Accelerate Conflict-Based Search for Multi-Agent Pathfinding

15-418 Final Project Report

Erke Xia (exia)

December 10, 2025

Summary

I implemented and evaluated a GPU-accelerated version of Conflict-Based Search (CBS) for the Multi-Agent Pathfinding (MAPF) problem. My implementation targets the computationally expensive single-agent pathfinding (SAPF) subproblems using CUDA on an AWS EC2 `g5.2xlarge` instance. I built a serial CBS baseline in C++ and then added two GPU breadth-first search (BFS) kernels: one for single-agent SAPF and one for parallel initialization of all agents. On large grid maps with many agents, the GPU-accelerated versions reduce end-to-end CBS runtime greatly compared to the serial baseline, while preserving solution optimality.

1 Background

1.1 Problem Description

Single-agent pathfinding refers to the problem of finding a path between two vertices in a graph. The Multi-Agent Pathfinding (MAPF) problem is a generalization of this to $k > 1$ agents. The input is a graph and a set of agents; each agent has a unique start state and a unique goal state. The task is to find a path for each agent from its start to its goal such that:

- time is discrete, and each agent moves to an adjacent vertex or waits in place at each time step,
- no two agents occupy the same vertex at the same time (vertex conflicts), and
- no two agents traverse the same edge in opposite directions at the same time (edge conflicts).

A solution is a set of paths, one per agent, with no conflicts. Because the final solution is a collection of individual paths, MAPF algorithms often expose natural parallelism across agents and across local subproblems.

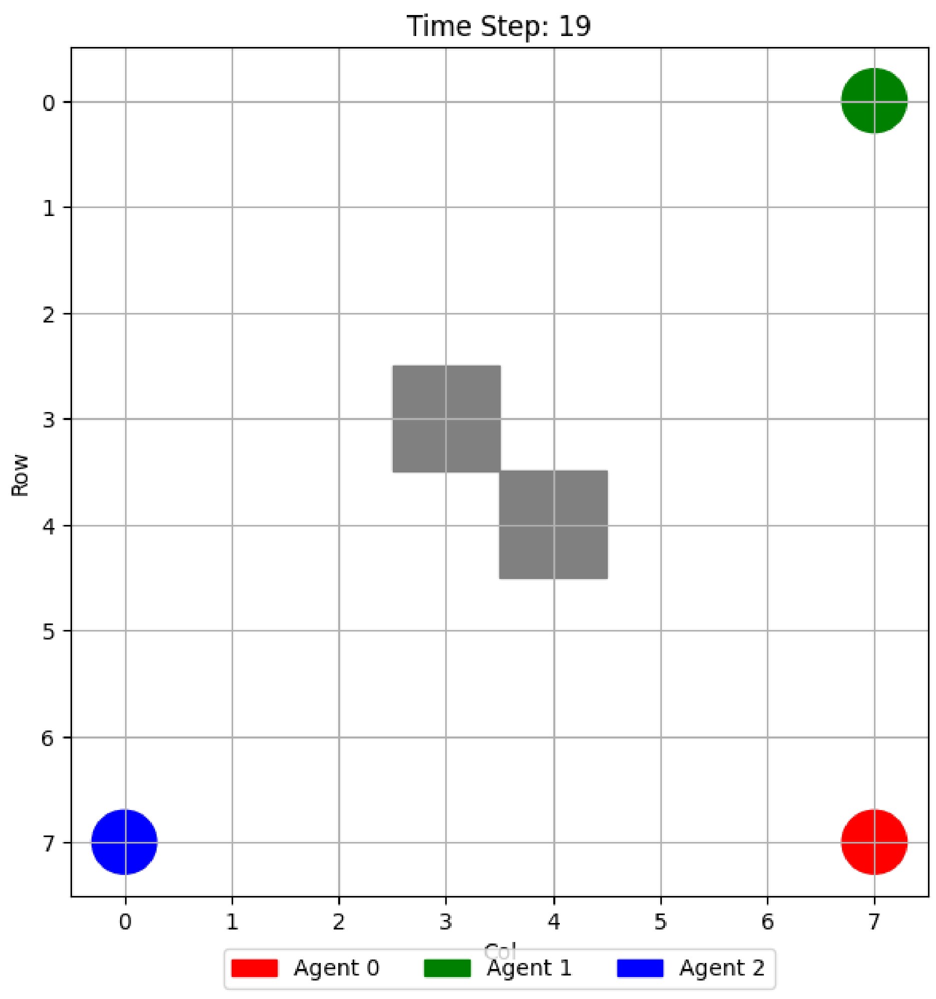

Figure 1: MAPF problem with three agents

1.2 Algorithm and Data Structures

1.2.1 CBS

Conflict-Based Search (CBS) is an optimal MAPF solver that separates the problem into a high-level search over conflicts and a low-level search for individual agents' shortest paths. In the MAPF setting, we have a graph, a set of agents, and discrete time. Each agent can move or wait at each step, and the goal is to avoid vertex and edge conflicts.

