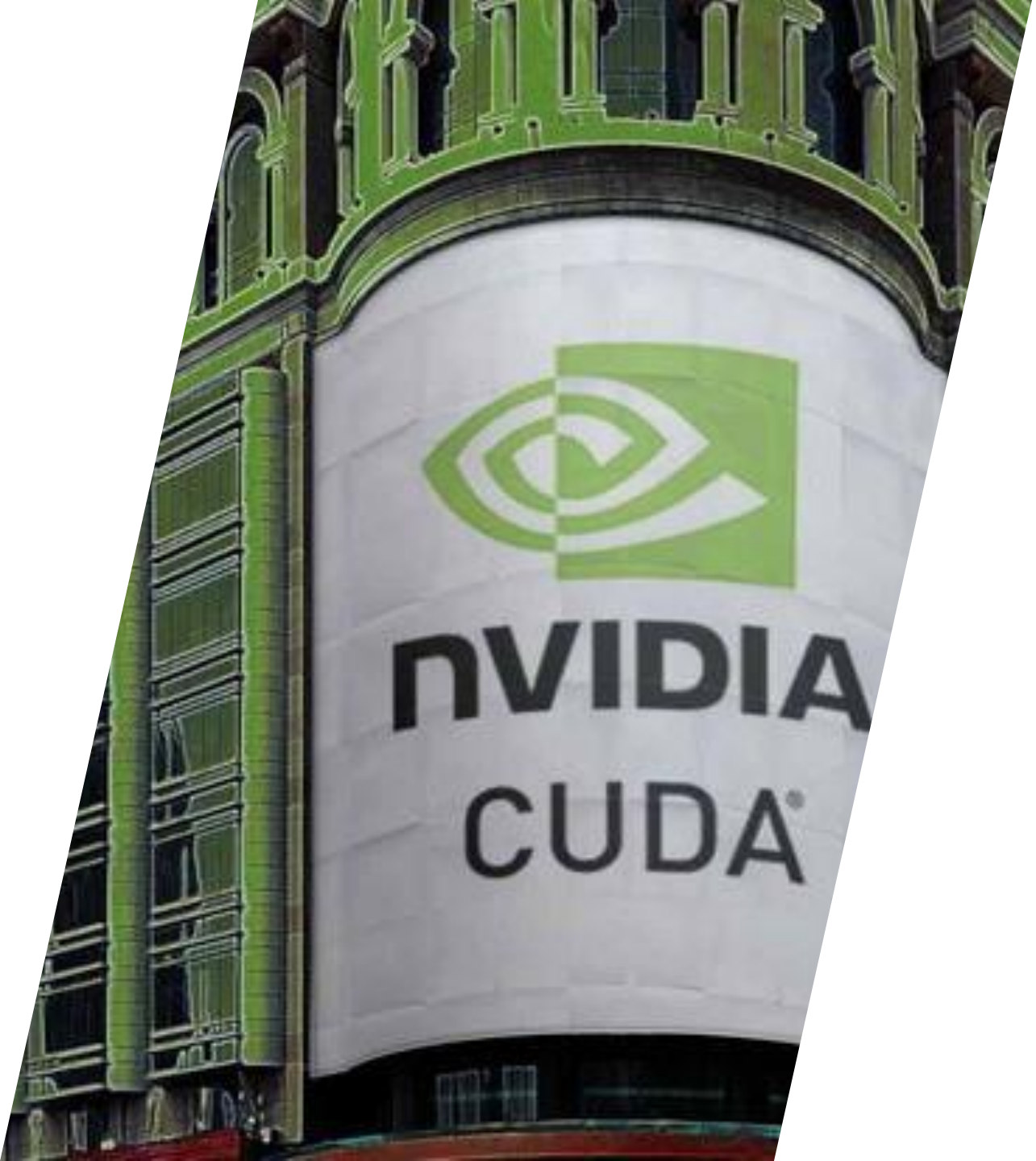
PARALLEL PROGRAMMING
FINAL PROJECT:
DYNAMIC PROGRAMMING MINIMUM COST TO CUT A STICK

