PDC Term Project

Analysis of A* Search Algorithm

M. Ibrahim Ayoubi | Kisa Fatima

TABLE OF CONTENTS

I. Project Overview

This project analyzes the performance of the A* search algorithm across different computing architectures. It focuses on four main implementations: Sequential, Parallel Random Access Machine (PRAM), OpenMP, and GPU-based using Compute Unified Device Architecture (CUDA).

II. The A* Search Algorithm

A* Search is a pathfinding and graph traversal algorithm, it is known for its efficiency and accuracy in finding the shortest path between two points. It combines the strengths of Dijkstra's Algorithm and Greedy Best-First Search by using a heuristic to guide the search.

The algorithm works by evaluating nodes using the formula:

$$f(n) = g(n) + h(n)$$

- **g(n)** is the cost from the start node to the current node
- **h(n)** is the estimated cost/distance from the current node to the goal (heuristic)
- **f(n)** is the total estimated cost of the cheapest solution through that node

In this project, we use the **Euclidean heuristic**, which estimates the distance between two points using the straight-line distance:

$$h(n) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Pseudocode

```
// A* Search Algorithm
1.  Initialize the open list
2.  Initialize the closed list
    put the starting node on the open
    list (you can leave its f at zero)
3.  while the open list is not empty
    a) find the node with the least f on
       the open list, call it "q"
    b) pop q off the open list

    c) generate q's successors and set their
       parents to q

    d) for each successor
        i) if successor is the goal, stop search

        ii) else, compute both g and h for successor
          successor.g = q.g + distance between
                               successor and q
          successor.h = distance from goal to
          successor (Euclidean)

          successor.f = successor.g + successor.h
        iii) if a node with the same position as
            successor is in the OPEN list which has a
           lower f than successor, skip this successor
        iV) if a node with the same position as
            successor  is in the CLOSED list which has
            a lower f than successor, skip this successor
            otherwise, add  the node to the open list
     end (for loop)

    e) push q on the closed list
    end (while loop)
```

Step by Step Explaination

1. **Initialize the open and closed lists:**

   Start by putting the **starting node** into the **open list** (places to explore). The closed list is initially empty (places already explored).

2. **Pick the node with the lowest total cost (f = g + h):**

   From the open list, choose the node with the smallest **f** value.

3. **Check if the goal is reached:**

   If the chosen node is the **goal**, the algorithm ends, the path is found.

4. **Generate neighboring nodes:**

   Find all valid neighbor nodes (top, down, left, right and diagonal).

   For each neighbor:

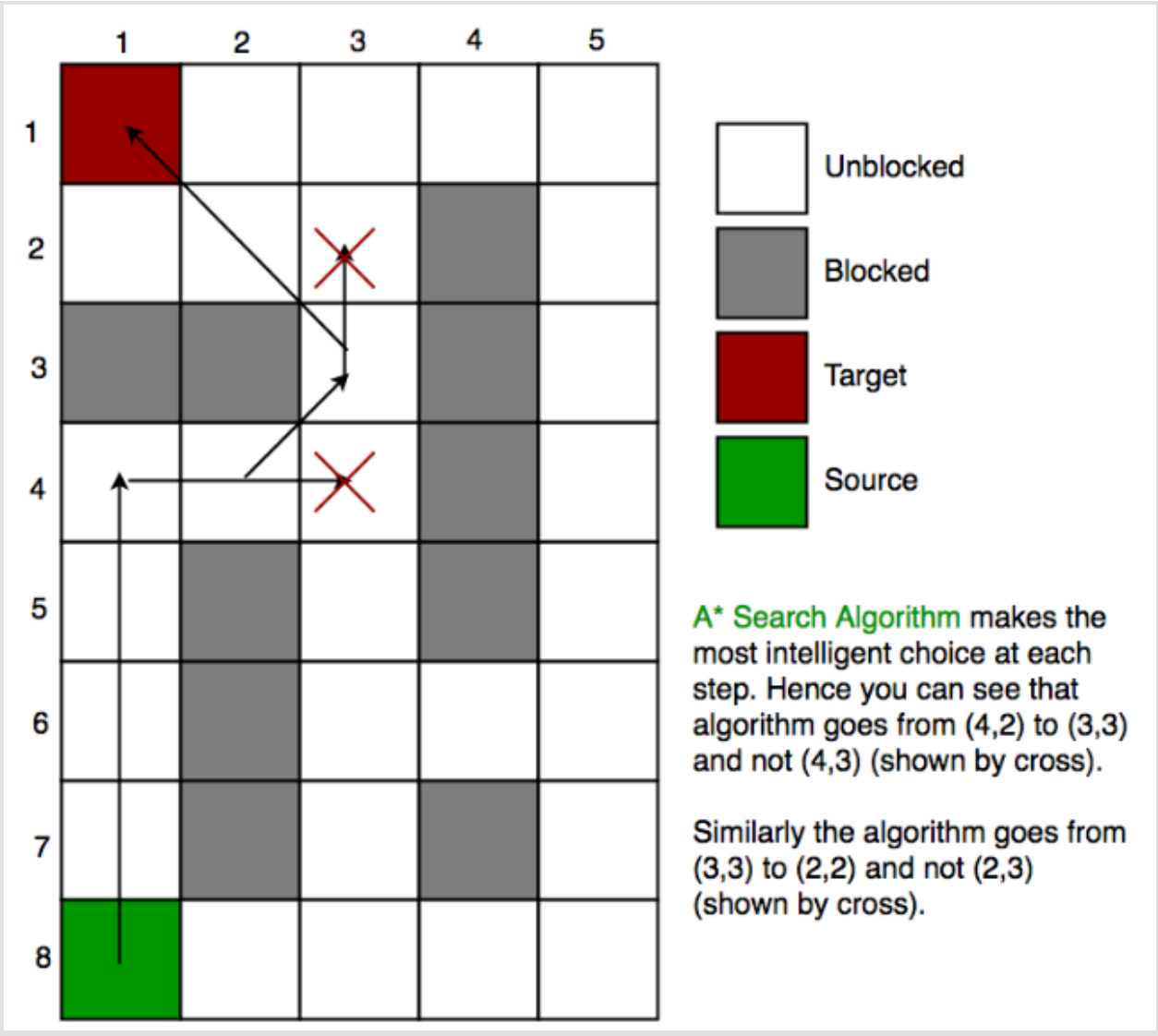
   - Calculate **g**, **h**, and **f** values.
   - Skip the neighbor if a **shorter path** to it already exists in the open or closed list.
   - Otherwise, **add the neighbor** to the open list with its **g**, **h**, and **f** values.

5. **Repeat the process:**

   Move the current node to the **closed list** and go back to step 2.

   Repeat until the **goal** is reached or the open list becomes empty (meaning no path exists).

Example


1
2
3
4
5
1
2
3
4
5
6
7
8
Unblocked
Blocked
Target
Source
A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).
Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

III. Sequential Implementation

Code Snippets

1. **Grid Setup:**

   Creates a 1000x1000 grid with random blocked and unblocked cells (20% blocked).

```
    // Allocate and initialize large grid
    int** grid = (int**)malloc(ROW * sizeof(int*));
    for (int i = 0; i < ROW; i++) {
        grid[i] = (int*)malloc(COL * sizeof(int));
        for (int j = 0; j < COL; j++) {
            grid[i][j] = rand() % 5 == 0 ? 0 : 1; // ~20% blocked
        }
    }
```