At the high level, CBS maintains a *constraint tree* (CT). Each CT node represents a candidate joint solution and stores:

- a set of constraints (forbidding an agent from being at a given vertex or traversing a given edge at a given time), and
- for every agent, a path that satisfies those constraints.

The root CT node has an empty constraint set. To create it, CBS solves k independent SAPF problems, one per agent, ignoring other agents. The cost of a CT node is typically the sum of individual path costs (SIC), and the high-level search expands nodes in order of increasing SIC cost (best-first search).

Once a CT node is generated, CBS checks the joint paths for conflicts. If no conflicts are found, the node encodes a valid joint solution and CBS terminates with this node as an optimal solution. If at least one conflict is found, CBS picks a conflict between agents a_i and a_j (at vertex v and time t, or an edge swap) and resolves it by branching:

- one child adds a constraint forbidding a_i from that vertex/edge at time t,
- the other child adds the symmetric constraint for a_j.

For each child, only the path of the agent whose constraints changed is replanned by a low-level solver; all other paths are copied from the parent node. If the low-level search fails to find a path consistent with the new constraints, that child node is discarded.

In standard CBS, the low-level search is usually A* on a time-augmented grid, using a priority queue and a heuristic (e.g., Manhattan distance). In my implementation, I also support BFS as the low-level search, which is simpler and still optimal on an unweighted grid.

1.2.2 Data structure

At the high level, CBS maintains a CT where each node contains:

- a map from agent ID to a list of constraints (time, location),
- a map from agent ID to a path (sequence of grid positions over time),
- an integer cost (sum of individual path costs).

The CT nodes are stored in a min-heap priority queue ordered by cost. The tree is expanded by best-first search.

At the low level, CBS solves Single-Agent Pathfinding (SAPF) with constraints. In the serial baseline, I use A* with:

- a binary heap for the open list (frontier),
- a hash set or map for visited states,
- a constraint set that we check before expanding a state.

In the GPU version, I use a BFS-based SAPF, with two frontier arrays for the current and next layers.

1.3 Parallelism and Dependencies

The overall algorithm exposes parallelism at two levels: (1) within each single-agent pathfinding (SAPF) search on the GPU, and (2) across different agents during initial path computation.

1.3.1 Sources of Parallelism

At the low level, the breadth-first search used to solve an SAPF problem can be parallelized. In each BFS layer, the active frontier is stored as an array; the nodes in this array can be processed in parallel, with threads expanding neighbors and enqueuing newly discovered cells into the next frontier.

In the initialization phase of CBS, all agents' shortest paths can be computed in parallel. At this root node, there are no constraints and the SAPF problems for different agents are independent, so we can map one agent to one GPU block.

1.3.2 Dependencies and Limitations

Within a single BFS search there is an inherent level-by-level dependency: all expansions at time step t must complete before the frontier at time step $t+1$ is known. This constrains us to a bulk-synchronous structure: parallel work is available within a level, but levels themselves are sequential. There are also intra-level dependencies on shared data structures: multiple threads may try to claim the same neighbor cell, which we resolve via atomic operations on the `visited` array and the frontier-size counters. These atomics serialize some updates and can reduce scalability when the frontier is dense.

At the CBS high level, different CT nodes depend on their parents' paths and constraints, so I keep CT expansion on the CPU and do not attempt to parallelize it. In my design, only the low-level SAPF solves and the initial per-agent searches run in parallel.

1.3.3 Memory Access Patterns

The grid is stored as a flattened 2D array in row-major order. When a thread expands a cell, it accesses its four neighbors, which are adjacent or nearby indices in this array; this gives some spatial locality. The search algorithm has to keep shared variables for visited nodes, parents, and frontier arrays in global memory. All threads need access to these arrays and writes to them use atomic operations.

