regime where the GPU's parallel work overwhelms its setup cost. On the other end, `ca-GrQc` is small enough that the CPU is 5× faster than the GPU, because the GPU spends most of its time launching kernels and synchronizing source chunks rather than doing useful BFS work.

4.3 Sampled BC at Large Scale

For graphs that are too large for exact all-source Brandes, the CPU implementation supports a sampled mode (described in Section 3.1). We evaluate it on `web-Google` (916,428 vertices) where exact BC is not practical, and use `ca-GrQc` as an accuracy proxy because exact BC is feasible there.

Runtime scaling. Runtime is linear in the number of sampled sources, with per-source cost essentially flat between 67 and 72 ms (Figure 7). At 1,000 samples the BC compute time is about 70 seconds. By comparison, exact all-source BC on `web-Google` would require 916,428 traversals at the same per-source cost, roughly 18 hours, which is outside the practical benchmark budget.

Sampled sources	BC compute (ms)	ms / source
100	6,730	67.3
500	35,481	71.0
1,000	70,835	70.8
5,000	361,282	72.3

Table 7: Sampled BC runtime on `web-Google` at 8 OpenMP threads.

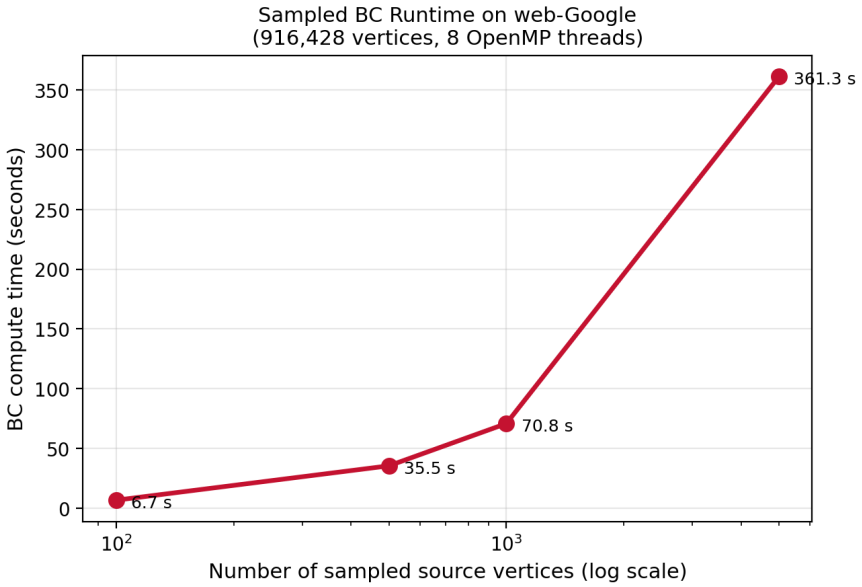

Figure 7: Sampled BC compute time on `web-Google` at 8 threads, log-x. Linear scaling with sample count.

Accuracy. We measure top-k overlap on `ca-GrQc` between the sampled BC ranking and the exact 8-thread OpenMP ranking. With 1,000 samples, the top-50 set matches the exact ranking at 82% overlap (Table 8, Figure 8). The general pattern is what one would expect, where more samples give better agreement and the high-BC vertices stabilize first. The 100-sample row is essentially noise, but at 1,000 samples the high-centrality structure is recovered well at a fraction of the exact source count.

Sampled sources	Top-10	Top-50	Top-100
100	30.0%	30.0%	50.0%
500	60.0%	62.0%	62.0%
1,000	70.0%	**82.0%**	76.0%

Table 8: Top-k overlap between sampled and exact BC rankings on `ca-GrQc`. Higher is better, 100% means identical to exact.

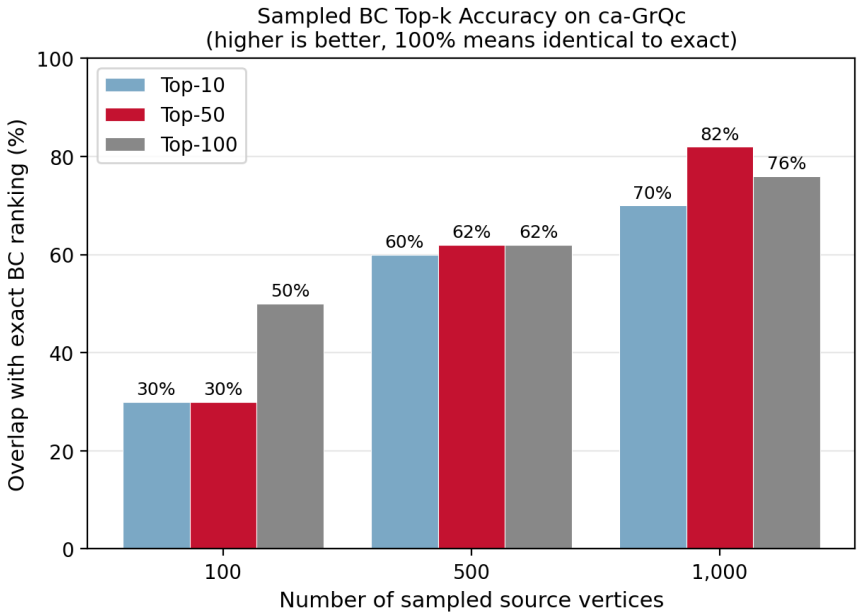

Figure 8: Top-k overlap as the sample size increases. With 1,000 samples on `ca-GrQc`, the top-50 matches the exact ranking at 82% overlap.