GROUP 33

112065528 洪巧雲

112062644 林煒博

12.17.2024

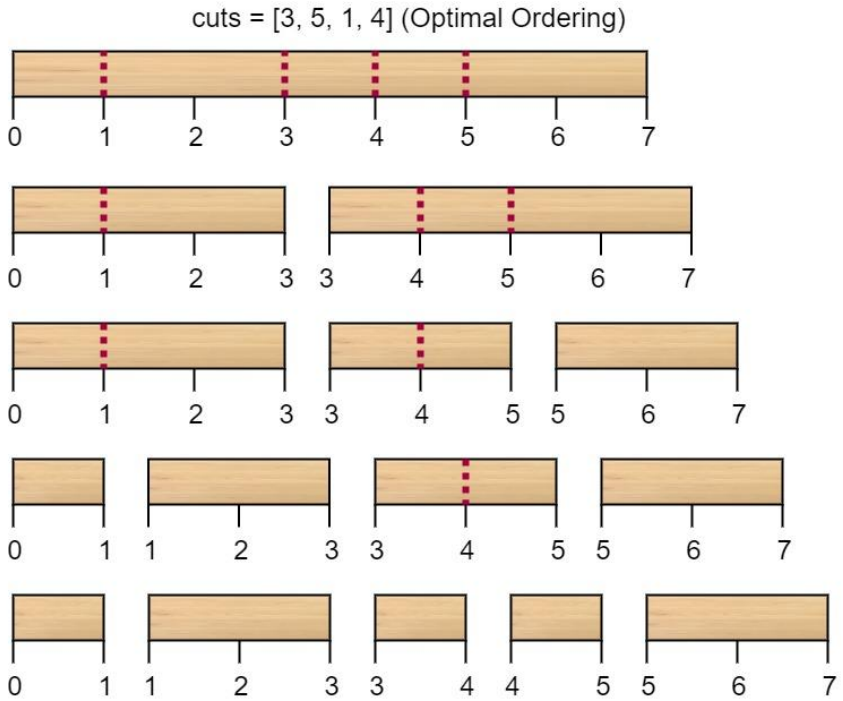


OUTLINE

- Introduction
- Optimization
  1. CUDA
  2. CPU
  3. Baseline
  4. Coalesce memory
  5. Parallel min reduce
  6. First reduce during load
  7. Unroll all
  8. Hybrid: MPI + OpenMP
  9. Pthread
- Experiment and Result
- Conclusion

INTRODUCTION

Dynamic Programming
Minimum cost to cut a stick



Given a wooden stick of length n units. The stick is labelled from 0 to n. For example, a stick of length 6 is labelled as follows:

Given an integer array `cuts` where `cuts[i]` denotes a position you should perform a cut at.

You should perform the cuts in order, you can change the order of the cuts as you wish.

The cost of one cut is the length of the stick to be cut, the total cost is the sum of costs of all cuts. When you cut a stick, it will be split into two smaller sticks (i.e. the sum of their lengths is the length of the stick before the cut). Please refer to the first example for a better explanation.

Return the minimum total cost of the cuts.

OPTIMIZATION

CPU

- Solved by using dynamic programming.
- State transition relation is as follows:
  - `dp[l][r]` means the minimum cost for a cutting segment `[l, r]` (inclusive)
  - `dp[l][r] = min(k = l+1 .. r-1){ dp[l][k] + dp[k][r] + (len of [l, r]) }`
  - `dp[l][r] = 0`; if `r - l = 1`
- Answer is `dp[0][c - 1]`
  - Set `c` = len of array `cuts`
- Time Complexity : `O(c^3)`
- Space Complexity : `O(c^2)`

OPTIMIZATION

CPU

```
38  int minCost(int n, int cnt, int *cuts)
39  {
40      cuts[cnt++] = 0;
41      cuts[cnt++] = n;
42      sort(cuts, cuts + cnt);
43      vector<vector<int>> dp(cnt, vector<int>(cnt));
44      for (int len = 3; len <= cnt; ++len)
45      {
46          for (int l = 0; l + len - 1 < cnt; ++l)
47          {
48              int r = l + len - 1;
49              int mm = INT_MAX;
50              for (int k = l + 1; k < r; ++k)
51              {
52                  mm = min(mm, dp[l][k] + dp[k][r]);
53              }
54              mm += cuts[r] - cuts[l];
55              dp[l][r] = mm;
56          }
57      }
58      return dp[0][cnt - 1];
59  }
```

