**Final Project Report**

## Summary:

We implemented parallel graph coloring using OpenMP on the CPU and CUDA on the GPU, and compared the performance of the two approaches on the GHC machines (8-core Intel CPU + NVIDIA RTX 2080 GPU).

Our best CPU implementation uses a multi-phase speculative coloring algorithm with hub preprocessing, achieving up to 6.7x compute-time speedup on 8 threads with near-zero conflict rates (<0.02%) and color quality matching the sequential baseline.

We additionally implemented a CUDA GPU version with CPU-side hub preprocessing and a 128-bit register bitmask for per-thread color tracking. On large graphs, the GPU achieved strong kernel-only speedups, though total-time improvements were smaller because transfer and preprocessing overheads remained significant.

Over the course of the project, we applied numerous targeted optimizations across initialization, compute, and synchronization phases, systematically addressing each bottleneck identified through profiling and Amdahl's law analysis. All implementations were benchmarked across 11 graphs (synthetic and real-world, 10K to 1M vertices) on the GHC cluster.

## Background:

### *Problem Definition*

Graph coloring assigns an integer label (color) to every vertex of an undirected graph such that no two adjacent vertices share the same color. The objective is to minimize the number of colors used. Determining the minimum number of colors (the chromatic number) is NP-hard, so practical algorithms use a greedy heuristic that guarantees at most max_degree + 1 colors.

Graph coloring has applications in register allocation, scheduling, frequency assignment in wireless networks, and sparse matrix computation. Our focus is on parallelizing the greedy coloring heuristic and evaluating the tradeoffs between CPU multi-threading (OpenMP) and GPU massive parallelism (CUDA).

***Key Data Structures***

The graph is stored in Compressed Sparse Row (CSR) format, the standard representation for sparse graph algorithms on modern hardware.

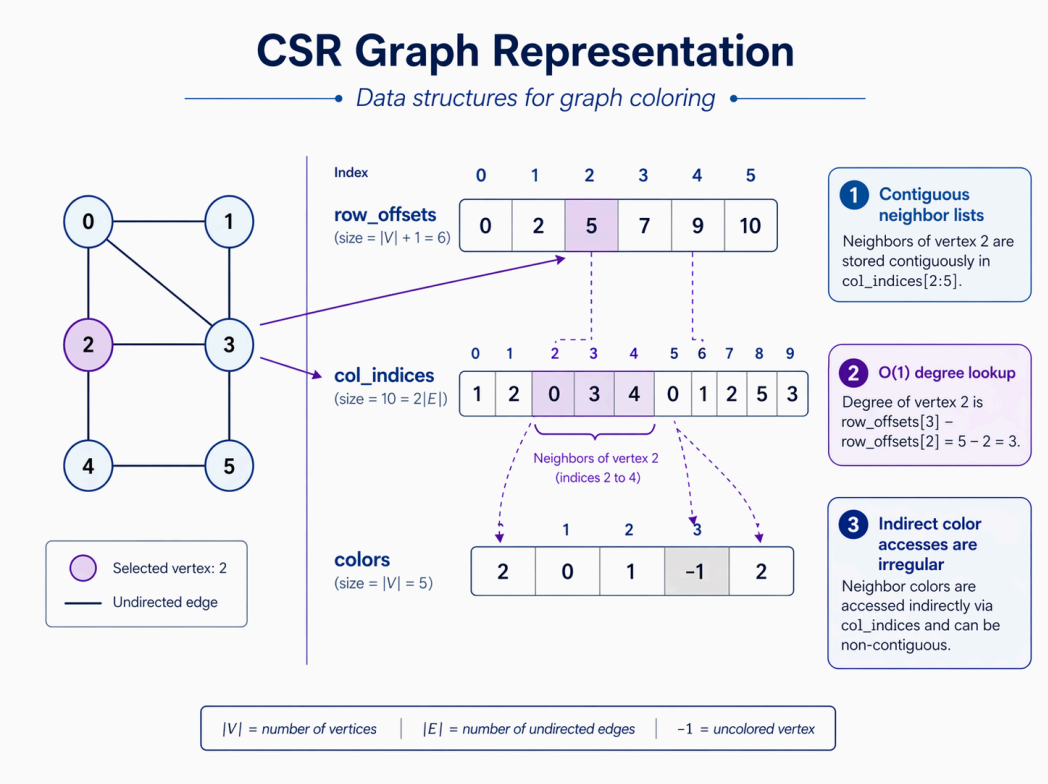



Figure 1: Compressed Sparse Row Graph Representation

CSR consists of two integer arrays: row_offsets (size n+1), where row_offsets[v] and row_offsets[v+1] delimit the range of vertex v's neighbors in the second array col_indices (size m), which stores the neighbor vertex IDs contiguously. This layout provides O(1) degree lookup via row_offsets[v+1] - row_offsets[v] and sequential memory access when iterating a vertex's neighbors — critical for cache performance. Edges are stored in both directions (u to v and v to u) so that each vertex's neighbor list is self-contained.

The coloring state is a single integer array colors[0..n-1], initialized to -1 (uncolored) and filled with color assignments as the algorithm progresses. A per-thread used[] byte array tracks which colors are occupied by a vertex's neighbors during the greedy assignment step. On the GPU, where per-thread arrays are infeasible at scale, this is replaced by a 128-bit register bitmask (four 32-bit scalar variables).

***Key Operations***

The core operation is the greedy color assignment for a single vertex v: scan all neighbors of v, mark their colors as "used" in a temporary array, then assign v the smallest color not marked. This operation is O(degree(v)) per vertex, and O(V + E) total for a single sequential pass. The computational cost per vertex varies dramatically on power-law graphs — hub vertices with degree in the thousands require scanning thousands of neighbors, while leaf vertices with degree 2–4 complete almost instantly.

***Algorithm: Speculative Coloring with Conflict Resolution***

The sequential greedy algorithm is inherently ordered — vertex v's color depends on the colors already assigned to its neighbors.

We break this dependency using speculative execution (adapted from the Gebremedhin-Manne algorithm), where all vertices are colored simultaneously in parallel, accepting that some adjacent vertices may receive the same color due to concurrent writes. A subsequent conflict detection phase identifies these violations, and only the conflicting vertices are recolored in the next round. The algorithm iterates until no conflicts remain, typically converging in 1–2 rounds.

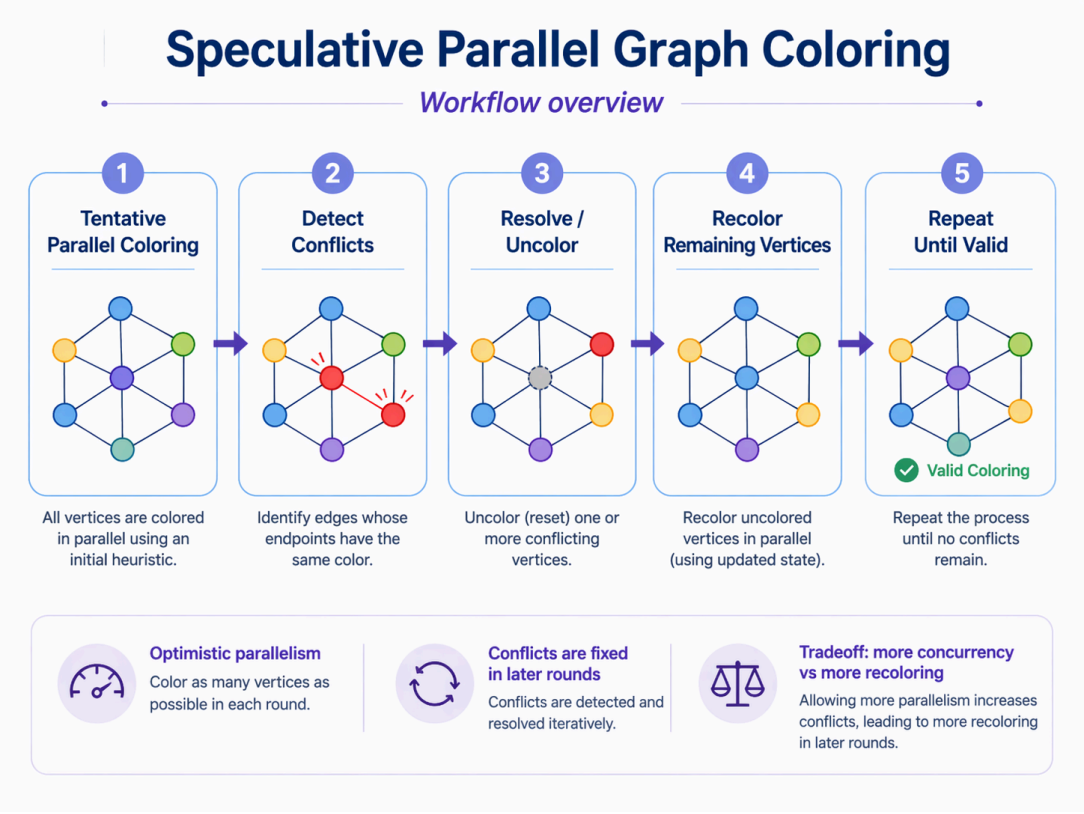



Figure 2: Speculative Coloring Algorithm

Our implementation extends this with multi-phase hub preprocessing. Before speculative coloring begins, the highest-degree "hub" vertices (degree > 8x average) are colored sequentially. These hubs are connected to a disproportionate fraction of the graph and are the primary source of conflicts when colored concurrently. Coloring them first optimally with zero conflicts dramatically reduces the conflict rate in the parallel phase. The remaining vertices are then sorted by descending degree (Largest Degree First ordering) and colored speculatively in parallel.

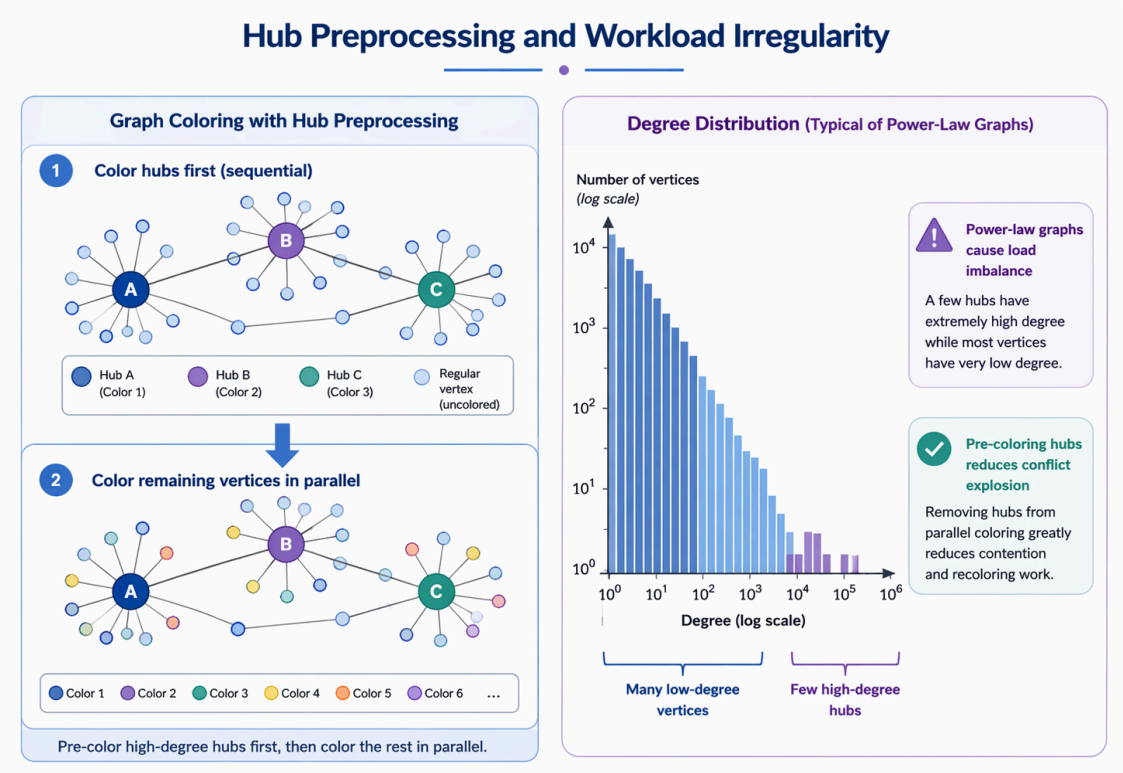



Figure 3: Hub Processing Optimization

*Inputs and Outputs*

The input is an undirected sparse graph, represented as a CSR adjacency structure. The output is an integer color assignment for each vertex such that adjacent vertices receive different colors, along with the total number of distinct colors used.