`web-Google` at 1,000 samples should give a similarly stable ranking of its highest-centrality vertices, at a cost of about 70 seconds rather than 18 hours.

4.4 What Limits Each Platform

The exact and sampled experiments together show where each platform's performance comes from and where it stops. We do not have phase-level timings for BFS versus dependency versus accumulation, only end-to-end BC compute time, so the per-component breakdown below relies on the Nsight Compute readings already reported in Table 4 for the GPU and on the OpenMP scaling pattern for the CPU.

CPU limits at 8 threads. Three factors cap the OpenMP speedup on the larger graphs:

- **Degree skew and load imbalance.** High-degree hub vertices dominate BFS work for some sources, so threads with hub-heavy assignments finish much later than threads with easy assignments. This is most visible on `wiki-Vote` and `soc-Slashdot0811`.
- **DRAM bandwidth.** BFS and the dependency pass are pointer-chasing workloads that do little arithmetic per edge. Once cache misses start landing in DRAM, the implementation is memory-bandwidth-bound rather than compute-bound, and adding more cores does not help.
- **L3 cache.** Larger graphs are too big for the L3 cache. `ego-facebook` is the exception that confirms the diagnosis, since it is dense but small enough that its working set fits comfortably in L3, and so it scales as well as the sparse graphs.

GPU limits. On graphs where the GPU wins, the constraint is the same hub-vertex memory contention that appeared in the original profiling (Bottleneck 1 from Section 3.1). The multi-source design fixed the SM-utilization and host-sync bottlenecks, but the per-vertex `atomicAdd` on path counts and dependencies is unchanged, and on dense social graphs warps still spend many cycles waiting on global memory at high-degree vertices.

Right Target Machine Depending on the graph, CPU or GPU is the better option. For social and web-scale graphs, the GPU was the right choice. The 1.58× improvement over 8-thread OpenMP on `soc-Slashdot0811`, on top of the 5.35× OpenMP speedup over serial, amounts to an 8.3× end-to-end improvement over the

serial baseline on a representative real-world graph. For the small graphs, OpenMP is faster and the GPU's overheads were too much. It's best to pick the platform by graph size at runtime, with small graphs going to OpenMP and large graphs going to the GPU.

5 References

References

[1] Memgraph, "Betweenness centrality and other centrality measures in network analysis." https://memgraph.com/blog/betweenness-centrality-and-other-centrality-measures-network-analysis.

[2] U. Brandes, "A faster algorithm for betweenness centrality," *The Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[3] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.

[4] Grammarly Inc., "Grammarly." https://www.grammarly.com, 2026.

6 Work Distribution

50% / 50% split.

Yi-Ning Huang:

- Implemented multi-core CPU code (serial baseline and OpenMP parallel version).
- Implemented sampled-source BC for large scale graphs and ran the runtime and accuracy experiments.
- Built the shared CSR graph loader, benchmark harness, and dataset pipeline.
- Ran all CPU benchmarks and produced the OpenMP scaling and dataset visualization figures.

Isaiah Velez:

- Implemented GPU code using CUDA, including the iteratively profiled optimization sequence.
- Conducted Nsight Compute and Nsight Systems profiling to identify the three primary bottlenecks and validate each optimization step.
- Ran all GPU benchmarks across the six datasets

A Brandes Algorithm Pseudocode

The following is the exact pseudocode for betweenness centrality in unweighted graphs as given in [2].

Algorithm 1 Betweenness centrality in unweighted graphs [2]

$C_B[v] \leftarrow 0,\ v \in V$
for $s \in V$ **do**
    $S \leftarrow$ empty stack
    $P[w] \leftarrow$ empty list, $w \in V$
    $\sigma[t] \leftarrow 0,\ t \in V;\ \sigma[s] \leftarrow 1$
    $d[t] \leftarrow -1,\ t \in V;\ d[s] \leftarrow 0$
    $Q \leftarrow$ empty queue
    enqueue $s \rightarrow Q$
    while Q not empty **do**
        dequeue $v \leftarrow Q$
        push $v \rightarrow S$
        for all neighbor w of v **do**
            if $d[w] < 0$ **then** ▷ w found for the first time?
                enqueue $w \rightarrow Q$
                $d[w] \leftarrow d[v] + 1$
            end if
            if $d[w] = d[v] + 1$ **then** ▷ shortest path to w via v?
                $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
                append $v \rightarrow P[w]$
            end if
        end for
    end while
    $\delta[v] \leftarrow 0,\ v \in V$
    while S not empty **do** ▷ S returns vertices in non-increasing distance from s
        pop $w \leftarrow S$
        for $v \in P[w]$ **do**
            $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$
        end for
        if $w \neq s$ **then**
            $C_B[w] \leftarrow C_B[w] + \delta[w]$
        end if
    end while
end for