OPTIMIZATION: CUDA

Baseline

- For each kernel function, each processing a starting point `l`, with length `len`.
- Basically, replacing every iteration of the loop `for (int l = 0; l + len - 1 < cnt; ++l)` from CPU version with kernel call.
- Execution time for testcase size 10000 : 73.30 sec

OPTIMIZATION: CUDA

Baseline

```
68    // Dynamic programming using CUDA
69    for (int len = 3; len <= c; ++len)
70    {
71        int num = c - len + 1;
72        int block_num = (num + NT - 1) / NT;
73        min_reduce<<<block_num, NT>>>(c, len, dp);
74    }
75    // Retrieve the result by get dp[0][c-1]
76    cudaMemcpy(&res, &dp[0 * c + c - 1], sizeof(int), cudaMemcpyDeviceToHost);
77    return res;
```

```
80    __global__ void min_reduce(int c, int len, int *dp)
81    {
82        int l = blockIdx.x * blockDim.x + threadIdx.x;
83        int r = l + len - 1;
84        if (r >= c)
85            return;
86        int mm = INT_MAX;
87        for (int k = l + 1; k < r; ++k)
88        {
89            mm = min(mm, dp[l * c + k] + dp[k * c + r]);
90        }
91        mm += DATA_GPU[r] - DATA_GPU[l];
92        dp[l * c + r] = mm;
93    }
```

OPTIMIZATION: CUDA

Coalesce Memory

- Change the transition relation with memory friendly one.
- Originally, each kernel call to solve `dp[l][r]` will iterate through `dp[l][k], dp[k][r]`.
  - Threads in the same warp will have consecutive `l`, but accessing isn't coalesced!
- Change meaning of `dp[l][r]` from min value for segment `[l, r]`.
  To `dp[len][l]`, which means min value for segment length `len` which starts at `l`.
- Thus, each kernel call to solve `dp[len][l]` will iterate through `dp[k][l], dp[len-k+1][l+k-1]`.
  - Threads in the same warp will have consecutive `l`, this way is coalesced!
- Execution time for testcase size 10000 : 15.94 sec

OPTIMIZATION: CUDA

Coalesce Memory

```
69      for (int len = 3; len <= c; ++len)
70      {
71          int num = c - len + 1;
72          int block_num = (num + NT - 1) / NT;
73          min_reduce<<<block_num, NT>>>(c, len, dp);
74      }
75      // Retrieve the result
76      // get dp[c][0]
77      cudaMemcpy(&res, &dp[c * c + 0], sizeof(int), cudaMemcpyDeviceToHost);
78      return res;
```

```
81  __global__ void min_reduce(int c, int len, int *dp)
82  {
83      int l = blockIdx.x * blockDim.x + threadIdx.x;
84      int r = l + len - 1;
85      if (r >= c)
86          return;
87      dp[len * c + l] = INT_MAX;
88      for (int leftLen = 2; leftLen < len; ++leftLen)
89      {
90          int rightLen = len - leftLen + 1, rightIdx = l + leftLen - 1;
91          dp[len * c + l] = min(
92              dp[len * c + l],
93              dp[leftLen * c + l] + dp[rightLen * c + rightIdx] + DATA_GPU[r] - DATA_GPU[l]);
94      }
95  }
```

OPTIMIZATION: CUDA

Parallel min reduce

- Use more threads to generate data, and them min reduce them for faster result.
- For example: `dp[100][0]` necessitates the result of `dp[2][0]` + `dp[99][1]`, `dp[3][0]` + `dp[98][2]`, `dp[4][0]` + `dp[97][3]`...
- If we put the result of `dp[2][0]` + `dp[99][1]` into `minData[0][2]`, `dp[3][0]` + `dp[98][2]` into `minData[0][3]`...
- And then do minreduce on `minData[0][:]`, we could utilize more threads.
- Execution time for testcase size 10000 : 10.57 sec