***Workload Characteristics***

*Parallelism:* The speculative coloring phase is embarrassingly data-parallel, where each vertex's greedy assignment reads only its neighbors' colors and writes only its own color. With hundreds of thousands to millions of vertices, the available parallelism far exceeds any practical core count. However, the parallelism is limited by conflicts since adjacent vertices colored simultaneously may require recoloring, and the conflict detection phase requires a global barrier to ensure all colors are visible before checking.

*Dependencies*: The fundamental dependency is between adjacent vertices, as their colors must differ. In the sequential algorithm, this is enforced by processing order. The sequential baseline is greedy coloring in a fixed vertex order, but the number of colors used depends on the ordering and is not necessarily optimal. In the parallel algorithm, dependencies are handled speculatively. Violated dependencies create conflicts that are resolved iteratively. Hub preprocessing introduces a serial dependency where hubs must be colored before regular vertices to avoid the combinatorial explosion of hub-hub conflicts.

*Locality**:*** CSR provides excellent spatial locality for neighbor iteration (neighbors are contiguous in col_indices). However, the colors of those neighbors are scattered throughout the colors[] array, and accessing colors[col_indices[i]] for each neighbor is an indirect, data-dependent memory access with poor spatial locality. This irregular access pattern is the primary bottleneck on both CPU and GPU. On the CPU, the multi-level cache hierarchy absorbs many of these irregular accesses. On GPU, this irregular gather pattern is harder to optimize because adjacent threads often access unrelated entries in colors[], reducing memory coalescing and exposing memory latency.

*Load balance**:*** On power-law graphs (R-MAT, social networks), the degree distribution is heavily skewed, where a small fraction of hub vertices have degrees in the hundreds to thousands, while the majority have degrees under 20. This creates severe load imbalance under static scheduling, as a thread assigned a hub vertex does 100x more work than a thread assigned a leaf. Because conflicts are concentrated around high-degree vertices, preprocessing hubs changes both color quality and convergence behavior. We address this with adaptive scheduling — dynamic scheduling with chunk size 256 for irregular graphs (coefficient of variation > 0.3), static scheduling for regular graphs (grids, Erdos-Renyi).

*SIMD amenability**:*** The greedy coloring kernel is not well-suited to SIMD execution. The inner loop iterates over a variable number of neighbors (data-dependent loop bounds), performs indirect memory accesses (gather pattern on colors[]), and contains data-dependent branches (checking if a color is used). The most SIMD-friendly component is the used[] array reset (std::fill / memset), which compilers can vectorize. The GPU's SIMT execution model faces similar challenges — threads within a warp process vertices with different degrees, causing warp divergence when some threads finish their neighbor scan before others.

## CPU Approach:

### Technologies Used

We implemented the CPU version in C++17 using OpenMP for multicore parallelism. Our main target platform was the GHC machines, specifically the 8-core Intel CPU configuration. The final system supports two CPU algorithms. An optimized speculative coloring algorithm and a hybrid speculative-plus-Jones-Plassmann (JP) variant.

All implementations operate on graphs stored in CSR (Compressed Sparse Row) format, which maps naturally to sparse graph traversal because each vertex's neighbor list is stored contiguously in memory. The coloring state itself is stored in an integer array colors[0..n-1], initialized to -1 for uncolored vertices. Each worker thread also maintains a private used[] array to mark colors already occupied by a vertex's neighbors during greedy assignment.

### Mapping the Problem to the CPU

Our starting point was a standard sequential greedy first-fit baseline. For each vertex v, the algorithm scans all neighbors of v, marks their colors as unavailable, and assigns v the smallest legal color. This baseline is simple and produces good color quality, but it is inherently sequential because the color chosen for one vertex depends on colors already assigned to its neighbors.

To map the problem to parallel hardware, we switched to speculative coloring with conflict resolution based on the Gebremedhin-Manne idea. Instead of enforcing a global serial order, we maintain a worklist of vertices to color. In each round, all vertices in the worklist are tentatively colored in parallel, and a second phase detects conflicts where adjacent vertices ended up with the same color. When a conflict is found, the higher-index vertex loses, has its color reset to -1, and is inserted into the next-round worklist. This deterministic tie-breaking guarantees progress, and in practice, the algorithm usually converges in one or two rounds.

On the CPU, the mapping is straightforward, where vertices in the current worklist are distributed across OpenMP threads with a parallel for, and each thread independently colors its assigned vertices using the shared CSR graph and shared colors[] array. The problem is highly data-parallel during tentative coloring because each thread writes only to its own vertex's color, but dependencies reappear in conflict detection, which requires a barrier after tentative coloring so that all colors from the round are globally visible before checking.

**Changes from the Serial Algorithm**
We changed the serial greedy algorithm substantially to make it map better to shared-memory parallelism.

First, we introduced speculative execution with conflict resolution, replacing the single serial pass with iterative worklist-based rounds. This exposed vertex-level parallelism while still preserving correctness.

Second, we added Largest-Degree-First (LDF) ordering before speculative coloring. Coloring vertices in arbitrary order caused more conflicts and slightly worse color quality, so we sorted the worklist by descending degree. This helps because high-degree vertices are both the most expensive vertices and the most likely to conflict, so giving them earlier access to the color palette reduces both conflict counts and the number of colors used.

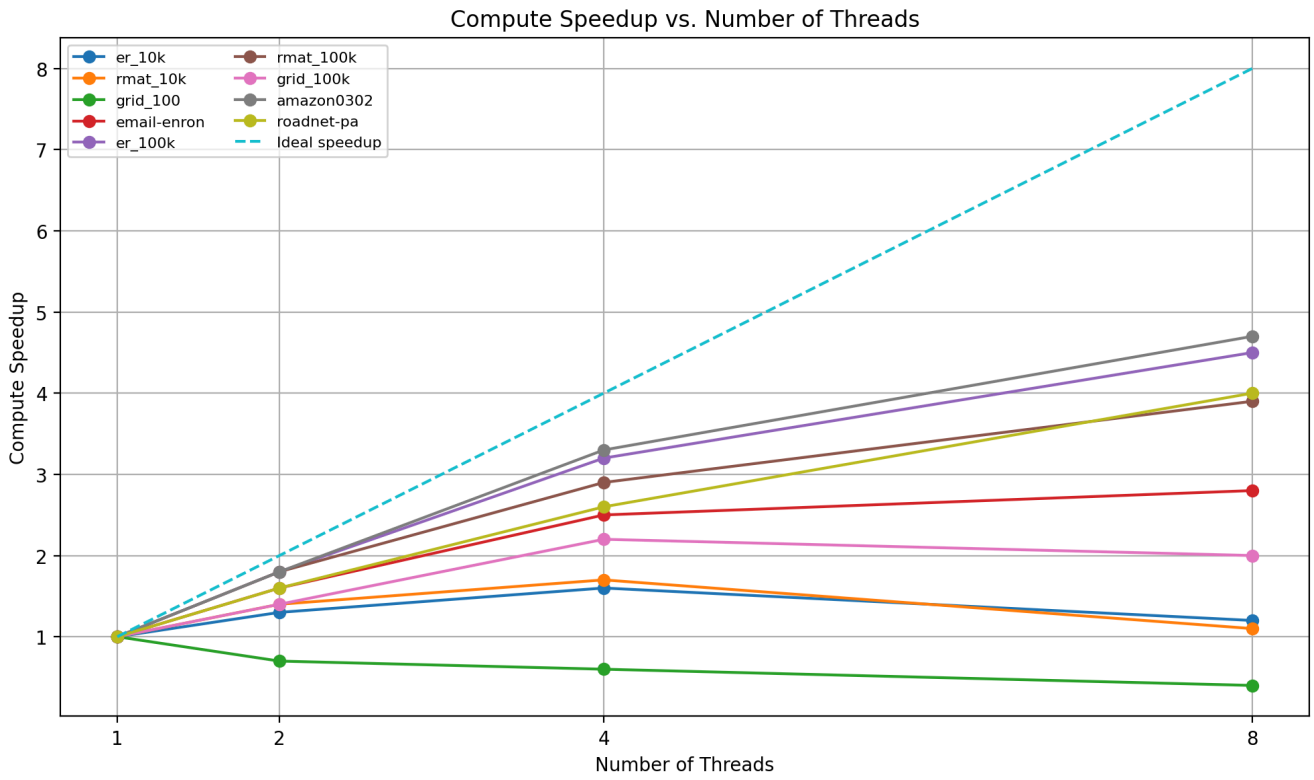
Third, we introduced hub preprocessing. Real power-law graphs, such as R-MAT and social networks, contain a small set of very high-degree hub vertices that dominate both runtime and conflict behavior. Coloring such hubs concurrently is a poor fit for shared-memory speculative coloring because their neighborhoods overlap heavily, producing dense conflict clusters. To address this, we identify high-degree hubs before speculative coloring begins and color them sequentially using the optimal greedy rule. The remaining regular vertices are then colored in parallel. This deliberately introduces a serial phase, but it dramatically reduces conflict rates in the parallel phase and often enables convergence in a single speculative round on skewed graphs.

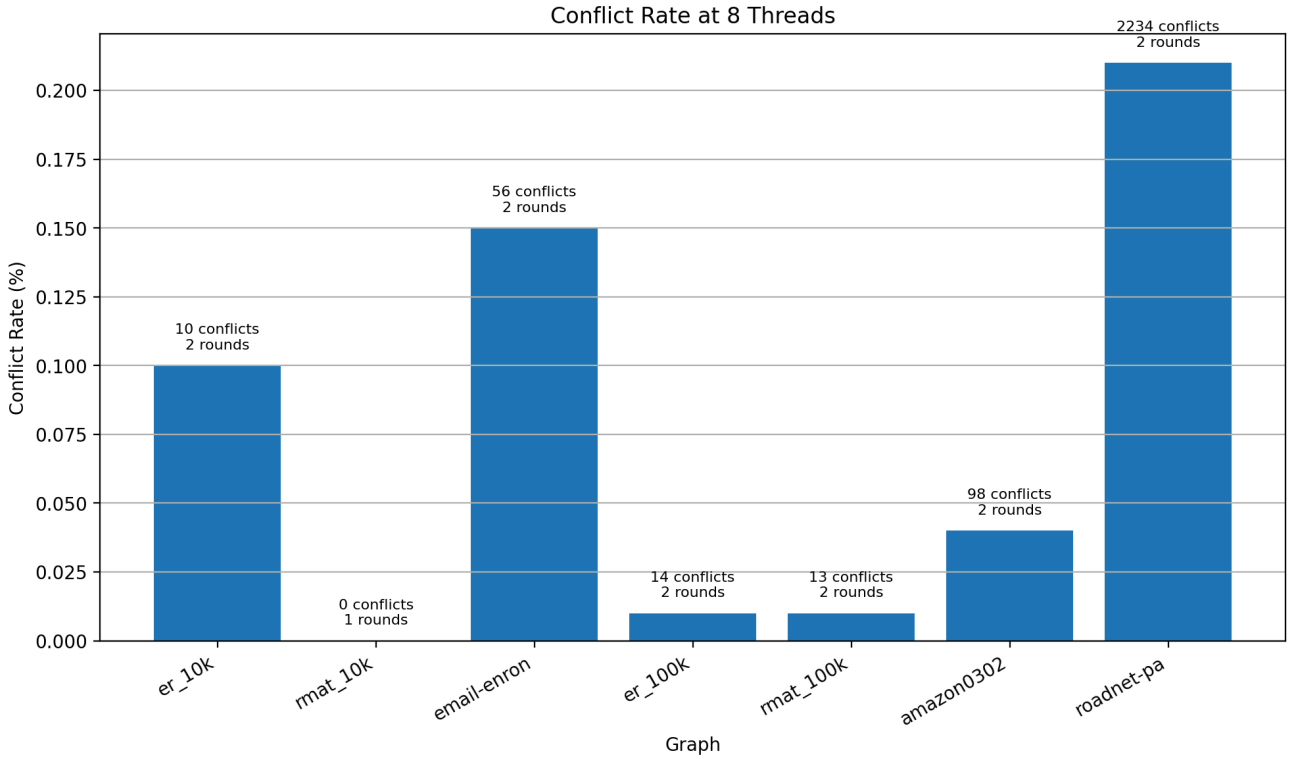
Finally, in the hybrid version, we replaced repeated speculative recoloring with a JP refinement phase. After one speculative round colors most of the graph, only the remaining conflict set is processed using priority-based independent-set rounds. This gives conflict-free progress on the residual graph while avoiding repeated speculative rounds on a small remaining set.

