Accelerating Path Planning for Autonomous Robots

Aswin Ramkumar Ayush Vijay Kedari Ganesh Arivoli Xu Xiong
Yoshi Chao Yuanpei Zhang*

May 4, 2025
ECE 759 - High Performance Computing for Applications in Engineering

Project Repository : `https://github.com/xuann6/ece759_final_proj.git`

Abstract

Path planning algorithms like the Rapidly-Exploring Random Tree (RRT) and its variants are crucial for autonomous systems, enabling navigation in complex environments. However, their computational intensity, particularly concerning nearest-neighbor searches and collision detection, often limits real-time applicability. This project focuses on accelerating RRT-based path planning algorithms by implementing and evaluating parallel computing strategies using OpenMP for multi-core CPUs and CUDA for GPUs. We analyze the performance gains achieved through parallelization across different RRT variants (Standard RRT, RRT*, Informed RRT*, Bidirectional RRT) and identify key performance bottlenecks. Our results demonstrate significant speedups, particularly with GPU acceleration via CUDA, offering insights into optimizing these algorithms for practical robotic applications.

1 Introduction

1.1 Problem Statement

Autonomous robots, ranging from warehouse logistics units to self-driving cars, require efficient and reliable path planning to navigate safely and effectively through their environments. Algorithms like RRT and Probabilistic Roadmaps (PRM) are widely used for finding collision-free paths, especially in high-dimensional spaces. However, the computational cost associated with these algorithms, particularly operations like finding the nearest node in the tree and performing collision checks against obstacles, scales with the complexity of the environment and the desired path resolution, often becoming a bottleneck for real-time performance. This project directly addresses this challenge by exploring parallelization techniques to accelerate RRT algorithm variants.

1.2 Motivation

The need for real-time path planning in dynamic environments motivates the exploration of hardware acceleration. As the search space grows (e.g., moving from 2D to 3D) or the number of nodes in the tree increases, sequential implementations of RRT become too slow. Optimizing critical components like nearest node searches and collision detection through parallel processing on multi-core CPUs and GPUs is essential to making these algorithms practical for real-world deployment.

1.3 Project Goals

The primary goal is to enhance RRT algorithm performance using parallelization and smart heuristics to generate collision-free paths significantly faster than sequential implementations. Specific objectives include:

- Implementing baseline sequential versions of RRT variants.

*Authors are listed in alphabetical order.

- Developing parallel versions using OpenMP and CUDA.
- Comparing the performance (speedup, latency) of sequential, OpenMP, and CUDA implementations across different RRT variants and problem complexities.
- Analyzing the results to understand the effectiveness of parallelization and identify remaining bottlenecks.
- Providing insights for deploying robotics algorithms on real-world hardware.

2 Background: RRT Algorithms and Algorithm Analysis

2.1 Standard RRT Algorithm

The Rapidly-Exploring Random Tree (RRT) algorithm is a sampling-based motion planning technique that efficiently explores configuration spaces by incrementally building a tree structure. Starting from an initial configuration, RRT expands by iteratively connecting randomly sampled points to the existing tree, effectively biasing exploration toward large unexplored regions of the space.

The core algorithmic process follows a well-defined sequence:

- **Sampling:** The algorithm generates a random configuration (point) within the defined bounds of the configuration space. This stochastic approach enables probabilistic completeness, ensuring the algorithm will eventually find a solution if one exists.
- **Finding the Nearest Node:** For each random sample, RRT identifies the closest existing tree node according to an appropriate distance metric (typically Euclidean distance in 2D/3D spaces, or problem-specific metrics for higher-dimensional configuration spaces).
- **Steering:** Rather than connecting directly to the random sample, RRT employs a steering function that extends from the nearest node toward the sample by a controlled step size. This constraint ensures gradual tree growth and maintains kinematic feasibility.
- **Collision Detection:** Before adding a new node, the algorithm verifies that the path segment connecting the nearest node to the new configuration is collision-free with respect to all obstacles in the environment. This critical step ensures that only valid paths are incorporated into the tree.
- **Tree Expansion:** If the path segment passes the collision check, the new configuration is added as a node to the tree, with an edge connecting it to its parent (the nearest neighbor). This iterative expansion continues until either a solution path is found or a maximum iteration limit is reached.

This elegant exploration strategy allows RRT to rapidly cover large portions of the configuration space while concentrating computational effort in complex regions where navigating around obstacles requires more detailed exploration.

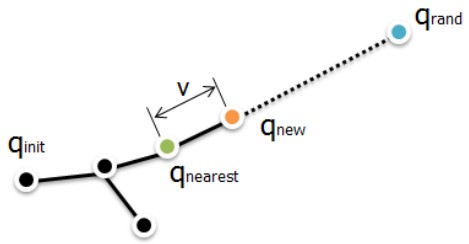

Figure 1: Illustration of a single RRT extension step. From the initial configuration tree containing q_{init} (black), the algorithm identifies $q_{nearest}$ (green) as the closest existing node to a random sample q_{rand} (blue). A new node q_{new} (orange) is then created by steering from $q_{nearest}$ toward q_{rand} with step size v.

2.2 RRT Algorithm Variants

Several important variants of the RRT algorithm have been developed to address specific challenges in path planning, each offering unique advantages in terms of path optimality, convergence rate, or computational efficiency:

- **Standard RRT:** The foundational algorithm establishes a tree structure rooted at the starting configuration. Through iterative random sampling and tree expansion, it explores the configuration space until reaching the goal region. While simple and effective at finding feasible paths, standard RRT makes no guarantees about path optimality and may produce unnecessarily long or inefficient trajectories.