OPTIMIZATION: CUDA

Parallel min reduce

```
76      for (int len = 3; len <= c; ++len)
77      {
78          int block_num_l = c - len + 1;
79          int block_len_cnt = 64;
80          preprocess_min_data<<<dim3(ceilDiv(block_num_l, NT), block_len_cnt), NT>>>(c, len, dp, minData);
81          do
82          {
83              block_len_cnt = ceilDiv(block_len_cnt, NT);
84              min_reduce<<<dim3(block_len_cnt, block_num_l), NT>>>(c, block_len_cnt, minData);
85          } while (block_len_cnt > 1);
86          postprocess_min_data<<<block_num_l, 1>>>(c, len, dp, minData);
87      }
```

```
94      __global__ void preprocess_min_data(int c, int len, int *dp, int *minData)
95      {
96          int l = blockIdx.x * blockDim.x + threadIdx.x;
97          int r = l + len - 1;
98          if (r >= c)
99              return;
100         int leftLen = blockIdx.y + 2;
101         int mm = INT_MAX;
102         for (; leftLen < len; leftLen += gridDim.y)
103         {
104             int rightLen = len - leftLen + 1, rightIdx = l + leftLen - 1;
105             mm = min(mm, dp[leftLen * c + l] + dp[rightLen * c + rightIdx]);
106         }
107         minData[l * c + blockIdx.y] = mm;
108     }
```

OPTIMIZATION: CUDA

Parallel min reduce

```
110  __global__ void postprocess_min_data(int c, int len, int *dp, int *minData)
111  {
112      int l = blockIdx.x;
113      int r = l + len - 1;
114      int baseCost = DATA_GPU[r] - DATA_GPU[l];
115      dp[len * c + l] = baseCost + minData[l * c + 0];
116  }
117
118  __global__ void min_reduce(int c, int dataLen, int *minData)
119  {
120      __shared__ int smem[NT];
121      int l = blockIdx.y;
122      int minDataIdx = blockIdx.x * blockDim.x + threadIdx.x;
123      int tid = threadIdx.x;
124      smem[tid] = minDataIdx >= dataLen ? INT_MAX : minData[l * c + minDataIdx];
125      __syncthreads();
126      for (int s = blockDim.x / 2; s > 0; s >>= 1)
127      {
128          if (tid < s)
129          {
130              smem[tid] = min(smem[tid], smem[tid + s]);
131          }
132          __syncthreads();
133      }
134      if (tid == 0)
135      {
136          minData[l * c + blockIdx.x] = smem[0];
137      }
138  }
```

OPTIMIZATION: CUDA

First reduce during load

- Half of the threads in minreduce are idle after first iteration,
- So if we first do a minimum operation on data load, we require half the threads.
- Execution time for testcase size 10000 : 7.52 sec

OPTIMIZATION: CUDA

First reduce during load

```
76      for (int len = 3; len <= c; ++len)
77      {
78          int block_num_l = c - len + 1;
79          int block_len_cnt = 64;
80          preprocess_min_data<<<dim3(ceilDiv(block_num_l, NT), block_len_cnt), NT>>>(c, len, dp, minData);
81          do
82          {
83              min_reduce<<<dim3(1, block_num_l), block_len_cnt / 2>>>(c, block_len_cnt, minData);
84              block_len_cnt = ceilDiv(block_len_cnt, NT);
85          } while (block_len_cnt > 1);
86          postprocess_min_data<<<block_num_l, 1>>>(c, len, dp, minData);
87      }
```

```
118     __global__ void min_reduce(int c, int dataLen, int *minData)
119     {
120         __shared__ int smem[NT];
121         int l = blockIdx.y;
122         int minDataIdx = blockIdx.x * blockDim.x + threadIdx.x;
123         int tid = threadIdx.x;
124         smem[tid] = minDataIdx >= dataLen ? INT_MAX : minData[l * c + minDataIdx];
125         smem[tid] = min(
126             smem[tid],
127             minDataIdx + blockDim.x >= dataLen ? INT_MAX : minData[l * c + minDataIdx + blockDim.x]);
128         __syncthreads();
129         for (int s = blockDim.x / 2; s > 0; s >>= 1)
130         {
131             if (tid < s)
```

OPTIMIZATION: CUDA

Unroll all

- If we compress the data so that minreduce can be done in a single warp, there's no need to call `__syncthreads()`, and calling to `min_reduce` can be reduced to single block on x dimension, thus eliminates the while loop.
- Execution time for testcase size 10000 : 7.10 sec