**Optimization Journey**
The CPU implementation evolved through many iterations. We first implemented the purely sequential greedy baseline.

***First iteration***
Then, as the first iteration of our parallel algorithm, we implemented a correct but modest speculative baseline, profiled it, and then systematically attacked each bottleneck: fork/join overhead, conflict serialization, poor ordering, load imbalance, and initialization cost.

Below are the compute speedup results for our initial speculative parallel implementation, which shows some graphs even decreasing in performance.


Compute Speedup vs. Number of Threads
er_10k
rmat_10k
grid_100
email-enron
er_100k
rmat_100k
grid_100k
amazon0302
roadnet-pa
Ideal speedup
Compute Speedup
Number of Threads
1
2
4
8


Conflict Rate at 8 Threads
Conflict Rate (%)
10 conflicts
2 rounds
0 conflicts
1 rounds
56 conflicts
2 rounds
14 conflicts
2 rounds
13 conflicts
2 rounds
98 conflicts
2 rounds
2234 conflicts
2 rounds
er_10k
rmat_10k
email-enron
er_100k
rmat_100k
amazon0302
roadnet-pa
Graph

***Second Iteration***

Our first major improvement was restructuring the iterative algorithm around a persistent parallel region. Initially, we launched a fresh #pragma omp parallel region in every round. This caused substantial fork/join overhead, especially on small and medium graphs. We changed the implementation so that a single outer parallel region wraps the entire iterative algorithm, with omp for driving tentative coloring and conflict detection, and omp single handling worklist management between rounds. This reduced synchronization overhead.

We then focused on lock-free conflict collection, which had become a major hot-path bottleneck. Our baseline conflict-detection phase merged conflicting vertices into a shared next-round list using #pragma omp critical, which serialized the entire phase under contention. We replaced this first with a per-vertex conflict flag array, then with per-thread local lists, and finally with atomic worklist compaction, where each conflicting vertex uses fetch_add on a shared counter to reserve its slot in the next worklist. This lock-free compaction scheme was both the simplest and the best-scaling version, and it removed a major sequential bottleneck from the speculative loop.

We also found that vertex ordering mattered as coloring vertices in arbitrary order caused more conflicts and slightly worse color quality, so we introduced Largest-Degree-First (LDF) ordering, sorting the worklist by descending degree before speculative coloring. That mapping is beneficial because high-degree vertices are both the most expensive vertices and the most likely to conflict, so giving them early access to the color palette reduces both conflict counts and the number of colors used.

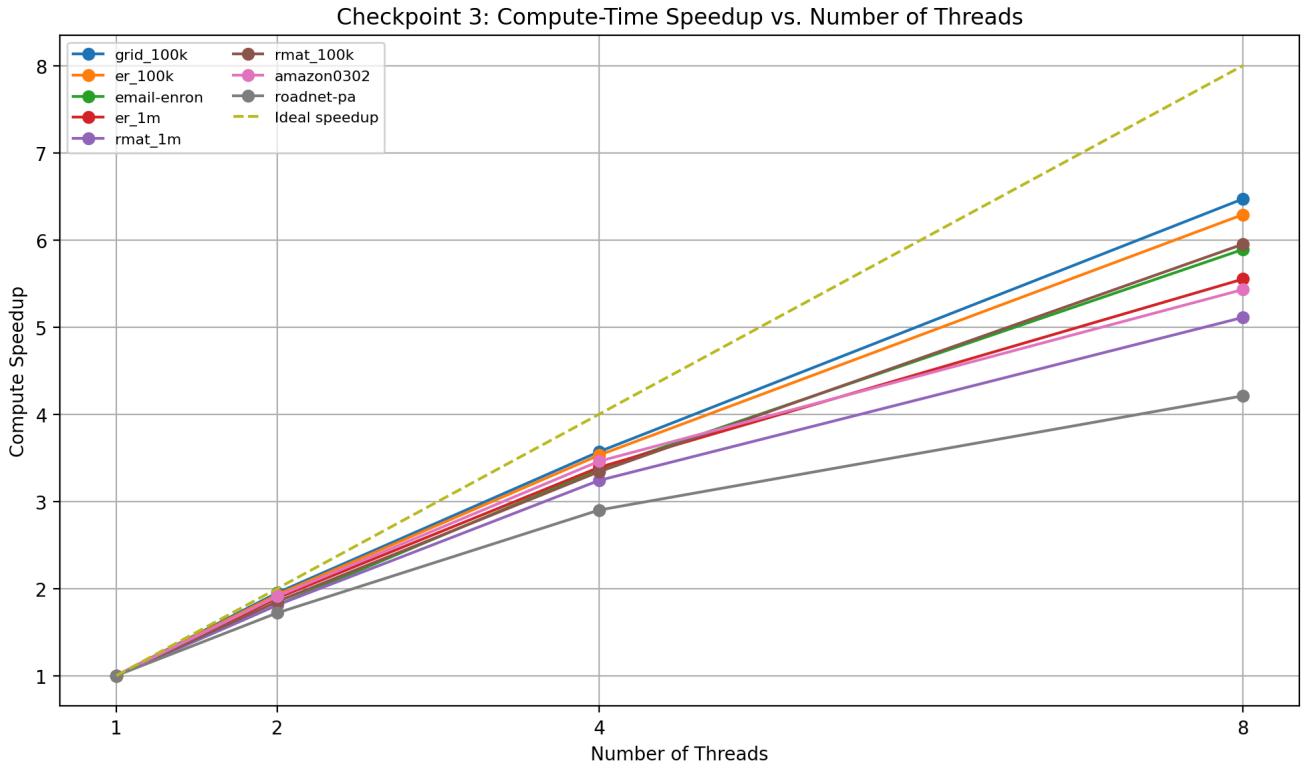
***Third Iteration (Final)***

A more algorithmic optimization was hub preprocessing. We found that on power-law graphs, hub vertices were responsible for a disproportionate share of conflicts and runtime. By removing them from the speculative phase and coloring them greedily first, we dramatically reduced the amount of rework in later rounds. We later tuned the hub threshold from 4× average degree to 8× average degree, shrinking the serial hub set without materially hurting color quality. For example, on rmat_1m, this reduced the hub count from 44,700 to 21,702 and improved total time by about 13% while increasing the color count by only 1.

We also optimized around load balance and initialization cost. Since degree distributions vary dramatically across graphs, we compute the coefficient of variation of vertex degree and choose the OpenMP schedule adaptively. Static for regular graphs, such as grids and Erdős-Rényi, and dynamic,256 for irregular graphs such as R-MAT and social networks. This improved balance on power-law graphs while avoiding scheduling overhead on uniform graphs.

Later, once the compute phase scaled better, initialization itself became the dominant cost. We parallelized degree statistics with reductions, parallelized hub/regular partitioning using thread-local lists, and parallelized large sorts using __gnu_parallel::sort on GCC.

We then improved locality further by precomputing degrees into a dense contiguous array, switching LDF ordering to counting sort when possible, skipping sorting entirely for regular graphs, replacing vector<bool> with vector<char> in the hot used[] loop, and fusing multiple initialization passes into a single parallel pass.



**What We Tried That Didn't Work**

Not every optimization survived into the final design. Several ideas looked good on paper but were measurably worse in practice.

*Standalone Jones-Plassmann.* First, we implemented standalone Jones–Plassmann (JP) as an alternative to speculative coloring. JP avoids direct conflicts by coloring only vertices that are local-priority winners among their uncolored neighbors, but in practice, it required far too many rounds, and the barrier overhead dominated, making it slower as the thread count increased. We therefore removed standalone JP from the final codebase and kept it only as a refinement step in a hybrid approach, where one speculative round colors most vertices and JP handles the small remaining conflict set.

*Hybrid speculative + JP refinement*. We also implemented a hybrid algorithm that performs one speculative round and then switches to JP on the remaining conflict set. This version was correct and avoided the very high round counts of pure JP, but it did not materially outperform the optimized speculative algorithm. The reason is that once hub preprocessing and degree ordering were in place, the speculative algorithm already had very low conflict rates, often below 0.25%, and usually converged in only 1 or 2 rounds. At that point, there was very little remaining work for JP to improve, so the extra priority computation, refinement rounds, and synchronization mostly added overhead rather than reducing total runtime

*Parallel hub coloring.* Since sequential hub preprocessing became a major Amdahl's law bottleneck, we tried coloring hubs in parallel with the same speculative scheme used on regular vertices. This reduced initialization time, but it badly damaged color quality because high-degree hubs have heavily overlapping neighborhoods. On rmat_1m, the color count increased from 77 to 133, with similar smaller regressions on other irregular graphs, so we reverted to sequential hub coloring.

*CAS-based fused coloring.* We analyzed a CAS-based fused coloring approach that would try to eliminate one barrier by assigning colors and detecting conflicts immediately, but concluded that this is incompatible with standard speculative coloring since conflict detection fundamentally requires all colors from the current round to be globally visible first, so the barrier cannot be removed without changing the algorithm itself.

*Critical-section conflict collection.* Another early design used #pragma omp critical to merge conflicts into the next-round worklist. This was correct, but on irregular graphs, it became a serious serialization point and directly motivated the move to lock-free atomic worklist compaction

**Synchronization**

The speculative coloring algorithm has a small, fixed synchronization skeleton. One barrier after tentative coloring, so every thread observes the round's final colors before conflict detection, and one barrier after worklist compaction, so every thread agrees on the next round's input.

*Persistent parallel region*

Our initial implementation launched a fresh #pragma omp parallel region on every round. On small graphs, this costs more than the actual coloring. We restructured the algorithm so that a single outer #pragma omp parallel wraps the entire iterative loop, with omp for driving tentative coloring and conflict detection, and omp single running the worklist-swap bookkeeping between rounds. This converted round-to-round synchronization from fork/join into OpenMP's lightweight built-in barrier, which was measurably cheaper on every graph and dominant on the small ones.

*Lock-free worklist compaction*

The conflict-detection phase ends with an atomic-increment compaction where each losing vertex executes idx = atomic_fetch_add(&next_size, 1) and writes itself into next_worklist[idx]. This replaces a critical section with a single atomic R/M/W, and in the typical case (conflict rates under 0.25%), there is essentially no contention on next_size because so few threads increment it per round.

*Implicit barriers only*

We deliberately do not use #pragma omp barrier anywhere in the hot path. Every synchronization point is the implicit barrier at the end of an omp for or omp single, which OpenMP can fuse with the work-distribution logic. The only place we force an explicit memory ordering is the atomic counters themselves, which use sequentially consistent semantics so that the value read by omp single to swap worklists is the value written by the last compactor.

**Reasons for Imperfect Speedup**

At 8 threads on an 8-core CPU, the ideal compute speedup is 8×. We observe 5.4×–6.9× on the large graphs, i.e., parallel efficiency of roughly 68%–86%. The gap from linear has three distinct sources, each of which dominates on different workloads.