2. **Start & End:**

   Starts at (0, 0), ends at (999, 999). Checks if both are valid and unblocked.

```
Pair src = {0, 0};
Pair dest = {ROW - 1, COL - 1};

if (!isValid(src.first, src.second) || !isValid(dest.first,
dest.second)) {
    printf("Source or Destination is invalid\n");
    return;
}

if (!isUnBlocked(grid, src.first, src.second) || !isUnBlocked(grid,
dest.first, dest.second)) {
    printf("Source or Destination is blocked\n");
    return;
}
```

3. **Tracking:**

   Uses closedList to track visited cells and cellDetails to store cost and parent info.

```
bool** closedList = (bool**)malloc(ROW * sizeof(bool*));
for (int i = 0; i < ROW; i++) {
    closedList[i] = (bool*)calloc(COL, sizeof(bool));
}

cell** cellDetails = (cell**)malloc(ROW * sizeof(cell*));
for (int i = 0; i < ROW; i++) {
    cellDetails[i] = (cell*)malloc(COL * sizeof(cell));
    for (int j = 0; j < COL; j++) {
        cellDetails[i][j].f = FLT_MAX;
        cellDetails[i][j].g = FLT_MAX;
        cellDetails[i][j].h = FLT_MAX;
        cellDetails[i][j].parent_i = -1;
        cellDetails[i][j].parent_j = -1;
    }
}
```

4. **A* Algorithm:**

   Uses an open list to explore cells with the lowest total cost f = g + h.

   Considers 8 directions (including diagonals).

```
int directions[8][2] = {{-1, 0}, {1, 0}, {0, 1}, {0, -1},
                        {-1, 1}, {-1, -1}, {1, 1}, {1, -1}};
double costs[8] = {1.0, 1.0, 1.0, 1.0, 1.414, 1.414, 1.414, 1.414};

while (!isEmpty(&openList)) {
    pPair p = extractMin(&openList);
    i = p.second.first;
    j = p.second.second;
    closedList[i][j] = true;

    for (int k = 0; k < 8; k++) {
        int new_i = i + directions[k][0];
        int new_j = j + directions[k][1];

        if (isValid(new_i, new_j)) {
```

```
            if (isDestination(new_i, new_j, dest)) {
                // destination found actions...
            }

            if (!closedList[new_i][new_j] && isUnBlocked(grid,
new_i, new_j)) {
                double gNew = cellDetails[i][j].g + costs[k];
                double hNew = calculateHValue(new_i, new_j, dest);
                double fNew = gNew + hNew;

                if (cellDetails[new_i][new_j].f > fNew) {
                    insertOpenList(&openList, (pPair){fNew, {new_i,
new_j}});
                    cellDetails[new_i][new_j].f = fNew;
                    cellDetails[new_i][new_j].g = gNew;
                    cellDetails[new_i][new_j].h = hNew;
                    cellDetails[new_i][new_j].parent_i = i;
                    cellDetails[new_i][new_j].parent_j = j;
                }
            }
        }
    }
}
```