OPTIMIZATION: CUDA

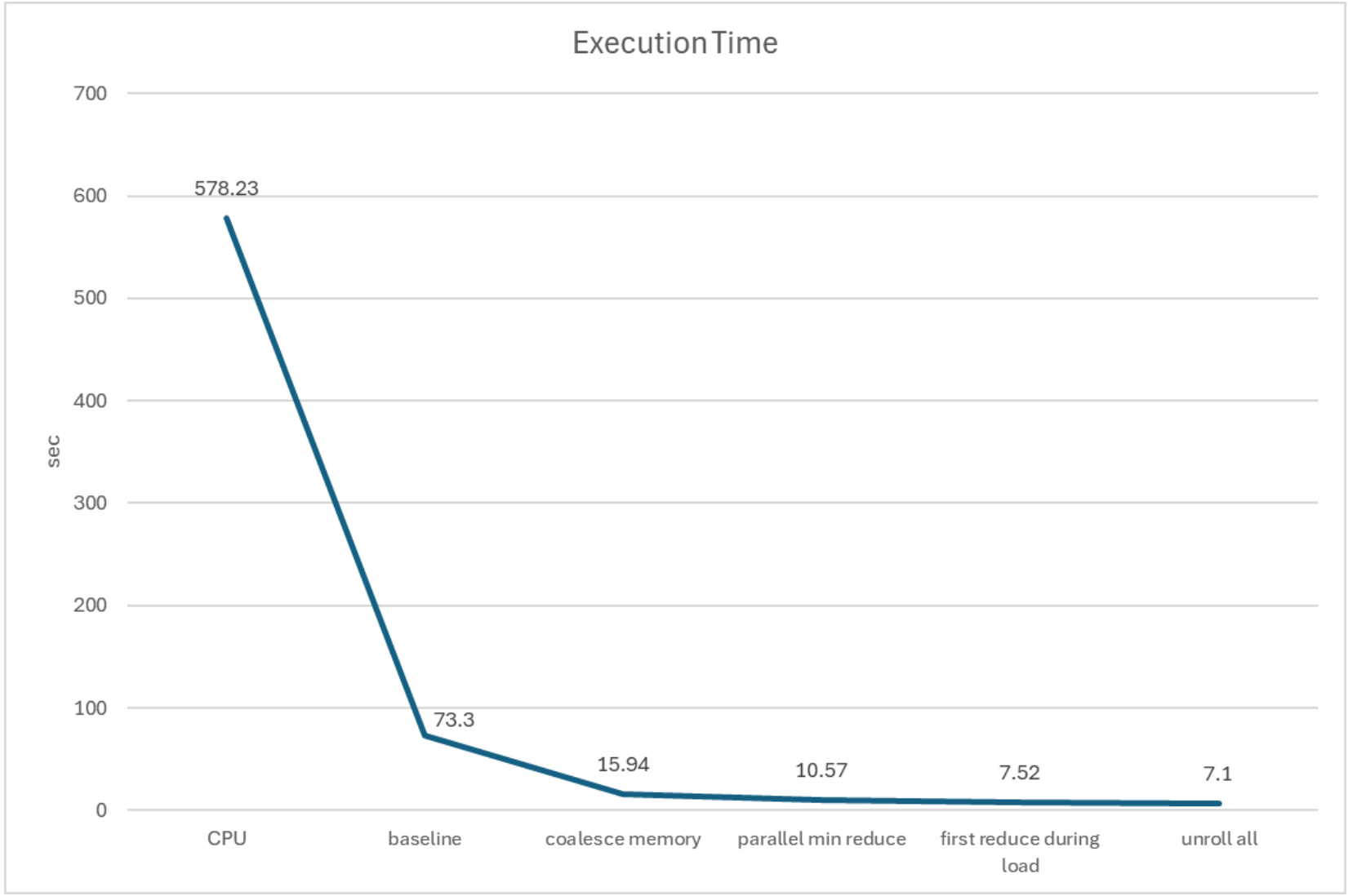
Unroll all

```
76      for (int len = 3; len <= c; ++len)
77      {
78          int block_num_l = c - len + 1;
79          preprocess_min_data<<<dim3(ceilDiv(block_num_l, NT), 32), NT>>>(c, len, dp, minData);
80          min_reduce<<<dim3(1, block_num_l), 16>>>(c, minData);
81          postprocess_min_data<<<block_num_l, 1>>>(c, len, dp, minData);
82      }
```

```
113   __global__ void min_reduce(int c, volatile int *minData)
114   {
115       __shared__ volatile int smem[16];
116       int l = blockIdx.y;
117       int tid = threadIdx.x;
118       smem[tid] = min(minData[l * c + tid], minData[l * c + tid + 16]);
119       smem[tid] = min(smem[tid], smem[tid + 8]);
120       smem[tid] = min(smem[tid], smem[tid + 4]);
121       smem[tid] = min(smem[tid], smem[tid + 2]);
122       smem[tid] = min(smem[tid], smem[tid + 1]);
123       if (tid == 0)
124           minData[l * c + blockIdx.x] = smem[0];
125   }
```