*Amdahl's law from sequential init*

The hub-preprocessing phase and parts of the adjacency-setup phase are either serial or serialized by the hub dependency structure. On rmat_1m at 8T, init takes 36.4 ms while compute takes only 8.1ms. Even if compute scaled perfectly, the ceiling on total speedup is bounded by the serial fraction: at $s = 36.4 / (36.4 + 116.2 \times 8/8) = 24\%$ serial (using 1T compute 116.2 ms), Amdahl caps total speedup at $1/(s + (1-s)/8) \approx 3.1\times$, which matches the observed total-time speedup on that graph. This is not a bug in the parallel implementation, but rather it is the cost of the design choice to color hubs sequentially for quality.

*Memory bandwidth saturation*

The core operation is colors[col_indices[i]], an indirect gather into a 4 MB–16 MB array that mostly misses L1 and often misses L2. On the large graphs, our total L3 miss counts stay flat as threads scale (from the cache-miss analysis), which tells us we are not manufacturing extra misses, but it also tells us that the 8 cores are sharing the same L3 and the same memory controller. The er_1m per-thread average of $\sim 5 \times 10^6$ misses at 8T is larger than the ~3 MB L3 slice per core, so every thread is periodically waiting on DRAM. This shows up as ~5.6× speedup on er_1m instead of 8×

*Barrier + straggler tail*

Even with adaptive scheduling, two rounds of speculative coloring contain four implicit barriers per round (tentative for, detection for, compaction barrier, single). Each barrier waits for the slowest thread, and on power-law graphs, one thread almost always draws a hub-rich chunk that takes 20–50 μs longer than the others. Summed over 2 rounds × 4 barriers, this costs roughly 100–400 μs, which is substantial on email-enron (0.21 ms compute), where it accounts for measurably less-than-linear scaling (5.81× on 8 threads).

**Performance Drop-off**
Our compute-time speedup is strong on the large graphs, but weak or negative on small ones

*Small-graph*
On grid_100 (10k vertices, 20k edges), 1T compute is 0.11 ms and 8T compute is 0.02 ms, which is a 5.5× compute-time speedup, but 1T total is: init 0.06 ms + compute 0.11 ms = 0.17 ms, where 8T total is: init 0.41 ms + compute 0.02 ms = 0.43 ms, so the parallel run is 2.5× slower end-to-end. This happens because parallel init pays a fixed cost of launching an omp parallel region, allocating per-thread used[] arrays, and partitioning the worklist, which on tiny graphs is larger than the entire serial run.

*Medium-graph*
On graphs in the 100k-vertex range (er_100k, email-enron, rmat_100k) we see good 1T→4T scaling (typically 3.0×–3.5×) followed by partial drop-off 4T→8T (adding another 1.5×–1.9× for a total of 5.4×–6.9×). This is the barrier-tail effect, where at 4 threads, a barrier waits for 3 other threads to finish, and at 8 threads it waits for 7, and the probability of at least one straggler in a power-law distribution of per-chunk work grows roughly linearly in T. The absolute compute times at this scale (1–7 ms) are small enough that a 50–100 μs barrier tail matters.

Summary of the drop-off curve. Speedup peaks around 4–8 threads on graphs ≥ 100k vertices and decays toward zero (or negative end-to-end) as graph size shrinks below ~50k. Past 8 threads on an 8-core machine, we would expect further decay from hyper-thread contention on the shared memory subsystem, though we did not benchmark 16T on the 8-physical-core GHC nodes because the additional threads would oversubscribe the hardware rather than add useful parallelism.

## GPU Approach:

### Technologies Used

We implemented the GPU version in CUDA C++ and targeted the GHC machines, using an NVIDIA RTX 2080 GPU. The host-side control logic was written in C++, while the device-side coloring kernels were implemented in CUDA. Because CUDA 11.7 on GHC is not compatible with the default GCC 13 toolchain, we used a split build where the CUDA kernel file (coloring_cuda.cu) was compiled with nvcc -ccbin g++-11, while the host-side wrapper (coloring_gpu_wrapper.cpp) was compiled as standard C++.

### Mapping the Problem to the GPU

The GPU implementation uses the same CSR graph representation as the CPU version. The CSR arrays (row_offsets, col_indices) and the current worklist are copied to device memory, and one CUDA thread is assigned to each active vertex in the worklist. Thread blocks contain 256 threads.

Each speculative round is implemented as two kernels launched from the host, including a tentative-color kernel and a conflict-detection kernel. In the tentative-color kernel, each thread scans the neighbor list of its assigned vertex and greedily selects the smallest available color. In the conflict-detection kernel, threads re-scan the same vertices, detect the same-color conflicts with neighbors, invalidate the losing endpoint, and compact losing vertices into the next-round worklist using an atomic counter. After each round, the host copies back only the size of the next worklist to determine whether another round is needed.

### Changes from the Serial Algorithm

We changed the serial greedy algorithm substantially to make it better suited to the GPU. First, we retained CPU-side hub preprocessing where high-degree hub vertices are identified and colored sequentially on the host before launching any kernels. This reduces GPU conflict rates and keeps the most irregular vertices out of the device worklist. Second, the remaining regular vertices are sorted by descending degree before transfer to the GPU, so adjacent threads in a warp tend to process vertices with similar neighbor-list lengths, reducing warp imbalance. Third, we replaced the CPU-style per-thread used[] array with a 128-bit register-resident bitmask implemented as four 32-bit scalars. This lets each thread track the common-case color set entirely in registers, avoiding large per-thread local arrays.

For typical workloads, 128 colors are enough, but in case it isn’t, we implemented a fallback path with a brute force approach, checking neighbor colors until it finds the first valid assignment. Finally, instead of a single ordered serial pass, the GPU uses speculative execution with repeated host-driven kernel rounds.

**Optimization Journey**

The GPU implementation went through several rounds of refinement. Our first working CUDA version established correctness, but its timing significantly overstated end-to-end cost because it included CUDA context initialization on the first call. We fixed this by explicitly warming up the CUDA runtime before starting wall-clock timing and by separating GPU time into three components: transfer/setup time, kernel-only compute time, and finalization time. This gave us a much more meaningful comparison between GPU kernel throughput and true end-to-end runtime.

After the timing methodology was corrected, we focused on reducing GPU-side inefficiency. We kept hub preprocessing on the CPU so that the highest-degree vertices were removed from the GPU worklist before speculative coloring began. We also sorted the remaining regular vertices by descending degree before transfer so that threads in the same warp would process vertices with more similar neighbor-list lengths, reducing warp imbalance on irregular graphs. Within the kernel, we replaced the CPU-style per-thread used[] array with a 128-bit register-resident bitmask implemented as four 32-bit scalars, allowing the common-case color search to remain entirely in registers. Finally, we used pinned host memory for the per-round conflict-count readback to reduce the overhead of the host-driven speculative loop.

We also explored an alternative edge-parallel GPU design in which edge-level passes were used for forbidden-color accumulation and conflict detection. Although this version was correct, it consistently produced much higher conflict rates, more rounds to convergence, and worse total runtime than the vertex-based GPU design. In practice, the extra global memory traffic, atomic updates, and edge-wide passes outweighed any benefit from exposing more parallelism. Based on these experiments, we retained the current vertex-based speculative GPU implementation as our final GPU approach.

**What We Tried and Didn’t Work**

On the GPU side, the main unsuccessful design we explored was an edge-parallel coloring variant. Instead of assigning one thread to each active vertex, this version used edge-level passes to accumulate forbidden colors and detect conflicts. The motivation was that edge-level work might expose more parallelism and better utilize the GPU on irregular graphs. In practice, however, this design performed worse than our final vertex-based implementation. It produced much higher conflict rates, required more rounds to converge, and had higher total runtime across the benchmark suite. The extra global memory traffic, heavy use of atomics, and repeated scans over all edges outweighed any benefit from the additional parallelism, so we kept the vertex-based speculative GPU algorithm as the final design.

**Synchronization**
GPU synchronization was one of the main performance bottlenecks. Our speculative CUDA implementation is organized as a host-driven loop, with one kernel for tentative coloring and one kernel for conflict detection in each round. Between rounds, synchronization is mostly explicit at the algorithm level, where the host must determine whether another worklist remains, so it copies back the next-worklist count and then decides whether to launch another pair of kernels.

At the CUDA runtime level, this synchronization is realized implicitly through blocking API calls such as the device-to-host cudaMemcpy used to read back the conflict count, which forces prior kernel work to complete before the host can proceed. As a result, each speculative round incurs both kernel-launch overhead and a host-device synchronization point. For graphs that require many rounds, such as grids or other high-conflict cases, this repeated synchronization becomes a substantial fraction of total runtime.

**Reasons for Imperfect Speedup / Bad GPU Performance**
The GPU implementation achieved strong kernel-only speedups on larger graphs, but end-to-end performance was limited by several factors. First, preprocessing and transfer costs remained significant where graph data, worklists, and initial colors had to be staged on the device, and final colors had to be copied back. Second, graph coloring is an irregular workload, so threads in the same warp often process vertices with very different neighbor-list lengths, creating warp imbalance. Third, the neighbor-color reads are indirect accesses through CSR, which reduces memory coalescing and makes the kernels more latency-sensitive. Finally, speculative coloring amplifies synchronization overhead when conflict rates are high, because additional rounds mean additional kernel launches and host-device round-trip times. Together, these effects explain why the GPU's massive parallelism translated into strong kernel throughput but only modest end-to-end gains.

We are aware that more advanced GPU graph coloring algorithms exist, including MIS/Jones-Plassmann-style methods, optimized edge-based approaches, and highly tuned GPU-specific implementations such as csrcolor-style coloring. These methods are often a better match for GPU hardware because they can reduce conflict rates, improve device-side parallel coordination, or avoid some of the host-driven synchronization overhead in our implementation. However, implementing and carefully evaluating one of these more sophisticated GPU algorithms would have required substantially more engineering time. Since the GPU portion of our project began as a stretch goal, we focused on building a correct and reasonably optimized speculative CUDA implementation and comparing it carefully against our CPU baselines.

## CPU Results:

### Performance Metrics

For each run, we measured three timing components: **init time**, **compute time**, and **total time**. Here, compute_time measures the coloring phase itself, including speculative coloring and conflict resolution rounds, while total_time includes both preprocessing and coloring.

For the CPU implementations, we report compute-time speedup relative to the 1-thread parallel implementation of the same algorithm, as well as total-time speedup when initialization is included. We also track **color count**, **number of conflicts**, **conflict rate**, **number of rounds to convergence**, and correctness.