5. **Path Found:**

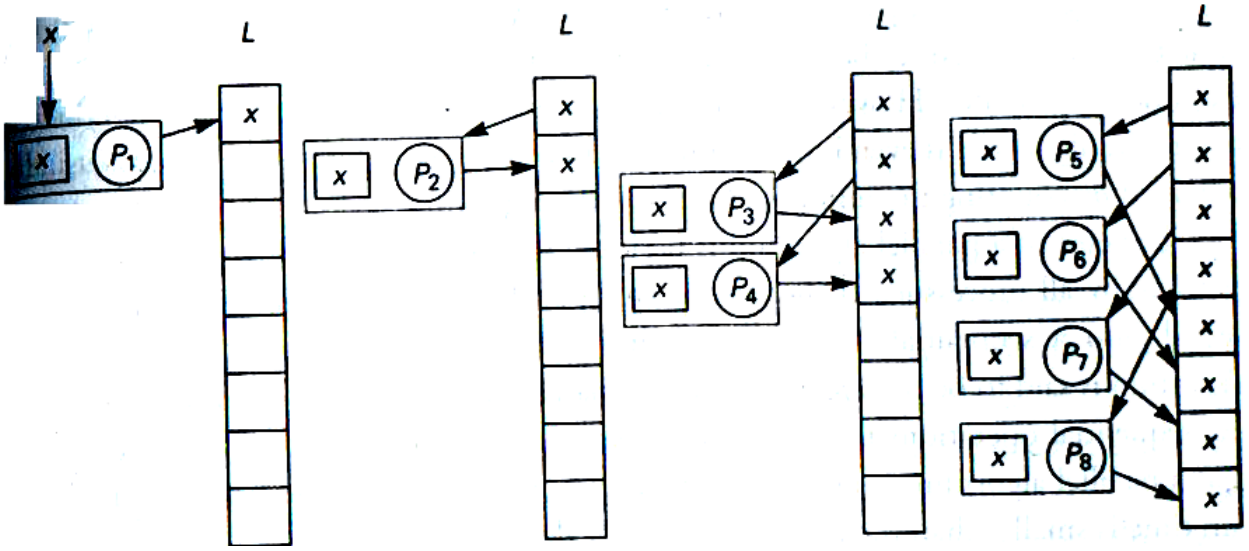
   If destination is reached, it backtracks using parent info to print the path and cost.

Performance Metrics

Total path cost: 1451.262

Execution Time: 3.94 seconds

IV. PRAM Implementation



Parallel Random Access Machine is a model for parallel computing where multiple processors operate in sync and share memory. In our project, we use the **EREW (Exclusive Read Exclusive Write)** model, which ensures that no two processors read from or write to the same memory location at the same time. This avoids data races and keeps the system consistent. We experimented with different numbers of threads and observed that using **10 threads** provided the fastest and most efficient result.

To implement this in practice, we use **mutex locks**. A mutex ensures that only one processor can access a critical section of memory at a time.

Code Snippets

1. **Open List with Mutex Protection:**

   To avoid race conditions when threads access the shared Open List, it is protected with a mutex:

```
void insertOpenList(OpenList* list, pPair element) {
    pthread_mutex_lock(&list->lock);
    if (list->size == list->capacity) {
        list->capacity *= 2;
        list->elements = (pPair*)realloc(list->elements,
list->capacity * sizeof(pPair));
    }
    list->elements[list->size++] = element;
    pthread_mutex_unlock(&list->lock);
}
```

   This ensures only one thread at a time modifies the OpenList, preserving data consistency across threads.

2. **Shared Data Structure**

```
    // Initialize shared data
    SharedData shared;
    shared.grid = grid;
    shared.src = src;
    shared.dest = dest;
    shared.foundDest = false;
    shared.activeThreads = NUM_THREADS;
    pthread_mutex_init(&shared.foundMutex, NULL);
    pthread_mutex_init(&shared.listMutex, NULL);
    pthread_cond_init(&shared.cond, NULL);
```

   The SharedData struct stores all data shared between threads, including the grid, open and closed lists, and path details. It also contains mutexes and a

condition variable to manage safe, synchronized access to shared resources during multithreaded A* execution.

3. **Thread Entry Function** – threadAStarSearch()

   Each thread repeatedly pulls nodes from the shared Open List:

```
void* threadAStarSearch(void* arg) {
      pthread_mutex_lock(&shared->listMutex);

      // Check if destination found or no more work
      if (shared->foundDest || (isEmpty(shared->openList) &&
shared->activeThreads == 1)) {
          shared->activeThreads--;
          pthread_mutex_unlock(&shared->listMutex);
          break;
      }

      // Wait if no work but other threads active
      while (!shared->foundDest && isEmpty(shared->openList) &&
shared->activeThreads > 1) {
          shared->activeThreads--;
          pthread_cond_wait(&shared->cond, &shared->listMutex);
          shared->activeThreads++;
      }
      pPair p = extractMin(shared->openList);
      pthread_mutex_unlock(&shared->listMutex);

      // If destination found
      if (isDestination(new_i, new_j, shared->dest)) {
          pthread_mutex_lock(&shared->foundMutex);
          if (!shared->foundDest) {
              shared->foundDest = true;   // mark destination found
              // update parent details...
          }
          pthread_mutex_unlock(&shared->foundMutex);
          pthread_cond_broadcast(&shared->cond);
          return NULL;
      }
```