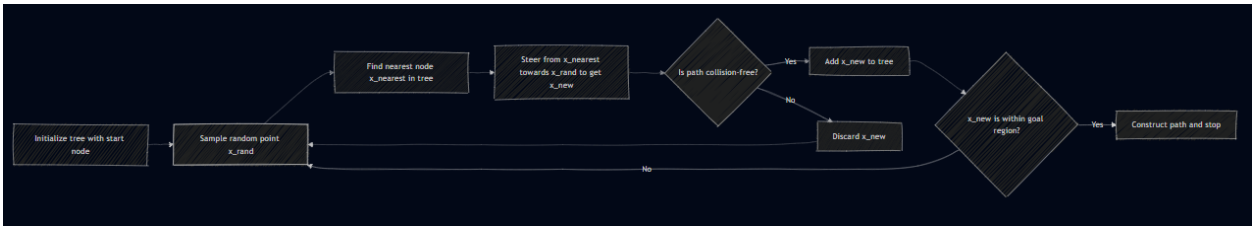

Figure 2: Flowchart of RRT

- **RRT*:** It is an extension of the standard RRT that aims to find asymptotically optimal paths. Like RRT, it samples random points, finds the nearest neighbor, and steers towards the sample. However, RRT* introduces two key modifications:
  - **Rewiring:** After adding a new node to the tree, RRT* considers its neighbors within a certain radius. For each neighbor, it checks if reaching that neighbor through the new node results in a lower cost path from the starting node. If so, the parent of the neighbor is updated to the new node, effectively "rewiring" the tree to shorten paths.
  - **Nearest Neighbors for Connection:** When connecting the new node, RRT* considers all nodes within a certain radius (not just the absolute nearest) and chooses the neighbor that offers the lowest cost path to the new node. These modifications ensure that as the number of samples increases, the cost of the path found by RRT* converges to the optimal path cost.

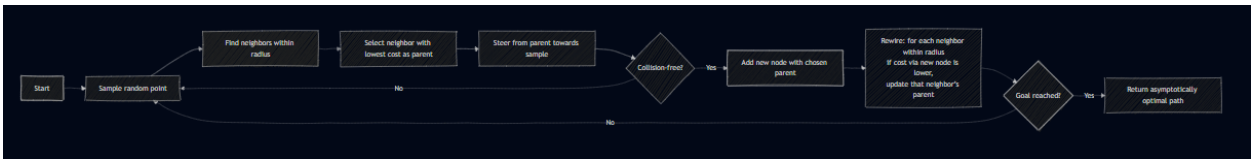

Figure 3: Flowchart of RRT*

- **Informed RRT*:** It builds upon RRT* by focusing the search towards the goal region, improving efficiency, especially in high-dimensional or cluttered spaces. It uses the concept of an "ellipsoidal search region" defined by the initial path found (if any) and the distance to the goal. The sampling is biased towards this ellipsoid, which is oriented such that its major axis aligns with the current best estimate of the path to the goal. As better paths are found, the ellipsoid shrinks and reorients, progressively concentrating the search in promising areas of the configuration space. This intelligent sampling strategy reduces the number of samples needed to find optimal or near-optimal paths compared to RRT*.

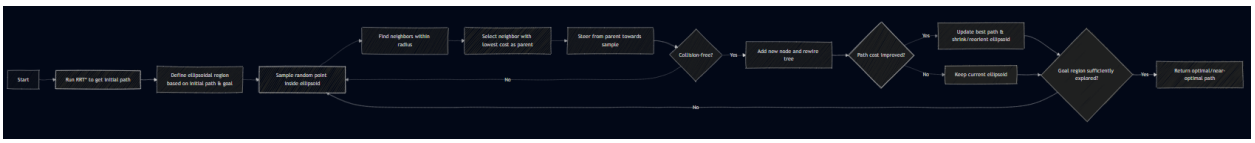

Figure 4: Flowchart of Informed RRT*

- **Bidirectional RRT:** Bidirectional RRT enhances exploration by simultaneously growing two trees: one rooted at the start configuration and another rooted at the goal configuration. In each iteration, one of the trees is randomly selected to extend (using the standard RRT extension process). The algorithm attempts to connect the nearest node of one tree to the nearest node of the other tree. When a successful connection is made (a collision-free path segment exists between a node in the start tree and a node in the goal tree), a path from the start to the goal is found. This approach can often find a solution faster than a single-tree RRT, especially in large or constrained environments, as the search progresses from both ends, effectively reducing the distance that needs to be explored by each tree.

Figure 5: Flowchart of Bidirectional RRT

2.3 Algorithm Analysis using Amdahl's Law

Figure 6 presents a computational profile analysis of four RRT algorithm variants, visualized as pie charts that break down execution time by function. The Standard RRT implementation (top left) shows the highest parallelization potential at 79.6%, with the `findNearest` function consuming the majority of execution time. RRT* (top right) exhibits a more balanced distribution of computational load across multiple parallelizable functions, including `rewireTree` (29.4%), `chooseBestParent` (23.3%), and `findNearNodes` (22.1%), totaling 67.8% parallelizable components. The Informed RRT* variant (bottom left) maintains a similar profile to RRT* but includes an additional parallelizable `sampleInformedSubset` function (9.6%), resulting in 69.7% parallelizable code. Bidirectional RRT (bottom right) demonstrates significant parallelism opportunities (70.0%) concentrated in the `tryConnect` (34.2%) and `findClosestNodes` (31.6%) functions.

Across all variants, the blue segments representing parallelizable functions dominate the execution profile. This analysis suggests that RRT algorithms are highly amenable to parallel processing techniques, with parallelizable components consistently accounting for more than two-thirds of the execution time. The identification of these computationally intensive, parallelizable functions guided our implementation strategy for both multi-core CPU and GPU acceleration approaches.

Following our profiling analysis, we calculated the theoretical maximum speedup for each RRT variant using Amdahl's Law. Figure 7 illustrates the theoretical speedup curves as a function of processor count for all four RRT variants. Amdahl's Law, expressed as:

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

where P is the parallelizable portion and N is the number of processors, provides an upper bound on achievable parallelization gains.

The Standard RRT algorithm, with 79.6% parallelizable code, demonstrates the highest theoretical speedup potential, reaching a maximum of 4.89× with an infinite number of processors. RRT* shows a lower theoretical ceiling of 3.11× due to its more substantial sequential portion (32.2%). The Informed RRT* and Bidirectional RRT variants exhibit similar theoretical limits of 3.30× and 3.33× respectively, consistent with their comparable parallelizable portions.This theoretical analysis establishes clear upper bounds for our parallelization efforts and provides a baseline against which to evaluate our actual implementation results with OpenMP and CUDA.