1.3.4 Amenability to SIMD/GPU Execution

The workload is only partially well-matched to SIMD/GPU execution. The inner-loop neighbor expansion is regular and data-parallel: each active frontier node runs the same small computation over up to four neighbors, so warps benefit from SIMD-style execution there. On the other hand, control flow is branchy at a larger scale: different frontier nodes reach obstacles or boundaries at different times, and some threads discover the goal earlier than others. This causes warp divergence in the neighbor loops and in the termination logic. The use of atomics for visited nodes and frontier-size updates also introduces contention on popular cells.

2 Approach

2.1 Implementation Overview

Our implementation is organized around a baseline CPU Conflict-Based Search (CBS) solver and two GPU-accelerated breadth–first search (BFS) kernels that target the computationally expensive single-agent pathfinding (SAPF) subproblems.

On the CPU side we implement a standard optimal CBS solver. Each CBS node stores a set of per-agent constraints and a per-agent path set (the solution), and is ordered in a priority queue by sum-of-individual-costs (SIC). For a given set of constraints, the node's solution is constructed by running a low-level SAPF search for each agent. We detect conflicts by scanning all pairs of agents and returning the earliest vertex or edge conflict; if a conflict is found we branch on it by creating two child nodes, each adding one disallowing constraint to one of the conflicting agents.

The GPU side focuses on accelerating these SAPF calls. We first implemented a single-agent GPU BFS kernel that, given one start–goal pair and a static grid with obstacles, computes a shortest path using a level-synchronous frontier expansion. We then extended this design to a multi-agent kernel for initialization: one thread block per agent, all sharing a single grid representation but using disjoint slices of the global arrays for `parent`, `visited`, and frontiers. This kernel computes shortest paths for all agents in parallel with no constraints (root CBS node). For CBS nodes created later in the search, we use the BFS planner (with constraints).

2.2 Mapping Data Structures to Parallel Hardware

Grid representation The MAPF environment is a 2D grid with static obstacles. On the CPU this is stored as a `GridWorld` class with a width, height, and a `std::set<Position>` of blocked cells. On the GPU we flatten the grid into a 1D array of bytes in row-major order:

- index $i = r \cdot width + c$
- `obs[i] = 1` if cell (r, c) is an obstacle, `0` otherwise

We allocate this occupancy array once in device memory (`DeviceGrid`) and reuse it across all kernel launches, avoiding repeated host–device transfers of static map data.

Single-agent BFS kernel For a single SAPF query, we launch one thread block. Within the block we use global-memory arrays for:

- `parent[num_cells]`: predecessor cell ID for path reconstruction;
- `visited[num_cells]`: visited bit (stored as `unsigned int`);
- `frontier_curr[num_cells]`, `frontier_next[num_cells]`: worklists for current and next BFS layers.

Each BFS iteration operates on a single layer. We store the current frontier size and next frontier size in `__shared__` variables so they can be updated cooperatively by threads in the block. Each thread processes a strided subset of the frontier, expanding up to four neighbors per cell. We use atomic operations to:

- mark `visited[ncell]` via `atomicExch`, and
- append to `frontier_next` via `atomicAdd` on the size counter.

Path reconstruction is done on the host: after the kernel finishes, we copy the `parent` array back for that agent, walk backwards from the goal cell to the start cell, and reverse the sequence of positions.

Multi-agent initialization kernel For root CBS initialization, we map one agent to one thread block. We pack all agents' data into large device arrays of size `num_agents * num_cells`:

- `parent[a * num_cells + i]` is cell i's parent for agent a,
- similarly for `visited`, `frontier_curr`, and `frontier_next`.

The kernel takes arrays of start/goal coordinates (`start_r[a]`, `start_c[a]`, `goal_r[a]`, `goal_c[a]`), and each block computes its own start/goal cell IDs and then runs the same level-synchronous BFS as above, but restricted to its slice of each array. We also maintain one `found[a]` flag per agent to signal success or failure.

This design is a combination of task parallelism (blocks are independent across agents) and data parallelism (threads within a block expand different frontier cells).

CBS on CPU. The high-level CBS loop, constraint generation, and conflict detection remain on the CPU. A CBS node stores:

- a `Solution` (map from agent ID to path),
- a map from agent ID to per-agent constraint sets, and
- its SIC cost.

We use a min-heap priority queue over CBS nodes. When a node is popped, we run the low-level planner (GPU BFS), update its solution, and then check for conflicts. Because CT-node expansion and conflict detection involve moderate amounts of branching, pointer-heavy data structures, and relatively little computation per node, we chose to keep this code on the CPU rather than attempting to map it to GPU threads.