- This function implements parallel A* search with multiple threads working simultaneously to find a path.
- Threads use mutex locks to safely access and update shared data like open and closed lists.
- If no work is available, threads wait on a condition variable until signaled by others.
- When a thread finds the destination, it sets a shared flag and broadcasts a signal to wake all waiting threads.

This is PRAM-style synchronization using mutexes to ensure exclusive access during critical updates.

4. **Condition Variable for Thread Coordination**

   If a thread finds the list empty, it waits for a signal from others:

```
pthread_cond_wait(&shared->cond, &shared->listMutex);
```

   And when a thread inserts a new node, it signals:

```
pthread_cond_signal(&shared->cond);
```

   This is important to **coordinate access without busy waiting**, where processors pause and resume based on shared state.

Performance Metrics

Execution Time: 12.57 seconds

Execution time was alot higher than in sequential because of overhead from thread synchronization, such as **locking and waiting on condition variables**.

V. OpenMP Implementation

OpenMP is an API that supports multi-platform shared memory multiprocessing programming. It allows to write parallel code using simple compiler directives, making it easier to utilize multiple CPU cores for faster execution.

In our A* implementation, we use OpenMP to parallelize parts of the algorithm and improve performance on large grids. Specifically, we use **#pragma omp parallel for** to parallelize loops, this tells the compiler to divide the loop iterations among available threads, allowing multiple parts of the loop to run at the same time.

Code Snippets

1. **Parallel Initialization of closedList and cellDetails**

   This OpenMP **directive #pragma omp parallel for** parallelizes the initialization of the 2D arrays used to track the A* algorithm's internal state. Each thread handles a different row, which speeds up memory setup for large grids.

```
#pragma omp parallel for
for (int i = 0; i < ROW; i++) {
    closedList[i] = (bool*)calloc(COL, sizeof(bool));
    cellDetails[i] = (cell*)malloc(COL * sizeof(cell));
    for (int j = 0; j < COL; j++) {
        cellDetails[i][j].f = FLT_MAX;
        cellDetails[i][j].g = FLT_MAX;
        cellDetails[i][j].h = FLT_MAX;
        cellDetails[i][j].parent_i = -1;
        cellDetails[i][j].parent_j = -1;
    }
}
```

2. **Grid initialization**

   This loop initializes the grid with random blocked/unblocked cells. The outer loop is parallelized so multiple rows are filled at once, significantly reducing the time needed to create large grids (1000×1000).

```
#pragma omp parallel for
for (int i = 0; i < ROW; i++) {
    grid[i] = (int*)malloc(COL * sizeof(int));
    for (int j = 0; j < COL; j++) {
        grid[i][j] = rand() % 5 == 0 ? 0 : 1; // ~20% blocked
    }
}
```

Performance Metrics

Execution Time: 3.70 seconds

Using OpenMP gave a slight performance improvement of **5.6%**.

This speedup is modest because only the **initialization loops** were parallelized; the core A* algorithm itself remains **sequential**, which limits overall performance gain.

Improved OpenMP

To improve performance, we parallelized the **successor node exploration** part of the A* algorithm using OpenMP. This helps utilize multiple CPU cores when checking the 8 possible neighbor nodes around the current node.