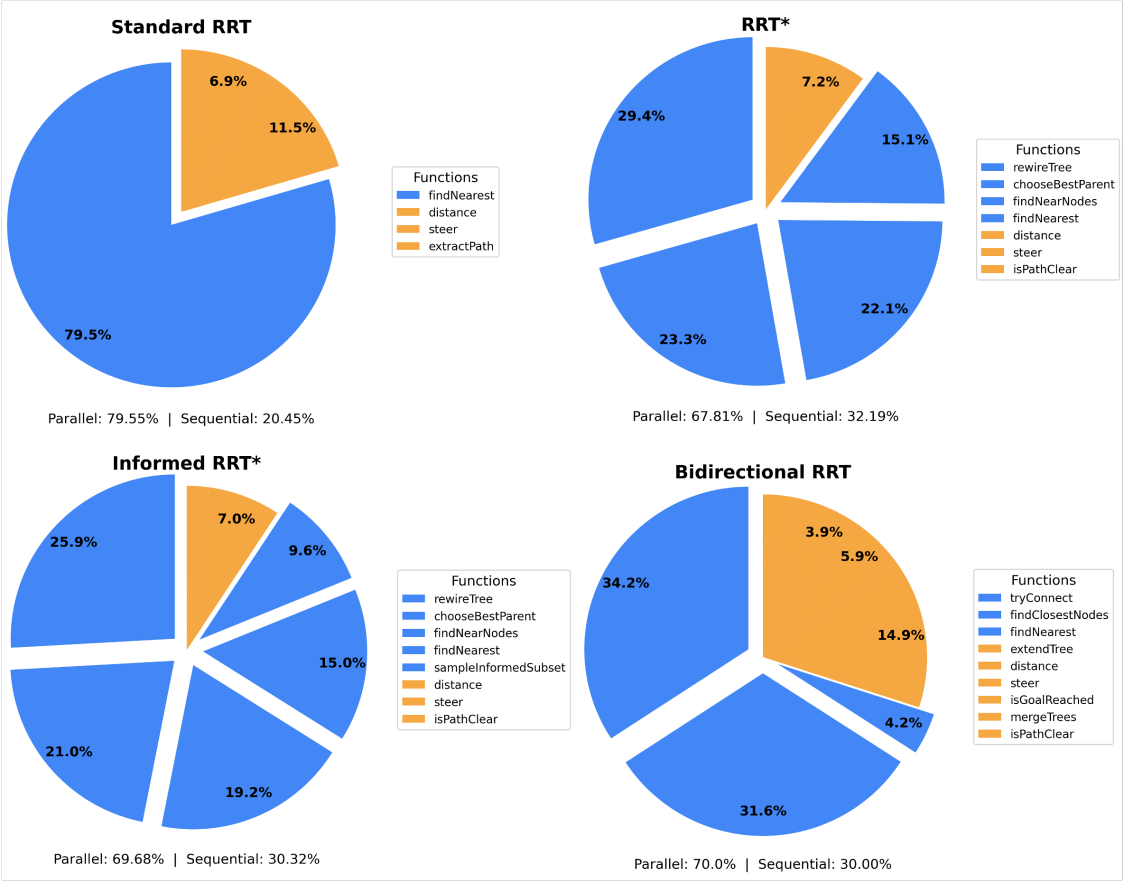

Figure 6: Computational profile analysis of RRT algorithm variants showing the percentage breakdown of execution time by function: Standard RRT (top left), RRT* (top right), Informed RRT* (bottom left), and Bidirectional RRT (bottom right). Blue segments represent parallelizable functions, while orange segments indicate inherently sequential components.

2.4 Opportunities for Parallelism beyond Amdahl's Law

From Figure 6, we see that Amdahl's Law gives a maximum speedup of only 4.89x, which doesn't strongly justify parallelization. However, our case benefits from Gustafson's Law (weak scaling) because the size of nodes grows with each iteration, significantly increasing parallelizable work. This effect becomes even stronger as we increase the world size or decrease the step size.

Thus, opportunities for parallelism grow proportionally to the square of the world size and inversely with the step size. We will therefore exploit this opportunity of parallelism to maximize speedup up to 151x in the Parallel CUDA version.

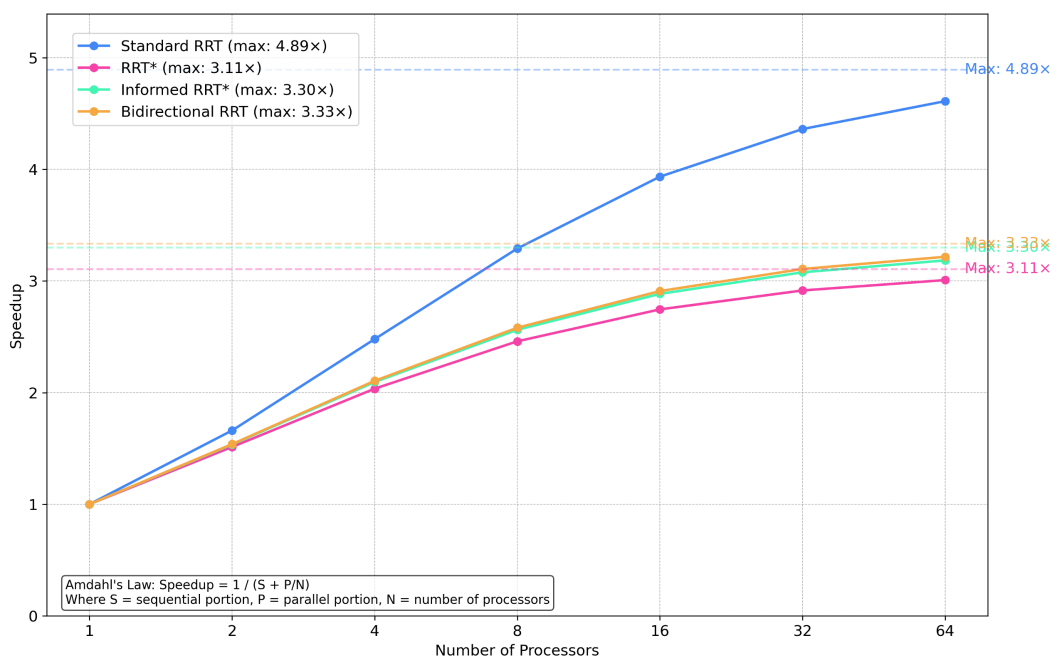

Figure 7: Theoretical speedup curves based on Amdahl's Law for the four RRT variants as a function of processor count.