2.3 Algorithmic Changes and Design Decisions

My original goal was to replicate the GPU-accelerated conflict-based search (GACBS) algorithm, including its GPU-resident parallel A* (GATSA) low-level solver. After studying the paper in detail, I concluded that reproducing GATSA faithfully was beyond the scope of this project: their design uses per-block node arrays, multiple heaps, a parallel hash table with PHR duplicate detection, and prefix-sum compaction steps to maintain a sparse open list. Implementing and debugging all of these structures correctly would consume most of the project time.

Instead, I simplified the low-level algorithm:

- We replaced A* with unweighted BFS, which avoids priority queues and complicated open-list structures, at the cost of potentially exploring more nodes. BFS enables simpler parallelism given that all nodes in the frontier are at the same time step.
- I limited multi-block GPU usage to unconstrained SAPF problems (root-node initialization); constrained replanning under additional CT-node constraints is still handled by the GPU accelerated BFS with only one block each kernel.

On the CPU side, we also refactored the CBS implementation so that the high-level solver accepts a function pointer for the low-level planner (`PlannerFunc`). This allowed us to swap between the serial A* and GPU BFS without modifying the CBS logic itself, and made it easier to perform baseline comparisons.

2.4 Optimization Iterations and Debugging

We iterated through several implementation stages:

1. **Serial CBS baseline.** We first implemented and debugged a fully serial CBS solver with A* as the low-level planner. This gave us a correctness reference and a performance baseline on standard MAPF benchmarks.
2. **Single-agent GPU BFS.** Our first CUDA kernel targeted a single SAPF query using level-synchronous BFS. We experimented with different block sizes, and we added early-exit logic when the goal cell enters the frontier to avoid unnecessary levels.
3. **Multi-agent initialization kernel.** After the single-agent kernel worked, we extended it to process all agents in parallel. Getting the per-agent slicing of the large arrays correct and avoiding race conditions between blocks required careful indexing and debug prints. We also validated correctness by comparing GPU-generated paths against the serial A* results on small test instances.

Map	Serial-CBS (ms)	GPU-BFS (ms)	GPU-BFS+MultiInit (ms)
maze-32-32-2, 4 agents	104	221	219
Paris_1_256, 4 agents	4596	221	222
w_woundedcoast, 4 agents	28414	229	226
maze-32-32-2, 10 agents	1823	224	223
Paris_1_256, 10 agents	38955	221	224
w_woundedcoast, 10 agents	107708	242	229
maze-32-32-2, 19 agents	3728	226	219
Paris_1_256, 19 agents	53671	237	223
w_woundedcoast, 19 agents	193548	337	278

Table 1: End-to-end runtime comparison.

3 Results

3.1 Experimental Setup

All experiments were run on an AWS `g5.2xlarge` instance with 1× NVIDIA A10G GPU (24 GB HBM2), 8 vCPUs, and 32 GB of host memory. All Wall-clock runtimes are measured on the host using C++ `std::chrono` around the full CBS call, i.e., from the moment we start planning until a conflict-free set of paths is returned or the solver declares failure.

We reuse the same benchmark grid maps as GACBS:

- `maze-32-32-2` (small),
- `Paris_1_256` (medium),
- `w_woundedcoast` (large).

For each map we tested with MAPF instances with varying numbers of agents (*e.g.*, 4, 10, 19 agents). For every configuration (map, agent count) we record and report the mean runtime.

We compare three implementations:

1. **Serial-CBS**: baseline CBS with fully serial A* low-level planner on the CPU.
2. **GPU-BFS**: CBS using the GPU BFS kernel as the low-level planner, but launching one BFS kernel per agent (no cross-agent parallelization).
3. **GPU-BFS + MultiInit**: our final system, which uses the multi-agent initialization kernel (one block per agent) to compute all root-node paths in parallel, and then reuses the same low-level planner configuration as GPU-BFS for subsequent replans.

3.2 Overall Runtime Comparison

Table 1 reports CBS runtimes for all three implementations. The serial baseline scales poorly in both map size and agent count: on `maze-32-32-2` with 4 agents it finishes in only 104 ms, but it grows to 3.7 s for 19 agents on the same map and reaches 28.4 s and 107.7 s on `w_woundedcoast` with 4 and 10 agents, respectively. In contrast, both GPU variants are almost flat across all tested configurations, staying in the 220–340 ms range regardless of map or number of agents.