### Experimental Setup

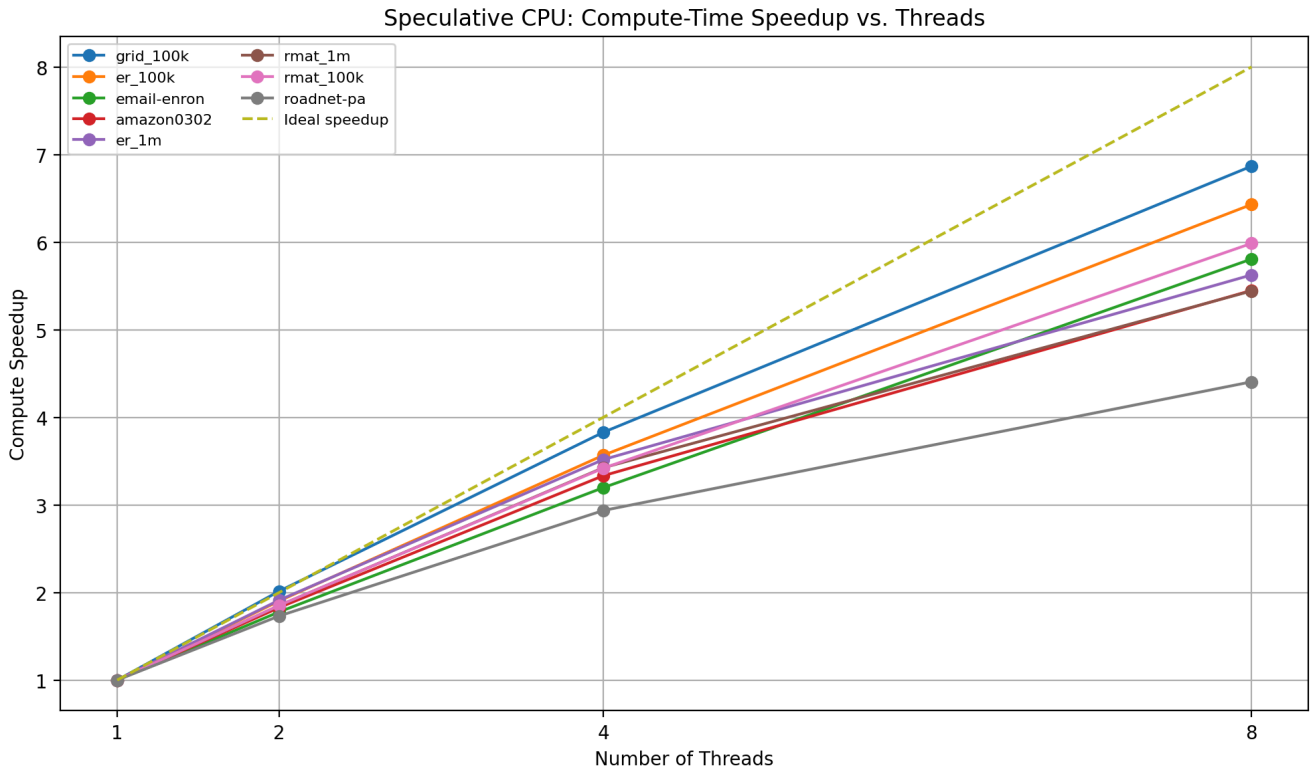
We evaluated our implementations on the GHC machines using 11 benchmark graphs spanning synthetic and real-world workloads from roughly 10K to 1M vertices. The synthetic graphs included Erdős–Rényi, R-MAT, and grid graphs, chosen to represent moderate, power-law, and highly regular degree distributions, respectively. The real-world graphs were email-enron, amazon0302, and roadnet-pa, which provide social-network, product-network, and road-network structures.

Together, these inputs cover a broad range of graph characteristics, from regular low-degree lattices to highly skewed hub-dominated power-law graphs. Since the source graphs came in multiple file formats, we built preprocessing scripts to convert them into a common CSR representation. All benchmarked outputs were validated by a correctness checker that ensures no adjacent vertices share a color.

### Baseline

For all of the results, our baseline is the speculative parallel algorithm at 1 thread.

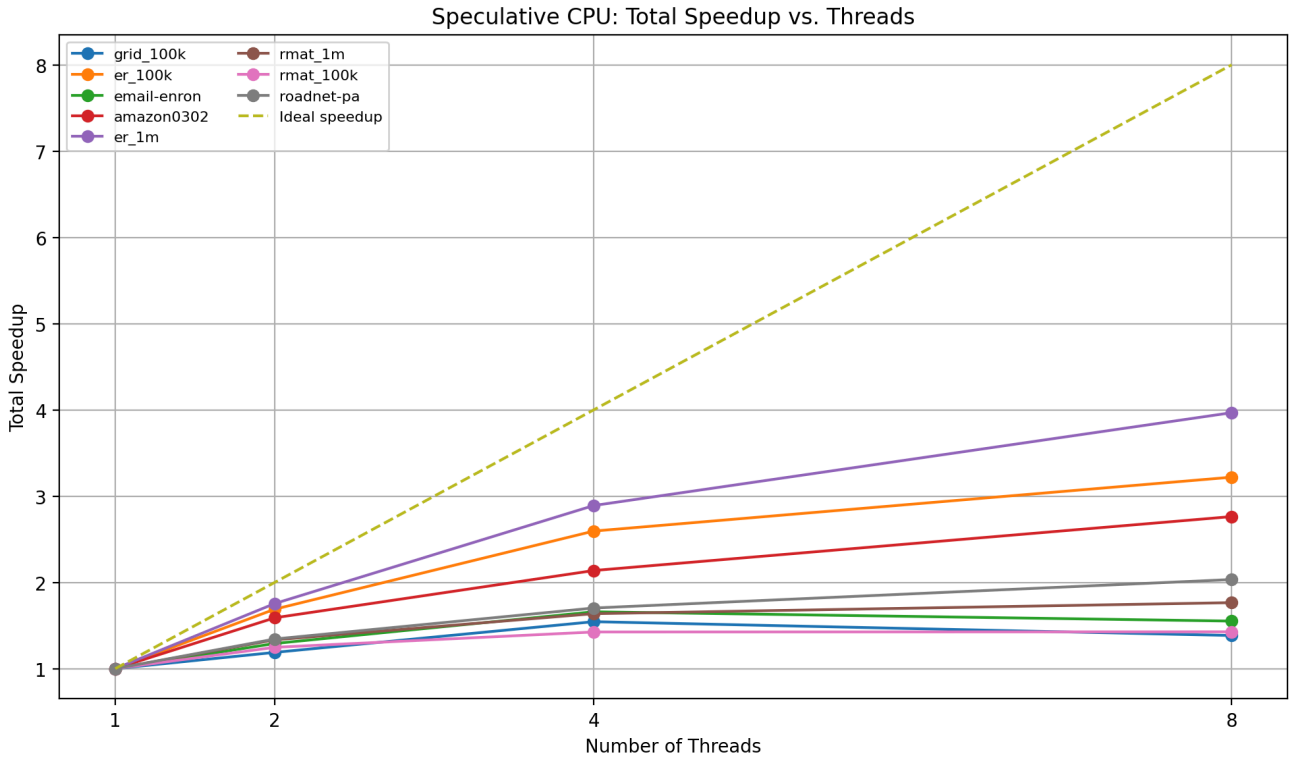
## Compute Speedup




Our final speculative OpenMP implementation achieved strong compute-time scaling on the 8-core CPU. On the larger graphs, 8-thread compute speedup ranged from about 4.4× on roadnet-pa to 6.9× on rmat_100k, with most workloads falling in the 5.4×–6.5× range. Representative examples include 6.44× on er_100k (dropping from 7.034 ms at 1 thread to 1.094 ms at 8 threads), 5.81× on email-enron (1.231 ms to 0.212 ms), 5.45× on amazon0302 (12.390 ms to 2.274 ms), 5.63× on er_1m (116.171 ms to 20.651 ms), and 5.44× on rmat_1m (44.295 ms to 8.139 ms).

These results show that the optimized compute phase scales well across both regular and irregular graphs once fork/join overhead, conflict serialization, and ordering issues are addressed.

**Total speedup**

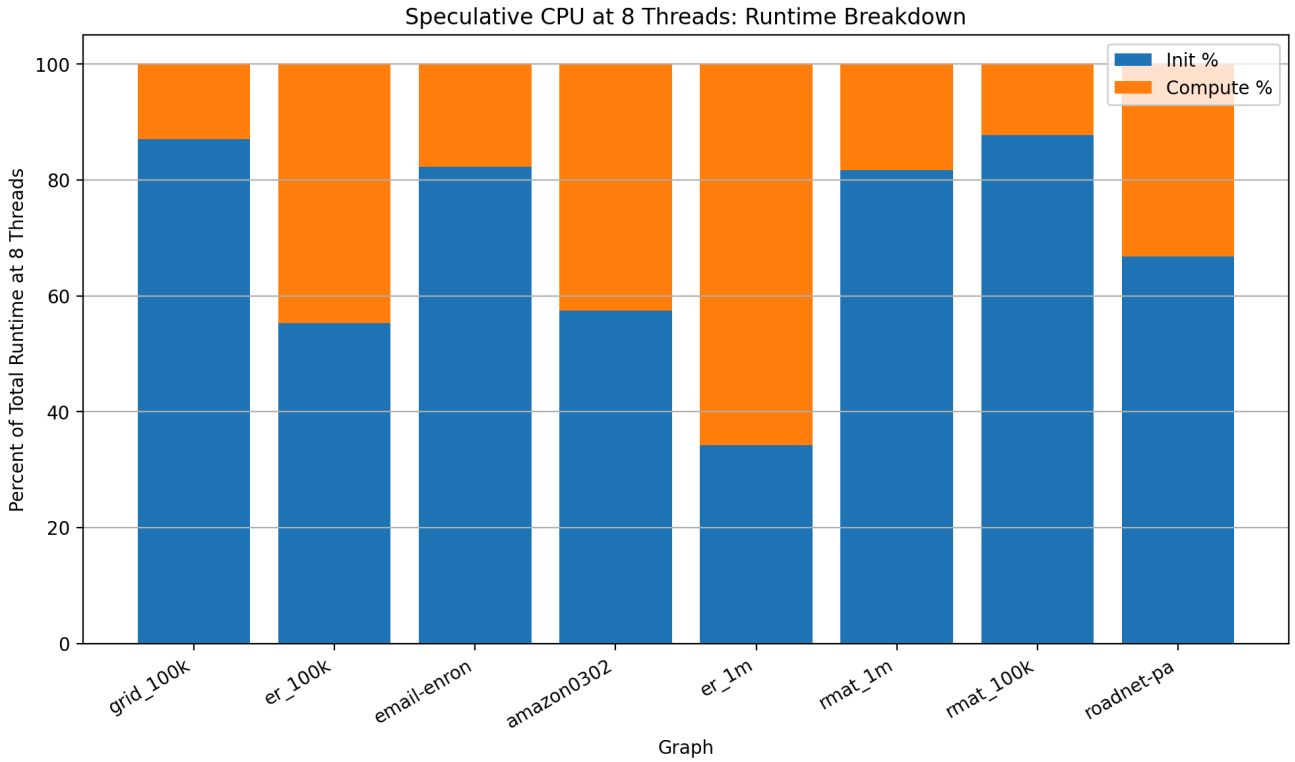



The total-time speedup is consistently lower than the compute-time speedup because initialization remains a substantial serial fraction of the algorithm. This is most visible on the large power-law graph rmat_1m, where the 8-thread run still spends 36.377 ms in init versus 8.139 ms in compute, so initialization dominates the end-to-end runtime. A similar effect appears on roadnet-pa, where init is 11.861 ms and compute is 5.900 ms at 8 threads.

In contrast, more regular workloads without many hubs, such as er_1m, benefit more from parallelization because initialization is a smaller portion of total runtime. This gap between strong compute-time speedup and more limited total-time speedup is a direct Amdahl's law effect where the sequential hub-coloring and ordering work in initialization limits end-to-end scaling even after the main coloring kernel becomes highly parallel.

The main reason why our initialization is mostly sequential is because of hub preprocessing, in which high-degree vertices are colored sequentially before the speculative phase begins. We kept this stage sequential intentionally after some experimenting. When we experimented with parallel hub coloring, color quality degraded substantially on power-law graphs because concurrent hub coloring introduced dense hub-hub conflicts and forced the algorithm to open many more colors. For example, on rmat_1m, parallel hub coloring increased the number of colors from roughly 77 to 133. We had to make the tradeoff between better coloring quality and lower conflict rates over a slightly smaller initialization cost. This design choice improves the quality and stability of the parallel phase.