Code Snippet

```
#pragma omp parallel for shared(foundDest)
for (int k = 0; k < 8; k++) {
```

```
    if (foundDest) continue;

    int new_i = i + directions[k][0];
    int new_j = j + directions[k][1];

    if (isValid(new_i, new_j)) {
        if (isDestination(new_i, new_j, dest)) {
            #pragma omp critical
            {
                if (!foundDest) {
                    foundDest = true;
                    cellDetails[new_i][new_j].parent_i = i;
                    cellDetails[new_i][new_j].parent_j = j;
                    printf("Destination found!\n");
                    tracePath(cellDetails, dest);
                }
            }
        } else if (!closedList[new_i][new_j] && isUnBlocked(grid, new_i,
new_j)) {
            double gNew = cellDetails[i][j].g + costs[k];
            double hNew = calculateHValue(new_i, new_j, dest);
            double fNew = gNew + hNew;

            #pragma omp critical
            {
                if (cellDetails[new_i][new_j].f > fNew) {
                    insertOpenList(openList, (pPair){fNew, {new_i,
new_j}});
                    cellDetails[new_i][new_j].f = fNew;
                    cellDetails[new_i][new_j].g = gNew;
                    cellDetails[new_i][new_j].h = hNew;
                    cellDetails[new_i][new_j].parent_i = i;
                    cellDetails[new_i][new_j].parent_j = j;
                }
            }
        }
    }
}
```

Here, **#pragma omp parallel for** allows each of the 8 neighbor directions to be processed in parallel. Since some shared variables are being updated (foundDest, open list insertions), we use **#pragma omp critical** to avoid race conditions. This results in significant speed-up on multi-core CPUs for large grids.

Performance Metrics

Execution Time: 0.55 seconds

Using OpenMP without algorithm-level parallelism, the execution time was 3.70 seconds, whereas with proper parallelization applied to the core algorithm, it dropped significantly to 0.55 seconds. This sharp improvement highlights the effectiveness of parallelizing computational logic rather than just outer structures, resulting in better thread utilization and a substantial boost in performance.

VI. CUDA Implementation

CUDA (Compute Unified Device Architecture) is NVIDIA's platform for parallel computing using GPUs. In our implementation of the A* algorithm, we used CUDA to parallelize the heuristic cost calculation on the GPU, allowing each thread to compute the cost for a single grid cell. This speeds up the most intensive part of the algorithm, while the rest of the pathfinding still runs on the CPU.

Code Snippets

1. **CUDA Kernel to Compute Heuristics in Parallel**

```
__global__ void computeHeuristics(float* h_costs) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N * N) {
        int x = idx / N;
        int y = idx % N;
        h_costs[idx] = sqrtf((float)(N - 1 - x) * (N - 1 - x) +
(float)(N - 1 - y) * (N - 1 - y));
    }
}
```

This kernel runs on the GPU. Each thread computes the Euclidean distance from its grid cell (x, y) to the goal (N-1, N-1) and stores it in **h_costs**. It parallelizes heuristic computation for all N*N cells.

2. **Kernel Launch Configuration**

```
#define N 1000
#define BLOCK_SIZE 256
int blocks = (N * N + BLOCK_SIZE - 1) / BLOCK_SIZE;
computeHeuristics<<<blocks, BLOCK_SIZE>>>(d_costs);
```

This launches the kernel with enough blocks and threads to cover all grid cells. BLOCK_SIZE is the number of threads per block here we used 256, and each thread calculates one heuristic value.

3. **GPU & CPU Memory Allocation**

```
float* h_costs;
float* d_costs;
size_t size = N * N * sizeof(float);
h_costs = (float*)malloc(size);
cudaMalloc(&d_costs, size);
```

Allocates space for **h_costs on CPU (host)** and **d_costs on GPU (device**) to store heuristic values for each cell in a 2D grid.

4. **Copying Data from GPU to CPU**

```
cudaMemcpy(h_costs, d_costs, size, cudaMemcpyDeviceToHost);
```

After the GPU finishes computing, this copies the heuristics back to the CPU so that the A* search can access them during pathfinding.