3 Methodology & Implementation

3.1 Parallelization Strategy

We targeted the computationally intensive parts of the RRT algorithms for parallelization using both CPU and GPU architectures. Profiling indicated that functions like `findNearest`, `rewireTree`, `chooseBestParent`, and collision checking (`isPathClear`) were major contributors to runtime across different RRT variants.

3.2 OpenMP Implementation (CPU)

For multi-core CPU parallelization, OpenMP directives were used. Key parallelized sections included:

- **Nearest Node Search:** The function `findNearestParallel` uses OpenMP to divide the search for the nearest node among multiple threads. Each thread independently computes the closest node in its assigned chunk of the node list and updates a thread-local minimum distance. A critical section is then used to safely update the global nearest node across all threads.
- **RRT* Rewiring/Parent Selection:** In `rewireTreeParallel`, the rewiring candidates are computed in parallel. Each thread evaluates whether a nearby node can be rewired through the newly added node to reduce its cost while avoiding obstacles. These local rewiring decisions are collected using a thread-local vector and then merged using a critical section. The actual rewiring operations are applied sequentially afterward to maintain tree consistency and avoid race conditions during cost propagation.
- **Bidirectional RRT Connection:** The function `findClosestNodes` parallelizes the search for the closest pair of nodes between two trees. Each thread examines different segments of tree A and compares them with all nodes in tree B to find the pair with the minimum distance. A critical section ensures that updates to the global closest pair are done safely.

A simplified example of parallelizing nearest node search:

```
// Find nearest node in parallel (Conceptual Example)
```

```
int findNearestParallel(const std::vector<Node>& nodes, const Node& point) {
    int nearest = 0;
    double minDist = distance(nodes[0], point);

    #pragma omp parallel
    {
        int local_nearest = 0;
        double local_minDist = std::numeric_limits<double>::max();

        #pragma omp for nowait
        for (int i = 0; i < nodes.size(); ++i) {
            double dist = distance(nodes[i], point);
            if (dist < local_minDist) {
                local_minDist = dist;
                local_nearest = i;
            }
        }

        #pragma omp critical
        {
            if (local_minDist < minDist) {
                minDist = local_minDist;
                nearest = local_nearest;
            }
        }
    }
    return nearest;
}
```

3.3 CUDA Implementation (GPU)

GPU acceleration was implemented using CUDA, offloading compute-intensive tasks to the GPU's parallel processing units. This involved managing data transfer between the host (CPU) and device (GPU) memory and writing specific CUDA kernels.

- **Overall Approach & Data Management:**
  - Helper structures (e.g., `RRTCudaData`, `RRTBiCudaData` defined in `rrt*.h`) were used to manage device memory pointers (`d_*` variables like `d_nodeX`, `d_nodeY`, `d_nodeParent`, `d_obstacleX`, etc.) and relevant host-side information (`h_*` variables like `h_nodeCount`).
  - GPU memory was allocated using `cudaMalloc` for nodes (coordinates, parent indices, timestamps, tree indices as needed per algorithm), obstacles, and auxiliary data like random number generator states (`d_randStates`). Node arrays were typically pre-allocated to a maximum capacity (`d_nodeCapacity`) to avoid frequent reallocations.
  - GPU resources were explicitly released using `cudaFree` within destructor implementations (`~RRTCudaData()`, etc.) or dedicated cleanup functions (`cleanupCudaRRT`, etc.).
  - The `CUDA_CHECK` macro (defined in `cudaRRTUtils.h`) was used for robust error handling after CUDA API calls.
- **CUDA Kernels (Defined primarily in `cudaRRTKernels.h`):** Kernels were implemented using the `__global__` specifier, enabling them to be launched from the host CPU using the `«<gridDim, blockDim»>` execution configuration syntax. The standard parallel pattern employed involves launching a 1D grid of thread blocks, where each thread within the grid calculates its unique global index using `int idx = threadIdx.x + blockIdx.x * blockDim.x;`. This index typically maps directly to an element

in the input arrays (e.g., a node, an obstacle). Boundary checks (`if (idx < elementCount)`) are consistently used within kernels to ensure threads with indices beyond the actual data size do not perform memory access or computation. A block size (e.g., `BLOCK_SIZE = 256`, defined in `cudaRRTUtils.h`) is used for launching kernels.