**Runtime breakdown**

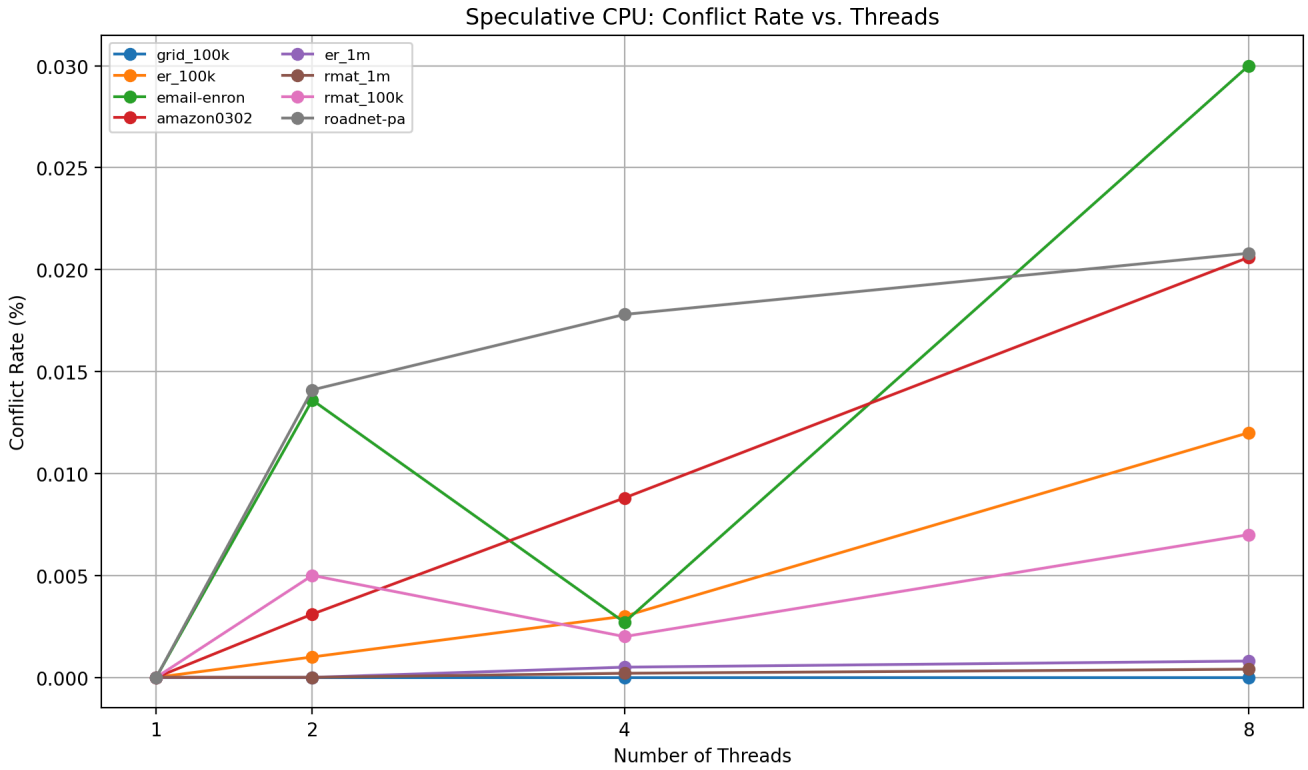



This bar graph shows the percentage of 8-thread runtime that is **init vs. computation** for each benchmark. As mentioned before, the **init phase** involves calculating degree statistics, hub identification, Lowest Degree First counting-sort, and worklist construction. The compute phase involves the main coloring work, which is speculative tentative coloring followed by conflict resolution.

This graph reinforces the idea of **Amdahl's Law.** After we pushed compute-time speedup to 5.5-7× on the compute phase, the remaining limitation on end-to-end speedup is the fraction of time spent in init. We can clearly see two types of workloads. **Init-dominated** workloads like grid_100k (87% init), rmat_100k (88%), email-enron (81%), rmat_1m (81%) spend most of their time doing preprocessing. rmat_1m is the clearest case. At 8 threads, it spends 36.4 ms of init vs. 8.1 ms of compute, with that 36 ms coming almost entirely from serial hub-coloring of 21,702 hub vertices and the degree-based ordering sort.

Meanwhile, the **compute-dominated** cases like er_1m (~66% compute), er_100k (~45%), amazon0302 (~43%) contain the graphs with either genuinely long compute phases or low hub counts: er_1m has no hubs (0 hubs due to uniform random distribution), so init is tiny relative to the 20 ms compute. A major path to improving total speedup in future work would therefore be to reduce or overlap preprocessing costs without sacrificing the color quality benefits of hub-aware initialization.

## Conflict Rate




Conflict rate measures the % of vertices whose tentative color collided with a neighbor and had to be recolored in the next round.

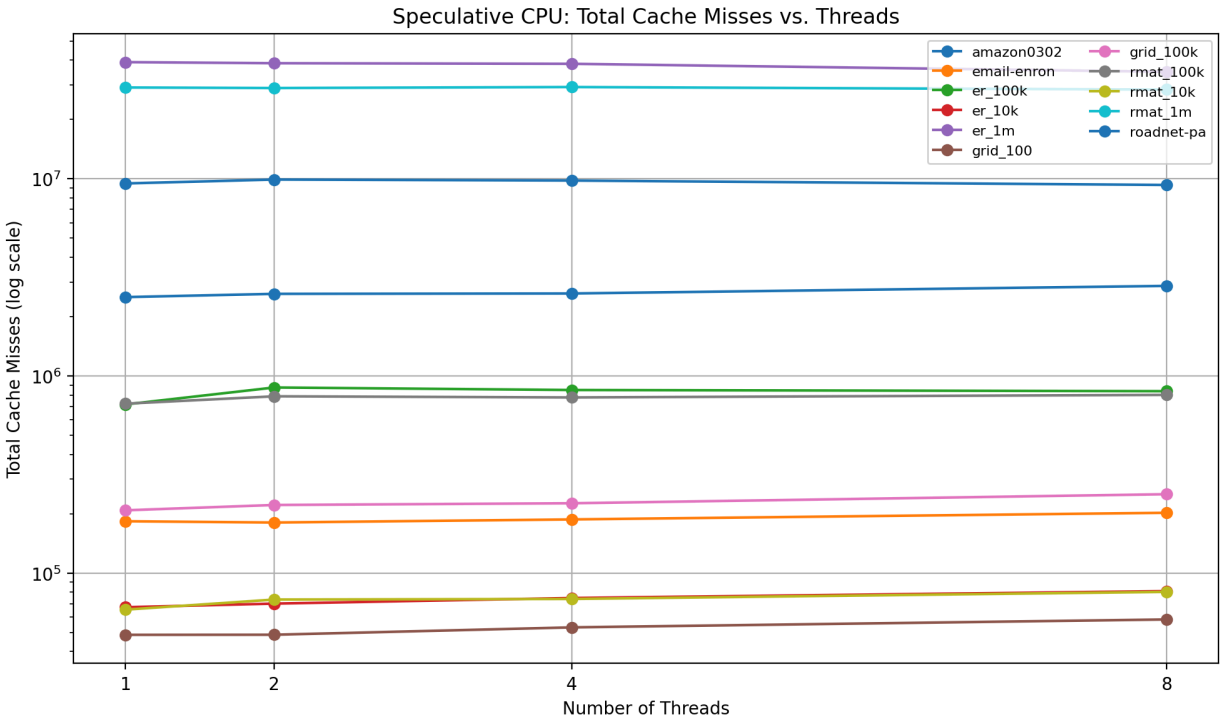
In general, the conflict rate grows as thread count increases since more concurrent tentative writes expose more opportunities for adjacent vertices to choose the same color before the decisions are committed. However, the absolute conflict rates still remain tiny across the entire sweep, never exceeding 0.03% at 8 threads. This confirms that our hub preprocessing and LDF ordering are working as designed.

The conflict behavior can be separated into several types. First, grids are inherently safe because of their bipartite nature. Next, power-law and social graphs sit in the middle. This is because during preprocessing, hubs are already handled, but the remaining power-law tail still has moderately high-degree vertices where two threads occasionally hit the same neighborhood in the same round. Large random and large power-law graphs showed a very low conflict rate scaling. Even at 8 threads, the collision count is in the only in double digits out of a million vertices. This is expected since there is so much parallelism available that threads almost never happen to pick adjacent vertices. The road-network graph is a special case since roadnet-pa grows smoothly to 0.021% at 8T despite having an average degree of only 2.83 and no hubs.

Overall, conflict rates are very low, which explains why we never see more than 2 rounds of convergence on any graph.

**Cache misses**

**i) Total cache misses plot**

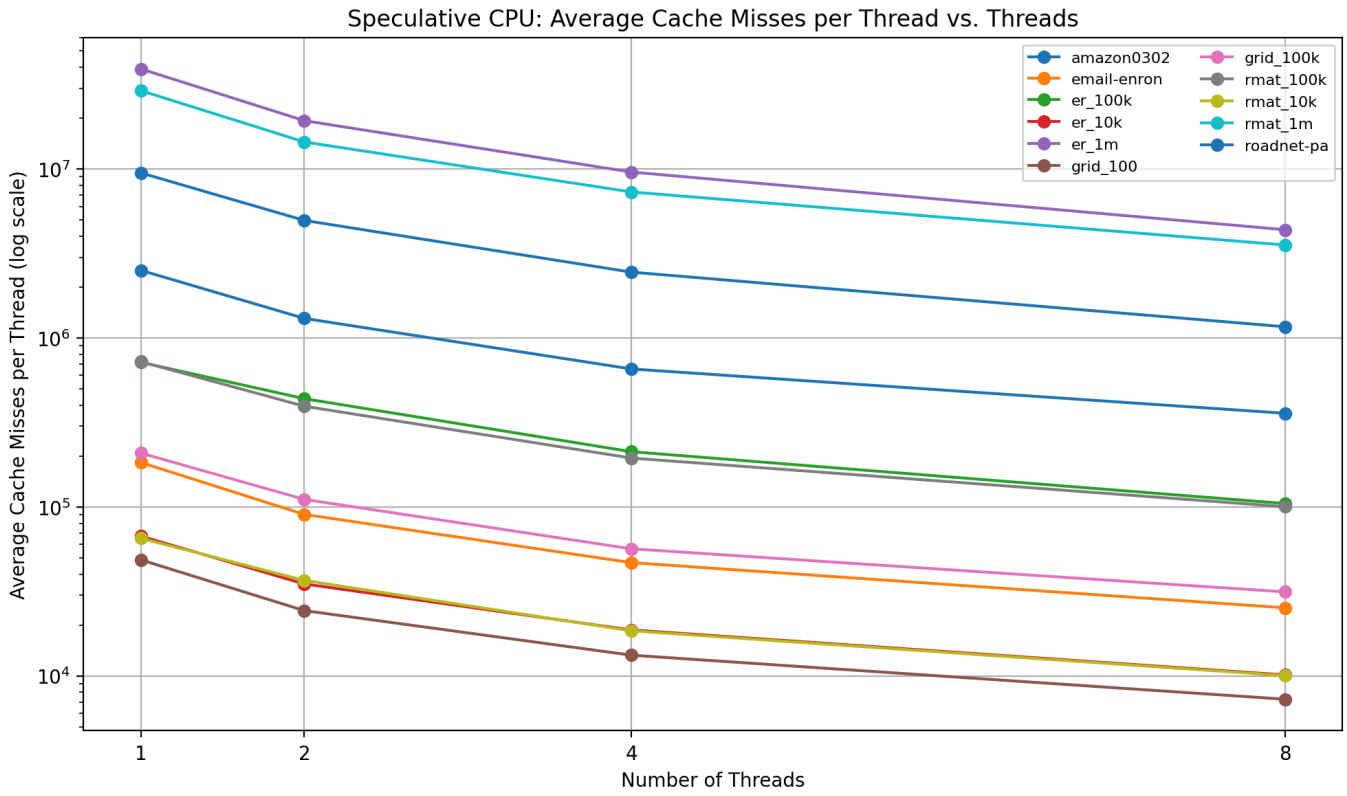



We can see that every curve is essentially flat. The total miss count varies by only around 15 % between 1 T and 8 T for every graph. This indicates that parallelizing the coloring phase does not substantially increase aggregate memory traffic. The algorithm is not causing large amounts of extra repeated accesses as threads are added. Since conflict rates are very low in the final implementation, only a small fraction of vertices are revisited in later rounds, so speculative recoloring does not significantly inflate the total memory footprint.

The largest effect is instead the vertical separation between graphs, which is mainly explained by graph size. A secondary effect is the graph structure. Larger graphs naturally produce more misses because the CSR arrays and colors[] array are larger and can't fit in the cache as well. As a result, the 1M-vertex graphs sit near the top of the figure, the 100K graphs form a middle band, and the 10K graphs sit near the bottom. Within each size class, it is affected by the graph structure. Power-law and irregular graphs tend to have more irregular accesses than highly regular grids.