5. **Using Heuristics in A***

```
open_list.push({0, 0, h_costs[to1D(0, 0)], 0});
```

The A* algorithm starts with the (0, 0) node, using the heuristic precomputed on the GPU. This helps prioritize nodes closer to the goal faster.

Performance Metrics

Execution Time: 0.005 seconds

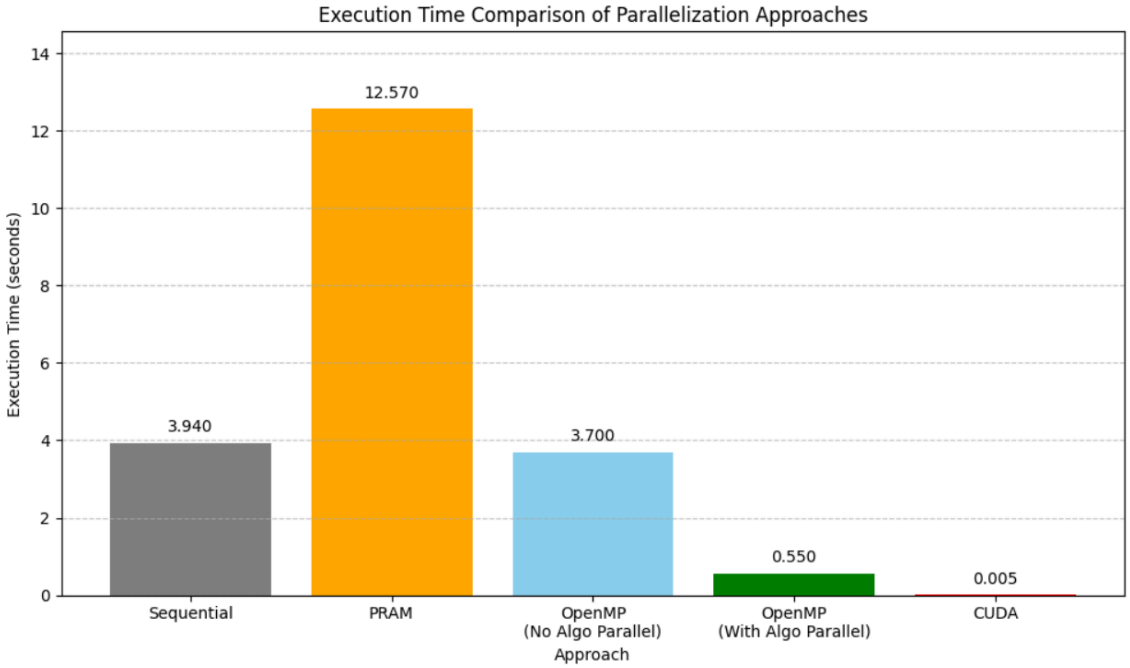
This is extremely fast as compared to other implementations, showcasing the power of **massive parallelism** on the GPU. By assigning each grid cell to a separate thread, CUDA drastically reduced computation time compared to a CPU loop, which would process one cell at a time.

VII. Overall Performance Analysis

Approach	**Execution Time (seconds)**
Sequential	3.94
PRAM (10 Threads)	12.57
OpenMP (without algorithm parallelization)	3.70
OpenMP (with algorithm parallelization	0.55
CUDA	0.005

The **sequential approach** serves as the **baseline** for evaluating performance. The **PRAM model**, while parallel in theory, **did not yield** the expected efficiency, possibly due to overhead or implementation limitations. **OpenMP without algorithm-level parallelization** showed **modest improvement**, but incorporating **algorithm-level parallelism significantly boosted performance**. **CUDA outperformed** all other methods, demonstrating the superior capability of GPU-based parallel processing for handling computationally intensive tasks.

Visual Analysis



The comparison graph visually highlights the performance differences across various implementations. It clearly shows the dramatic speedup achieved by CUDA and the benefits of algorithm-level parallelism with OpenMP.