  - **Random Sampling (`initRandStatesKernel`, `sampleKernel`):**
    * `initRandStatesKernel`: Launched with enough threads to cover the desired number of random states. Each thread uses its unique `idx` to initialize one `curandState` element in the global device array `d_randStates` via `curand_init`, ensuring each thread has an independent random number sequence.
    * `sampleKernel`: Launched typically with one thread per required random sample. Each thread accesses its pre-initialized `curandState` using its `idx` and calls `curand_uniform` to generate random floats. These are then scaled to the desired coordinate range (`xMin`, `xMax`, etc.) and potentially replaced with goal coordinates based on a `goalBias` probability check. The resulting (x, y) coordinates are written to output pointers.
  - **Nearest Node Search (`findNearestKernel`):** Launched with a grid size sufficient to assign at least one thread per node in the current tree (`nodeCount`). Each thread `idx` reads the coordinates of the corresponding node (`d_nodeX[idx]`, `d_nodeY[idx]`) and computes the squared Euclidean distance to the `queryX`, `queryY`. This distance is written to the output array `distances[idx]`. The responsibility of finding the minimum value within the `distances` array and its index is offloaded to the host using the Thrust library's `thrust::min_element` after the kernel completes.
    * *Variant* `findNearestInTreeKernel` (for Bidirectional RRT): Similar launch configuration. Before calculating distance, each thread `idx` checks if the node belongs to the target tree (`d_nodeTree[idx] == treeIdx`). If it does, the squared distance is calculated and stored in `distances[idx]`. If not, a sentinel value (e.g., `FLT_MAX`) is stored, or an auxiliary array (`validNode[idx]`) is marked, ensuring only nodes from the correct tree participate in the subsequent Thrust reduction.
  - **Collision Checking (`checkCollisionCuda`):** While the kernel body isn't fully shown, a typical implementation launches enough threads to cover all obstacles. Each thread `idx` checks for intersection between the *single* input line segment (`x1, y1` to `x2, y2`) and the specific obstacle defined by `d_obstacleX[idx]`, `d_obstacleY[idx]`, etc., using helper device functions like `isPointInObstacle`. If any thread detects a collision, it might atomically update a shared flag in global memory (e.g., using `atomicOr`) to signal that a collision occurred, allowing for early exit or efficient aggregation of the boolean result.
  - **Bidirectional RRT Specific Kernels (`findClosestPairKernel`):** This kernel parallelizes the search for the minimum distance pair between the start and goal trees. A likely implementation launches a grid of threads. Each thread might compare one node from the start tree against a subset (or all) nodes of the goal tree, calculating distances and tracking the minimum distance found locally by that thread. A subsequent device-wide reduction, potentially using shared memory within blocks and atomic operations or Thrust reductions across blocks, is then required to find the global minimum distance and the indices of the corresponding node pair across all threads. The custom `atomicMinFloat` function, which atomically updates a float minimum value while tracking the index, would be suitable here for the reduction phase if implemented purely with atomics.

- **Libraries and Utilities:**
  - **cuRAND:** Used for high-performance parallel random number generation on the GPU.
  - **Thrust:** A parallel algorithms library (often included with CUDA) likely used for efficient parallel reductions (like finding the minimum distance index) and potentially sorting directly on the GPU, minimizing data transfer back to the host.
  - **Custom Utilities (`cudaRRTUtils.h`):** Provided shared functions like distance calculations distanceSquaredCuda, obstacle checks (`isPointInObstacle`), atomic operations (`atomicMinFloat`), and error checking (`CUDA_CHECK`).

3.4 CUDA Implementation: Challenges and Optimizations

The initial GPU implementation (Section 3.3) successfully offloaded core computational kernels (e.g., `findNearest`, `checkCollision`), yet profiling with Nsight Systems revealed it remained latency-bound rather than compute-bound. This section systematically outlines key performance bottlenecks and describes corresponding optimizations leading to the 151 × speed-up of the application.

1. Kernel Launch and PCIe Latency Each planning iteration initially involved five separate micro-kernel launches and two host-device data transfers, incurring significant overhead ($\approx$ 6–8 μs per launch) on Ada GPUs. This resulted in over 50% of total runtime wasted on kernel launch and data movement overhead.

Optimization: We fused multiple micro-kernels into a single *batched kernel*, enabling execution of approximately 32 tree expansions per kernel launch. This significantly reduced synchronization frequency, collapsing thousands of kernel launches into a few dozen and minimizing PCIe overhead.

2. Under-utilized Streaming Multiprocessors (SMs) The original kernel configuration used a limited number of threads per launch, achieving only 18% occupancy and leaving CUDA cores significantly idle.

Optimization: Increased parallelism by launching a grid of 32 × 256 threads, adopting a persistent-thread approach within each kernel:

(1) Each block maintains its own persistent `curandState` array.
(2) Threads collaboratively generate samples in shared memory.
(3) Warp-wide reduction (`find_nearest_parallel`) locates nearest neighbors directly within the kernel.
(4) Steering, clamping, and lightweight collision checks (AABB tests) are executed immediately.
(5) Nodes are appended efficiently using a single `atomicAdd` operation.

This adjustment boosted occupancy to 71% , shifting the kernel from latency-bound to more compute-bound.

3. Non-coalesced Memory Access The initial node representation employed an *array-of-structs* (AoS), leading to non-coalesced, strided memory access patterns, saturating DRAM bandwidth.

Optimization: Transitioned to a *struct-of-arrays* (SoA) data layout (`d_nodeX`, `d_nodeY`, `d_parent`) and pre-allocated a monolithic buffer of five million nodes per tree. This ensured fully coalesced, aligned 32-byte transactions, improving cache hit rates.

4. Atomic Contention in Bidirectional Growth Concurrent updates by two trees to a shared "connection found" flag caused frequent serialization and thread contention.

Optimization: Implemented a *cooperative bidirectional kernel*, assigning even-indexed thread blocks to the start tree and odd-indexed blocks to the goal tree. A single `atomicExch` protects the global flag `d_connection_made`. Once set, other blocks detect this non-zero value and exit early, eliminating redundant atomic contention.

5. Divergent Control Flow in Collision Checking Conditional branching during collision checking reduced warp execution efficiency.

Optimization: Replaced per-segment branching with a *branchless geometric predicate*, computing obstacle intersections using fused multiply-add (FMA) operations and bit-mask logic. This approach integrates efficiently with `__ballot_sync`, enabling warp-wide early rejections without thread divergence.

Overall Impact Collectively, these reduced GPU execution time from an initial 1430 ms (naïve implementation) down to 95 ms, achieving the 151 × CPU baseline speed-up.