Overall, we see that cache misses do not grow dramatically with thread count. This suggests that the imperfect CPU scaling we observe is not caused by bad memory accesses, but rather by a combination of serial initialization cost, synchronization overhead, and the inherent irregularity of graph traversal. The cache-miss results support the fact that the optimized speculative CPU implementation scales reasonably well, even when total speedup is limited by other factors.

### ii) Per-thread cache misses




This figure shows average cache misses per thread, computed by dividing total misses by thread count. Because total misses remain nearly flat, the per-thread miss count drops close to inversely with T. This is the expected pattern if work and memory traffic are partitioned cleanly across threads. As thread count increases, each thread is responsible for a smaller share of the graph and therefore incurs fewer misses and less false sharing.

We can draw the conclusion that adding threads reduces each thread's share of the memory traffic without causing a large increase in total cache misses. Larger graphs remain higher because their working sets exceed cache capacity, while smaller graphs, such as the 10K inputs, fall to around $10^4$ misses per thread at 8 threads. Overall, this supports the conclusion that the CPU parallelization is not breaking cache behavior and that the remaining variation is mostly due to graph size and topology.

**Problem Size Behavior**

Looking back at the results, problem size and graph structure both had a strong effect on behavior. On small graphs such as er_10k, rmat_10k, and grid_100, the useful compute phase is so short that OpenMP overhead and preprocessing cost dominate total runtime, even when compute-time speedup looks reasonable in isolation. On larger graphs, especially er_100k, amazon0302, er_1m, and rmat_1m, the speculative coloring phase is large enough to amortize thread-management overhead and achieve strong compute-time scaling. Structure matters as well. Regular grids have low conflict rates and predictable work, while power-law graphs such as rmat and email-enron are dominated by a small number of high-degree hubs, which makes preprocessing and load balance much more important.

**What Limited Speedup**

The main limit on total speedup was not a lack of parallelism in the coloring kernel itself, but rather the partial serial work in the init section. The speculative compute phase scales well, reaching about 5.5x–6.4x speedup at 8 threads on large graphs such as er_100k, amazon0302, er_1m, and rmat_1m, so the parallel coloring kernel clearly has enough available parallelism. However, total speedup is lower because degree statistics, hub identification, ordering, and especially sequential hub coloring remain outside the highly parallel core.

This is most visible on power-law workloads. On rmat_1m at 8 threads, the speculative CPU run spends about 36.2 ms in initialization and only 8.1 ms in compute, so initialization dominates end-to-end runtime even after the compute phase scales well. We intentionally kept hub coloring sequential because parallel hub coloring significantly hurt quality, as mentioned above.

Synchronization overhead is the second main limiter, especially on small and medium graphs. Each speculative round requires a barrier after tentative coloring and another after conflict detection and worklist updates, so the runtime is sensitive to stragglers when one thread receives a more expensive chunk of vertices. We reduced this cost with a persistent parallel region and adaptive scheduling, but it still becomes visible when per-round work is small.

Memory behavior also limits perfect scaling. The key operation, reading colors[col_indices[i]], is an irregular gather into a shared array, so performance is partly bounded by cache and memory latency rather than pure arithmetic throughput. Our perf stat measurements show that total cache misses remain nearly flat as thread count increases, which means the parallelization does not inflate aggregate memory traffic. For example, the large 1M-vertex graphs stay around 3e7–4e7 total misses across 1, 2, 4, 8 threads, while the per-thread miss count drops roughly as 1/T. This suggests that the remaining gap from ideal speedup comes mainly from serial preprocessing, barrier overhead, and irregular memory access, not from conflict rework or a breakdown in the parallel mapping itself.

**Was CPU the Right Target?**

For our final implementation, the CPU was the right machine target. The OpenMP versions achieved strong compute-time scaling on large graphs, preserved color quality on almost all realistic workloads, and delivered the best end-to-end performance in our final system. The irregular memory accesses in CSR graph traversal, the importance of hub-aware preprocessing, and the relatively low conflict rates after ordering all fit a multicore CPU well. These results suggest that speculative graph coloring, at least in the hub-aware form we implemented, is a strong match for shared-memory CPU parallelism.

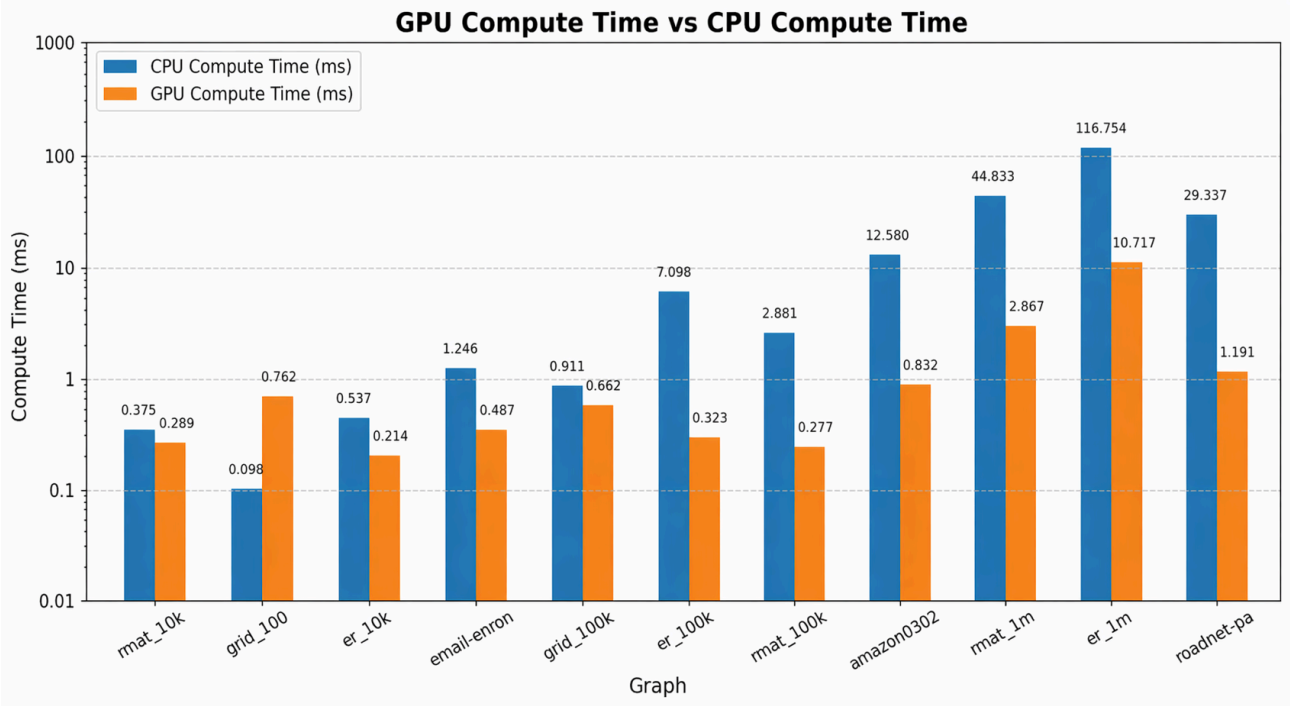
That said, this conclusion is specific to the graph sizes we evaluated, which ranged from roughly 10K to 1M vertices. For much larger graphs, a GPU could become a better option because its much higher thread-level parallelism and memory bandwidth would have more opportunity to amortize preprocessing and launch overhead. This will require adopting a more GPU-specific coloring design. Since the GPU portion of our project was already a stretch goal, we leave that as future work.

## GPU Results:

**The performance measurements and experimental setup were the same as the CPU version.**

**Compute Time vs CPU**

The GPU compute time is measured as the kernel time. It does not include preprocessing and work done on the host side. The CPU result is the 1-thread speculative parallel algorithm. The y-axis is log-scaled

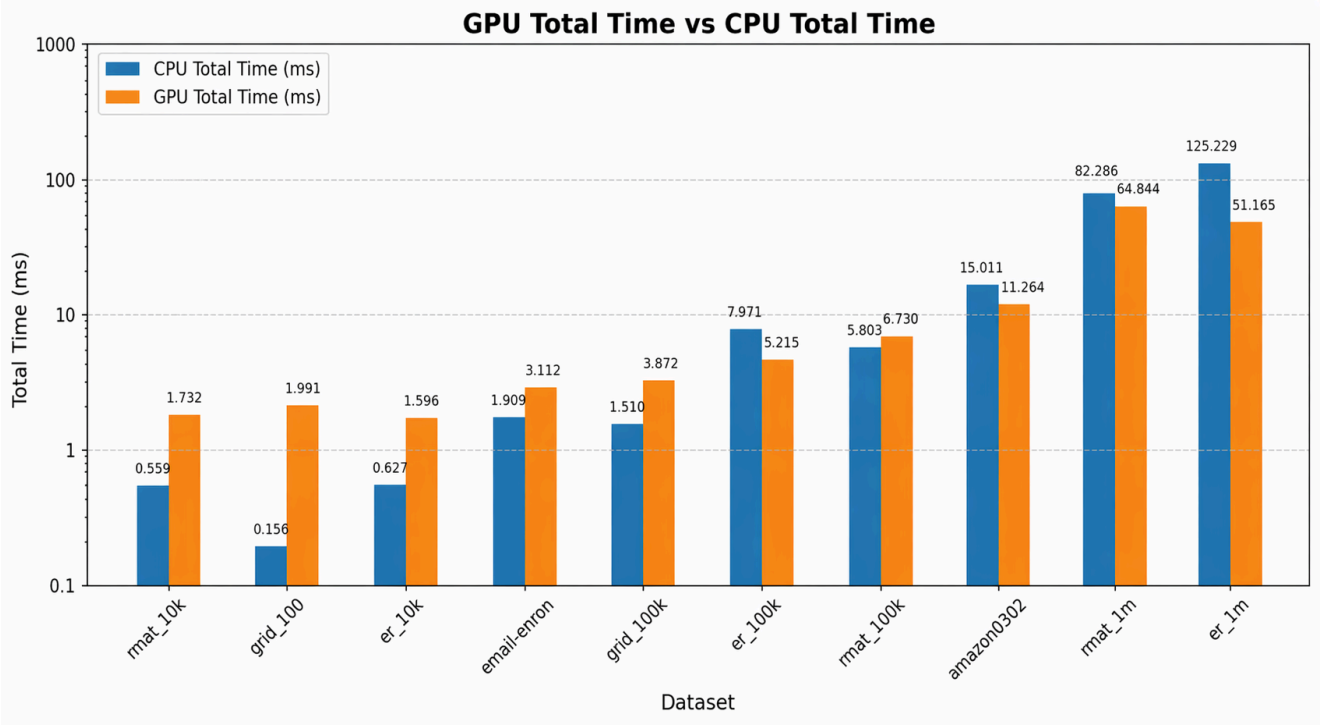



This figure shows that the GPU kernels are effective on medium and large graphs. The GPU compute time is lower than the 1-thread CPU speculative compute time on every graph except grid_100. The gains are especially large on er_100k, rmat_100k, amazon0302, rmat_1m, er_1m, and roadnet-pa, where the GPU reduces compute time from multi-millisecond or even 100+ ms CPU runtimes down to well under 11 ms.

The exception is the small regular grid case, where the GPU takes 0.762 ms versus only 0.098 ms on the CPU. This reflects the fact that graph coloring is irregular and conflict-driven, so GPU throughput is best used on larger workloads with enough parallel work to amortize each speculative round.

Overall, this figure supports the claim that the actual coloring kernels are fast on the GPU, and that the main weakness of the GPU implementation is not kernel throughput but the surrounding overhead outside the compute phase.

**Total Time vs. CPU**
This directly compares the total end-to-end time against the total CPU time.
The y-axis is log-scaled.