On very small problems the GPU overhead dominates. For `maze-32-32-2` with 4 agents, Serial-CBS (104 ms) is about 2× faster than the GPU versions (≈ 220 ms), mainly because the BFS frontiers remain small and there is not enough parallel work to amortize kernel launches and synchronization. However, as soon as either the map or the number of agents increases, the GPU versions very quickly overtake the CPU baseline. For example:

- On `maze-32-32-2` with 10 and 19 agents, GPU-BFS is about 8.1× and 16.5× faster than Serial-CBS, respectively (224 ms vs. 1.8 s, and 226 ms vs. 3.7 s).
- On the medium map `Paris_1_256`, speedups range from about 20.8× (4 agents: 4.6 s → 221 ms) to 226× (19 agents: 53.7 s → 237 ms).
- On the largest map `w_woundedcoast`, GPU-BFS achieves roughly 124× speedup for 4 agents (28.4 s → 229 ms) and about 445× speedup for 10 agents (107.7 s → 242 ms).

The multi-agent initialization kernel (GPU-BFS+MultiInit) performs very similarly to GPU-BFS in these experiments. Its runtimes differ by only a few milliseconds from GPU-BFS in most cases (e.g., 221 ms vs. 222 ms on `Paris_1_256` with 4 agents). This suggests that, for the tested instances, the cost of the root-node unconstrained searches is relatively small compared to the total CBS runtime, so parallelizing initialization alone does not dramatically change the end-to-end numbers. Nevertheless, it never hurts performance and is slightly faster than GPU-BFS in some of the larger settings (e.g., 278 ms vs. 337 ms on `w_woundedcoast` with 19 agents).

3.3 Effect of Grid Size

Fixing the number of agents and varying the map size highlights how sensitive the serial solver is to the size of the search space. With 4 agents, Serial-CBS grows from 104 ms on `maze-32-32-2` to 4.6 s on `Paris_1_256` and 28.4 s on `w_woundedcoast`. The GPU versions, however, remain around 220–230 ms on all three maps. Similar behavior appears for 10 agents: the serial runtime explodes from 1.8 s to 38.9 s to 107.7 s as the map enlarges, whereas GPU-BFS stays between 224 ms and 242 ms.

This confirms that the GPU kernels handle large grids well: as the map grows and BFS frontiers become wider, there is more parallel work per layer, so runtime stays almost constant, while the CPU baseline must expand many more nodes and slows dramatically.

3.4 Effect of Agent Count

For a fixed map, increasing the number of agents increases both the number of single-agent searches and the likelihood of conflicts that trigger replanning. On `maze-32-32-2`, Serial-CBS grows from

104 ms (4 agents) to 1.8 s (10 agents) and 3.7 s (19 agents). On `Paris_1_256`, it grows from 4.6 s to 38.9 s to 53.7 s over the same agent counts, and on `w_woundedcoast` it goes from 28.4 s to 107.7 s, to 193.5s for 19 agents.

In contrast, GPU-BFS is almost insensitive to agent count in this regime: times on `w_woundedcoast` are 229 ms, 242 ms, and 337 ms for 4, 10, and 19 agents, respectively. The GPU-BFS+MultiInit variant tracks these numbers closely or slightly improves on them. This reflects the per-agent BFS kernels exploit intra-search data parallelism effectively.

3.5 Correctness and Path Quality

For all benchmark instances where the serial solver finds a solution, the GPU-based implementations return paths with identical path lengths (sum-of-individual-costs) to the serial version. This is expected because BFS on an unweighted grid is optimal, and our CBS high-level logic is unchanged. In cases with no solution, all three implementations correctly report failure.

3.6 Summary of Findings

Overall, our experiments show that:

- GPU acceleration is not universally beneficial: on small, easy instances the overhead of launching kernels and moving data can outweigh the saved computation.
- For larger grids and higher agent counts, GPU-BFS can meaningfully reduce the cost of low-level searches compared to a purely serial A* baseline.
- Exploiting task parallelism across agents via the multi-agent initialization kernel yields additional gains, especially when many agents must be planned simultaneously.

References

[1] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding.

[2] M. Tang, R. Xin, C. Fang, et al. GPU-accelerated conflict-based search for multi-agent embodied intelligence.

Work Distribution and Credit

I worked on the project all on my own. I used some code from my former individual project.