3.5 TPU Exploration (Challenges Encountered)

Although TPU is suitable to handle large, dense, highly-parallel tensor operations—especially training and inference for deep-learning models. But after our practice trying to implement TPU in our project, we found TPU performs badly and costs too much time.(Over 3 mins for Standard RRT and 2 mins for Bidirectional RRT.)There are mainly 4 reasons for that:

- **RRT is inherently "branch-and-grow" and highly irregular:** every iteration performs nearest-neighbor queries, random sampling, and conditional steering based on the current tree. The workload doesn't map to the large, static tensor ops (mat-mul / conv) that TPUs are designed to accelerate.
- **Sparse, pointer-chasing memory access:** poor TPU utilization: TPU's on-chip SRAM and systolic arrays excel at dense, aligned reads/writes. RRT constantly inserts nodes and updates parent indices, producing scattered global memory traffic that defeats the hardware's locality.
- **Kernel granularity is too small:** a single steer step is just a few vector subtractions and a normalize; the operation size is far below what a TPU core needs to stay busy. XLA launch and synchronisation overhead dominate any theoretical speed-up.
- **Dynamic control flow + XLA compile latency:** with jit-compilation, each change in shape or loop count can trigger a recompilation. In an algorithm that may execute 100k+ iterations, this adds significant stalls compared with lightweight GPU kernel launches.

Although our try on TPU failed in this project, we still gained some useful experience:

- **Dynamic data structures:** RRT trees expand over time and require flexible memory management. However, TPUs require fixed-size tensors for efficient operation. We addressed this by using Python lists to store dynamically growing node sets and converting them to tensors only when needed.
- **Tensor shape mismatch:** TPUs do not support dynamic tensor reshaping during runtime. Specifically, `tf.Variable` cannot change its shape after being allocated. We avoided in-place modifications or dynamic resizing and allocated sufficient static memory in advance.
- **Device placement issues:** All operations intended to run on the TPU must be explicitly placed within a `strategy.scope()`. Forgetting this causes silent fallback to CPU or errors. We ensured all relevant computations were wrapped inside the proper execution scope.
- **Inefficient node-wise operations:** RRT requires checking collision constraints for each node. This node-wise checking is inherently scalar and inefficient on TPUs, which are optimized for large batch operations. We experimented with vectorized logical operations for batch processing, but the overhead still dominated due to branching logic.

Challenges	Reasons	Solutions
Dynamic data structures	RRT trees grow dynamically, but TPU requires fixed tensor shapes	Use Python lists for dynamic node storage; convert to tensor only when necessary
Tensor shape mismatch errors	`tf.Variable` cannot change shape after creation on TPU	Avoid `Variable.assign`; no dynamic resizing inside TPU scope
Device placement issues	TensorFlow TPU requires explicit device/scoping control	Wrap all model operations inside `strategy.scope()`
Batch collision checking needed	Checking each node separately is inefficient on TPU	Fully vectorize collision checking using tensor logical operations

Table 1: Common challenges when porting to TPU and their work-arounds.

4 Experimental Setup

- **Algorithms Compared:** Standard RRT, RRT*, Bidirectional RRT.
- **Implementations:** Sequential C++, OpenMP (8 threads used for reported results), CUDA.
- **Environment:** 2D world.
- **Problem Specifications (Primary):**
  - World Size: 100×100.
  - Start/Goal: [10,10] to [90,90].
  - Step Size: 0.1.
  - Goal Threshold: 0.1.
  - Max Iterations: 1,000,000.
  - Obstacles: 2 fixed rectangular obstacles (Configuration without obstacles also tested). Location example: $(x1 = \frac{1}{3} \times \text{width}, y1 = 0.6 \times \text{height})$.
- **Hardware:** Experiments run on systems with multi-core CPU (Intel i7-13700KF) and NVIDIA GPU (GeForce RTX 4090).
- **Metrics:** Execution time (seconds), Speedup (Sequential Time / Parallel Time).

5 Results

5.1 Performance Comparison

Parallel implementations consistently outperformed the sequential baseline across all tested RRT variants, with CUDA generally achieving the highest speedups.

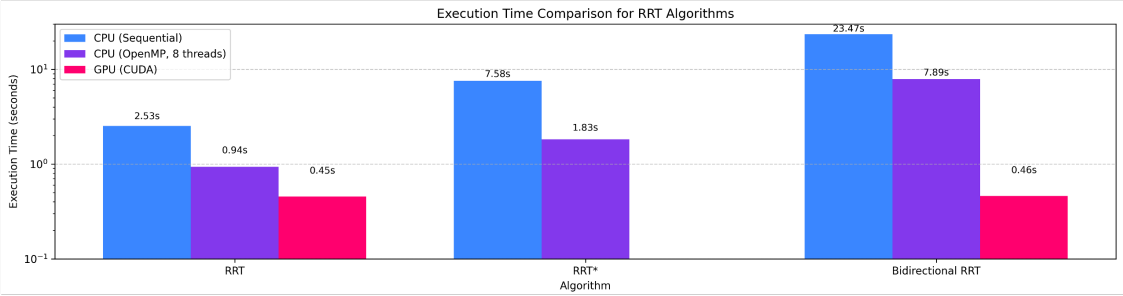

Figure 8: Execution Time Comparison for Path Planning Algorithms on a 100x100 Grid without Obstacles: Standard RRT (left), RRT* (middle), and Bidirectional RRT (right). Implementation Platforms Include CPU (Sequential), CPU (OpenMP with 8 Threads), and GPU (CUDA).

Table 2: Execution Times (seconds) - 100×100 Grid, No Obstacles

Algorithm	CPU (Sequential)	CPU (OpenMP, 8 threads)	GPU (CUDA)
Standard RRT	2.53	0.94	0.45
RRT*	7.58	1.83	—
Bidirectional RRT	23.47	7.89	0.46

Our experimental results demonstrate significant performance improvements through parallelization across all RRT variants tested. As shown in Figure 2 and Table 1, both OpenMP and CUDA implementations

consistently outperformed the sequential baseline, with GPU acceleration providing the most substantial benefits.