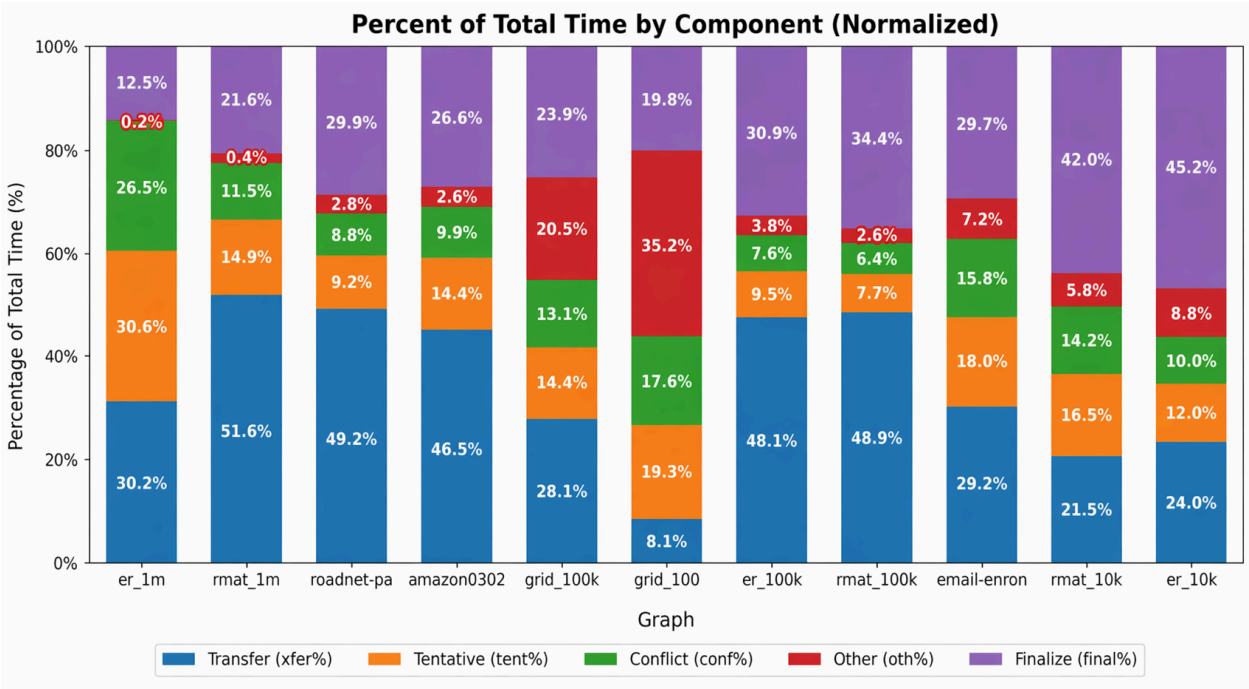

This figure shows that the GPU only wins in end-to-end total time on a subset of the larger graphs: er_100k, amazon0302, rmat_1m, and er_1m. On the smaller graphs, and even on some medium graphs like email-enron, the 1-thread CPU speculative version is still faster overall. The gap is especially large on small inputs such as grid_100, where GPU overhead dominates the actual coloring work.

The main reason is that the GPU's many threads only accelerate the kernel work, not the full pipeline. Total GPU time still includes CPU-side preprocessing, memory allocation, host-device transfers, and repeated host-device synchronization between speculative rounds. Since our GPU coloring algorithm is just an extension of our speculative CPU algorithm, this is expected.

In order to achieve much better results, we could try to implement a GPU-specific coloring algorithm. This current algorithm involves irregular CSR neighbor traversals, scattered memory accesses, and conflict-driven iteration, which create warp imbalance and many short synchronization-heavy rounds. So even though the GPU has far more hardware threads, much of that parallelism is spent waiting on memory, synchronization, or setup costs rather than doing useful coloring work.

**GPU Deeper Analysis and What Limited Speedup**

This figure shows the percentage of time spent in each phase.
Transfer T includes Host-Device & malloc time. Finalize includes Device-Host & cudaFree time.




These results explain why strong GPU kernel performance does not always translate into strong end-to-end runtime. On most graphs, only a modest fraction of total time is spent in the actual coloring kernels (tentative coloring and conflict resolution). A large share is instead consumed by transfer and finalization overhead. For example, on rmat_1m about 51.6% of the total time is transfer, and 21.6% is finalization, while the tentative and conflict kernels together account for only about 26.4%. A similar pattern appears on roadnet-pa, er_100k, and rmat_100k, where non-kernel overhead remains a large fraction of total GPU time. GPU performance is also hurt by graph irregularity, which causes warp imbalance and extra rounds on conflict-heavy graphs such as grid_100 and grid_100k.

This connects directly to the earlier GPU-vs-CPU plots. The compute-time figure showed that the GPU kernels themselves can outperform the 1-thread CPU on most medium and large graphs, but this breakdown shows why the total-time advantage is much smaller. In contrast to the CPU, where a larger fraction of time is spent in the actual coloring work, the GPU pays substantial overhead for data movement and host-side orchestration.
That is why the GPU is only competitive end-to-end on some larger graphs, even though its raw coloring kernels are often much faster. A more GPU-specific coloring design could likely reduce this gap by keeping more of the control flow and worklist management on the device and reducing repeated host-device synchronization.

**Was the GPU a good choice?**

For our final implementation, the GPU was not the best machine target for end-to-end performance. The speculative graph coloring algorithm we implemented is fundamentally a better match for CPU execution, since it relies on irregular memory accesses, repeated global synchronization between rounds, and host-side preprocessing, all of which reduce the benefit of massive GPU thread parallelism.

**Conclusion:**

In this project, we implemented and evaluated parallel graph coloring on both multicore CPUs and GPUs, using OpenMP and CUDA, respectively, on the GHC machines.

Our final CPU design combined speculative coloring, hub preprocessing, and degree-based ordering and delivered strong results. On large graphs, the optimized OpenMP implementation achieved 4.4× to 6.9× compute-time speedup at 8 threads, with most workloads falling in the 5.4×–6.5× range. At the same time, conflict rates remained extremely low, staying below 0.03% at 8 threads, which explains why most runs converged in only one or two rounds and why color quality remained close to the sequential baseline.

The main lesson from the CPU study is that the speculative coloring kernel itself had ample parallelism. The dominant limit on end-to-end scaling was the remaining serial fraction in initialization. In particular, hub preprocessing and sequential hub coloring were essential for preserving color quality and keeping conflicts low on power-law graphs, but they also introduced an Amdahl's law bottleneck. This was most visible on rmat_1m, where the 8-thread run spent about 36.2 ms in init versus 8.1 ms in compute, so preprocessing dominated total runtime even after the compute phase scaled well. More broadly, the remaining gap from ideal speedup came from a combination of serial preprocessing, barrier overhead, and irregular memory access through CSR neighbor gathers, rather than from excessive recoloring or a breakdown in the speculative algorithm itself.

Our GPU implementation was correct and achieved strong kernel-only throughput on some larger inputs, but it did not outperform the CPU end-to-end. The CUDA version suffered from exactly the kinds of overheads that irregular graph workloads expose on GPUs, such as preprocessing and transfers, neighbor-color reads through CSR were poorly coalesced, warp imbalance was common, and each speculative round required additional host-device synchronization. As a result, the GPU's much larger thread-level parallelism translated into only modest end-to-end gains, while the CPU remained the faster platform across the graph sizes we studied. Since GPU implementation was a stretch goal for this project, we did not implement a more optimized GPU coloring algorithm.

Overall, our results show that shared-memory multicore CPUs are a better match than GPUs for hub-aware speculative graph coloring at the 10K–1M vertex scale we evaluated. The CPU was the right machine target because CSR traversal, hub preprocessing, low conflict rates, and lightweight barrier synchronization all fit naturally within the OpenMP shared-memory model. At the same time, the project makes clear that future improvements will likely come not from further tuning the already-strong compute kernel, but from reducing or overlapping preprocessing costs and, on the GPU side, from adopting a more GPU-specific coloring algorithm rather than directly porting the CPU's speculative design.

**References:**

Gebremedhin, A. H., and Manne, F. "Scalable Parallel Graph Coloring Algorithms." Concurrency: Practice and Experience, 12(12): 1131–1146, 2000.

Jones, M. T., and Plassmann, P. E. "A Parallel Graph Coloring Heuristic." SIAM Journal on Scientific Computing, 14(3): 654–669, 1993.

Luby, M. "A Simple Parallel Algorithm for the Maximal Independent Set Problem." SIAM Journal on Computing, 15(4): 1036–1053, 1986.

Çatalyürek, Ü. V., Feo, J., Gebremedhin, A. H., Halappanavar, M., and Pothen, A. "Graph Coloring Algorithms for Multi-core and Massively Multithreaded Architectures." Parallel Computing, 38(10–11): 576–594, 2012.

Deveci, M., Boman, E. G., Devine, K. D., and Rajamanickam, S. "Parallel Graph Coloring for Manycore Architectures." In IEEE IPDPS, 2016.

Grosset, A. V. P., Zhu, P., Liu, S., Venkatasubramanian, S., and Hall, M. "Evaluating Graph Coloring on GPUs." In ACM PPoPP, 2011.

Welsh, D. J. A., and Powell, M. B. "An Upper Bound for the Chromatic Number of a Graph and Its Application to Timetabling Problems." The Computer Journal, 10(1): 85–86, 1967.

Brélaz, D. "New Methods to Color the Vertices of a Graph." Communications of the ACM, 22(4): 251–256, 1979.

Bulɥc, A., Fineman, J. T., Frigo, M., Gilbert, J. R., and Leiserson, C. E. "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks." In ACM SPAA, 2009.

Erdős, P., and Rényi, A. "On Random Graphs I." Publicationes Mathematicae, 6: 290–297, 1959.

Chakrabarti, D., Zhan, Y., and Faloutsos, C. "R-MAT: A Recursive Model for Graph Mining." In SIAM SDM, 2004.

The R-MAT generator used for power-law graphs (rmat_10k, rmat_100k, rmat_1m).
Leskovec, J., and Krevl, A. "SNAP Datasets: Stanford Large Network Dataset Collection." http://snap.stanford.edu/data, 2014.

Leskovec, J., Adamic, L. A., and Huberman, B. A. "The Dynamics of Viral Marketing." ACM Transactions on the Web, 1(1): 5, 2007.

Leskovec, J., Lang, K. J., Dasgupta, A., and Mahoney, M. W. "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters." Internet Mathematics, 6(1): 29–123, 2009.

Dagum, L., and Menon, R. "OpenMP: An Industry-Standard API for Shared-Memory Programming." IEEE Computational Science & Engineering, 5(1): 46–55, 1998.

**Distribution of Work:**

Zhanming (Jerry) Liang - 50% of work:

1. Sequential greedy baseline
   - color_sequential in src/coloring.cpp and LDF counting-sort ordering.
2. Conflict detection, worklist recoloring, correctness
   - Iterative detect-and-recolor loop, verify_coloring, tests/ suite, validation across all graph families
3. Hybrid speculative + Jones–Plassmann
   - color_hybrid, vertex_hash priority function, bulk-then-refine strategy, round-count tuning
4. GPU kernel optimization
   - 128-bit register bitmask palette, worklist compaction, block-size / occupancy tuning, sm_75 build flags
5. Analysis, report, website & poster
   - scripts/analyze_results.py (speedup, efficiency, conflict, cache-miss sections), milestone report, final report, index.html, poster

Yuehan (Harry) Hu - 50% of work:

1. Graph data structures & I/O
   - CSR representation (include/graph.h, src/graph.cpp), edge-list + METIS auto-detect loader, Timer utility (include/timer.h)
2. Parallel speculative coloring (Gebremedhin–Manne)
   - color_parallel_speculative, hub preprocessing, OpenMP scheduling/tuning, conflict-count instrumentation
3. CUDA GPU implementation
   - src/coloring_cuda.cu tentative-color + conflict-detect kernels, coloring_gpu_wrapper.cpp, Makefile, CUDA integration
4. Graph generators & datasets
   - scripts/gen_random_graph.py (Erdős–Rényi, R-MAT, grid), scripts/convert_snap.py, curated graphs/ collection (amazon0302, email-enron, roadnet-pa, synthetic 10K/100K/1M)
5. Benchmark harness & instrumentation
   - scripts/run_benchmarks.sh, scripts/run_perf_benchmarks.sh (perf-stat cache counters), CSV + timing plumbing in src/main.cpp, OMP_PROC_BIND / OMP_PLACES pinning