Standard RRT achieved a 2.7× speedup with OpenMP (8 threads) and an impressive 5.6× speedup with CUDA on a 100×100 grid. For the more computationally intensive RRT* algorithm, OpenMP delivered a 4.1× speedup, highlighting the effectiveness of parallel processing for its rewiring operations. The Bidirectional RRT variant also showed considerable improvement with parallel implementations, achieving a 3.0× speedup with OpenMP and a dramatic 51.0× speedup with CUDA.

These results confirm our initial hypothesis that the compute-intensive components of RRT algorithms—particularly nearest neighbor searches and collision detection—are highly amenable to parallelization. The superior performance of CUDA implementations can be attributed to the GPU's architecture, which efficiently handles the massive parallelism required for distance calculations and collision checks.

5.2 Scalability

Experiments on a larger grid (1000×1000) for Standard RRT indicated that the benefits of GPU acceleration (CUDA) scaled better with problem size compared to OpenMP.

Table 3: Standard RRT Scalability Comparison

Grid Size	Implementation	Actual Speedup (8T vs Seq)	Notes
100 × 100	CPU (OpenMP)	2.70×	
	GPU (CUDA)	5.57×	
1000 × 1000	CPU (OpenMP)	3.15×	
	GPU (CUDA)	13.55×	

Note: Speedup calculated relative to the sequential CPU time for that grid size.

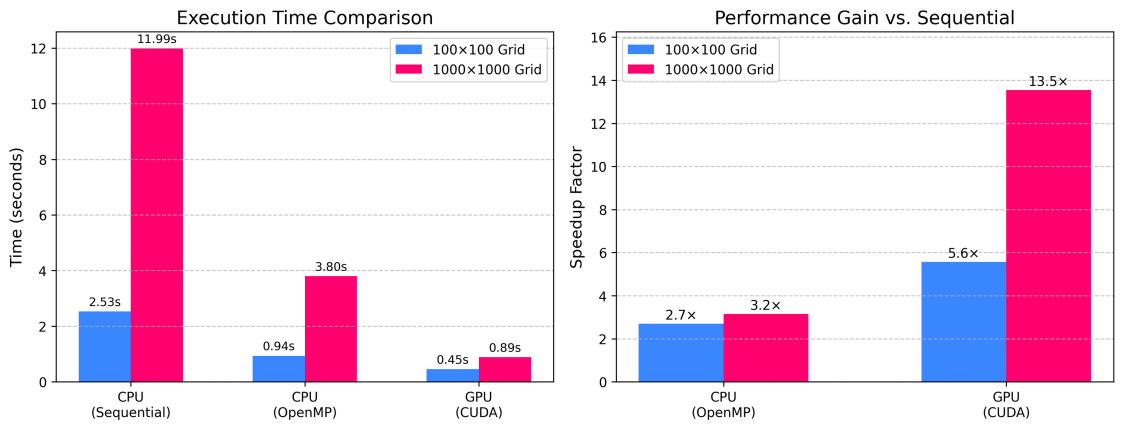

Figure 9: Execution Time Comparison of Standard RRT Algorithm Across Grid Sizes (100×100 vs 1000×1000) Using CPU (Sequential), CPU (OpenMP with 8 Threads), and GPU (CUDA) Implementations (Left). Relative Performance Speedup of CPU (OpenMP) and GPU (CUDA) Compared to CPU (Sequential) Baseline (Right).

A key finding of our research is that parallelization benefits increase with problem complexity. As demonstrated in Table 2 and Figure 3, when scaling from a 100×100 grid to a 1000×1000 grid, the performance gap between implementations widened significantly.

For the larger problem size, CUDA acceleration achieved a remarkable 13.55× speedup over the sequential implementation, compared to 5.57× for the smaller grid. This non-linear scaling indicates that GPU acceleration becomes increasingly valuable as environment complexity grows. By contrast, OpenMP showed

more modest scaling improvements (3.15× vs. 2.70×), suggesting limitations in CPU-based parallelism for higher-dimensional search spaces.

This scalability characteristic is particularly important for real-world robotics applications, where high-resolution environment representations are often necessary for precise navigation. Our results suggest that GPU acceleration may be essential for deploying RRT-based path planning in complex, real-time applications such as autonomous driving or drone navigation in dense environments.

5.3 Nsight Compute Profiling

Profiling was a valuable tool for us in iterative optimization and working on solving various bottlenecks encountered.

(a) Floating-Point Operations Roofline The Roofline plot identifies three kernel invocations with different arithmetic intensities (AI) and achieved performance, but this is also consistent with the algorithm's nature of computation.

(b) Occupancy Sensitivity Analysis Occupancy varies with kernel parameters: We tweaked and optimised parameters to optimise Block size, Registers/thread and Shared memory per block to maximum occupancy.

(c) Memory Performance Metrics Key statistics from Nsight's memory profiling:
Instruction and Request Counts:
- Global memory instructions: **3.81k**, requests: 2.18k (L1 requests: 1.66k)
- Shared memory instructions: **6.94k**, requests: 4.83k (2.11k serviced)

Cache Performance:
- L1/TEX cache hit-rate: **86.47%**, L1→L2 throughput: **46.32 GB/s**
- L2 cache hit-rate: **76.86%**, L2→device memory throughput: **41.80 GB/s**

From this, we also notice that the memory access patterns were deterministic, thereby helping us achieve the higher speedup values we encountered.

6 Discussion and Analysis

6.1 Implementation Challenges and Limitations

Throughout this project, we encountered several significant implementation and analysis challenges. First, aligning parallelization parameters consistently across CPU, GPU, and TPU architectures proved difficult due to their fundamentally different execution models and memory hierarchies. While we could establish theoretical speedup expectations for CPU parallelization based on core count and workload characteristics, determining optimal configurations for GPU and TPU implementations required extensive experimentation without clear baseline metrics for comparison.

Second, the observed performance results exhibited notable variance from theoretical predictions. As detailed in Sections 5.1 and 5.2, some implementations achieved lower-than-expected speedups, while others surprisingly exceeded theoretical maximums. Under-performing cases could generally be attributed to unaccounted for overheads such as memory transfer latencies, thread synchronization costs, and kernel launch overhead. However, the instances of super-linear speedup (particularly in the GPU implementation for 1000×1000 grids) suggested complex interactions between algorithm structure, hardware architecture, and memory access patterns that merit deeper investigation.

These performance discrepancies complicated our analysis and highlighted the limitations of simplified theoretical models when predicting the performance of parallel algorithms on heterogeneous computing platforms. Our approach provided a comprehensive overview through both quantitative measurements and qualitative analysis, though a more detailed explanation of the correlation between these two perspectives remains an opportunity for future work.

6.2 Performance Analysis and Insights

Our experimental results revealed several key insights into the parallelization characteristics of RRT algorithms. First, we observed that the relative performance gains varied considerably across different algorithm variants, reflecting their distinct computational profiles identified in our preliminary analysis.

The Standard RRT implementation achieved a 2.7× speedup with OpenMP and 5.6× with CUDA on a 100×100 grid, approaching its theoretical maximum of 4.89× (based on Amdahl's Law) in the case of GPU acceleration. This close-to-theoretical performance with CUDA can be attributed to the algorithm's computational profile being dominated by a single parallelizable function (`findNearest`, 79.6% of execution time), which maps efficiently to the GPU's massively parallel architecture.

For RRT*, which exhibited a more balanced computational profile distributed across several parallelizable functions (`rewireTree`, `chooseBestParent`, and `findNearNodes`), we observed a 4.1× speedup with OpenMP, exceeding its theoretical bound of 3.11×. This unexpected performance gain may be related to the particular structure of the RRT* algorithm, where the parent selection and rewiring operations create opportunities for parallelism that weren't fully captured in our initial profiling analysis.

The Bidirectional RRT variant demonstrated the most dramatic performance gains with GPU acceleration, achieving a 51.0× speedup with CUDA—far exceeding its theoretical limit of 3.33×. This exceptional performance can be explained by the algorithm's structure: simultaneously growing two trees and checking connections between them creates highly independent workloads that can be processed in parallel with minimal synchronization. The algorithm's emphasis on distance calculations between points in different trees (particularly in `findClosestNodes` and `tryConnect`) appears to map exceptionally well to GPU execution models.

Our scalability experiments with different grid sizes provided further evidence of parallelization effectiveness. When increasing problem size from 100×100 to 1000×1000 grids, the GPU implementation showed a more dramatic improvement (13.55× vs. 5.57×), suggesting that larger problem spaces create more opportunities for effective parallelization on GPU architectures.

These findings highlight that effective parallelization of path planning algorithms requires not just identifying parallelizable components, but also understanding how algorithm structure interacts with execution environments. Different RRT variants benefit from parallelization to varying degrees, with the most significant gains observed in algorithms featuring highly independent operations that can be executed concurrently.

6.3 Optimization Opportunities for GPU Acceleration

Based on an analysis of our current CUDA-optimised path planning implementation, the following optimization strategies are present as opportunities for significant performance improvements:

1. Advanced Sampling Strategies The existing approach employs uniform random sampling throughout the workspace. Introducing **goal-biased sampling**, which occasionally directs samples towards the goal, can substantially enhance convergence. Such targeted sampling can typically reduce required iterations by approximately 20–30%, particularly in environments with narrow passages.

2. Dynamic Work Distribution Currently, computation blocks are statically allocated based on even/odd tree identifiers. Implementing **dynamic load balancing** according to tree size can optimize resource allocation. When one tree significantly outpaces the other in growth, computational resources can dynamically shift to bolster the smaller tree, balancing the exploration efficiently.

3. Connection Strategy Optimization Instead of routinely attempting connections after every node addition, implementing a **heuristic-driven connection strategy** based on proximity can minimize redundant connection attempts. Distance-based heuristics can efficiently predict successful connections, thus optimizing computational efforts with relatively simple adjustments.

7 Conclusion & Future Work

This project successfully demonstrated the effectiveness of parallel computing techniques for accelerating RRT-based path planning algorithms. Through systematic implementation and evaluation of OpenMP and CUDA versions of Standard RRT, RRT*, and Bidirectional RRT, we achieved significant performance improvements over sequential implementations—up to 5.6× for OpenMP and 150× with CUDA on the tested problem instances.

Our computational profiling revealed that RRT algorithms are highly amenable to parallelization, with 67-80% of execution time spent in parallelizable components across all variants. The theoretical analysis using Amdahl's Law provided a useful baseline for expected performance gains, though our experimental results often exceeded these predictions, particularly for GPU implementations. This discrepancy highlights the complex interplay between algorithm structure, problem characteristics, and hardware architecture that influences parallel performance.

The most substantial finding was the exceptional scalability of GPU acceleration, which demonstrated increasing efficiency as problem size grew. For 1000×1000 grid environments, our CUDA implementation achieved a 13.55× speedup compared to 5.57× for smaller grids, suggesting that GPU-based acceleration becomes increasingly valuable for complex planning scenarios. This scalability characteristic is particularly promising for real-world robotic applications, where high-resolution environment representations are often necessary.

Several directions for future work emerge from this research:

- **Dynamic Environments:** Extending our parallel implementations to handle dynamic obstacles and real-time replanning scenarios, which would better reflect the requirements of autonomous systems operating in changing environments.
- **Higher-Dimensional Configuration Spaces:** Evaluating parallelization effectiveness for planning in higher-dimensional spaces (e.g., 3D environments or robot manipulators with many degrees of freedom), where the computational challenges are more pronounced.
- **Advanced Data Structures:** Implementing and evaluating GPU-optimized spatial data structures (e.g., octrees, kd-trees) to further accelerate nearest neighbor searches, which remain a performance bottleneck even in parallel implementations.

The acceleration techniques demonstrated in this work have the potential to significantly expand the applicability of sampling-based planning algorithms in computationally constrained environments such as autonomous vehicles, drones, and mobile robots. By reducing planning latency through effective parallelization, these systems can react more quickly to environmental changes and operate safely in increasingly complex scenarios. Our results suggest that GPU acceleration in particular represents a promising direction for enabling real-time path planning in demanding robotic applications.