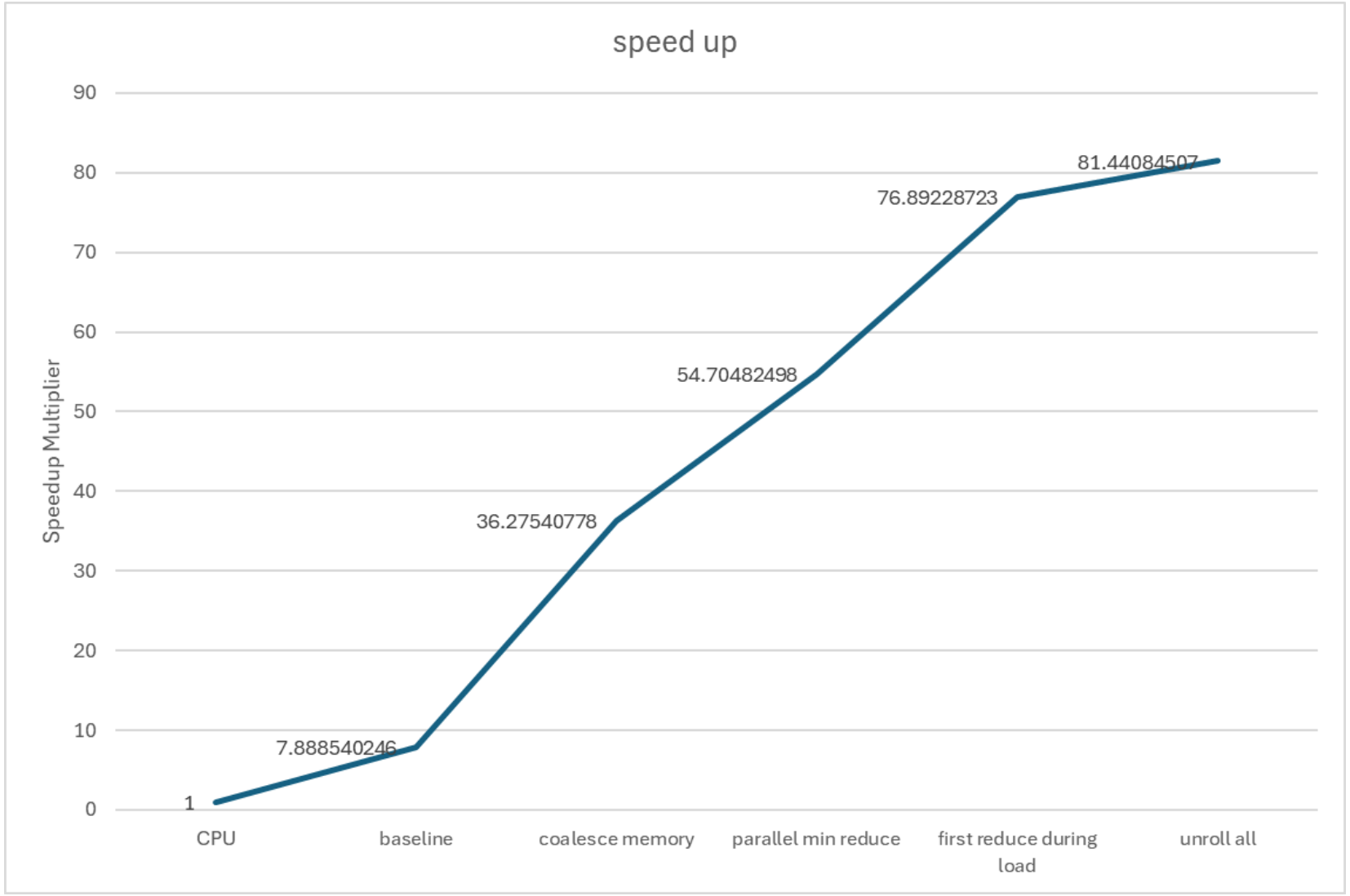
GPU EXECUTION TIME

Testcase Size: 10000

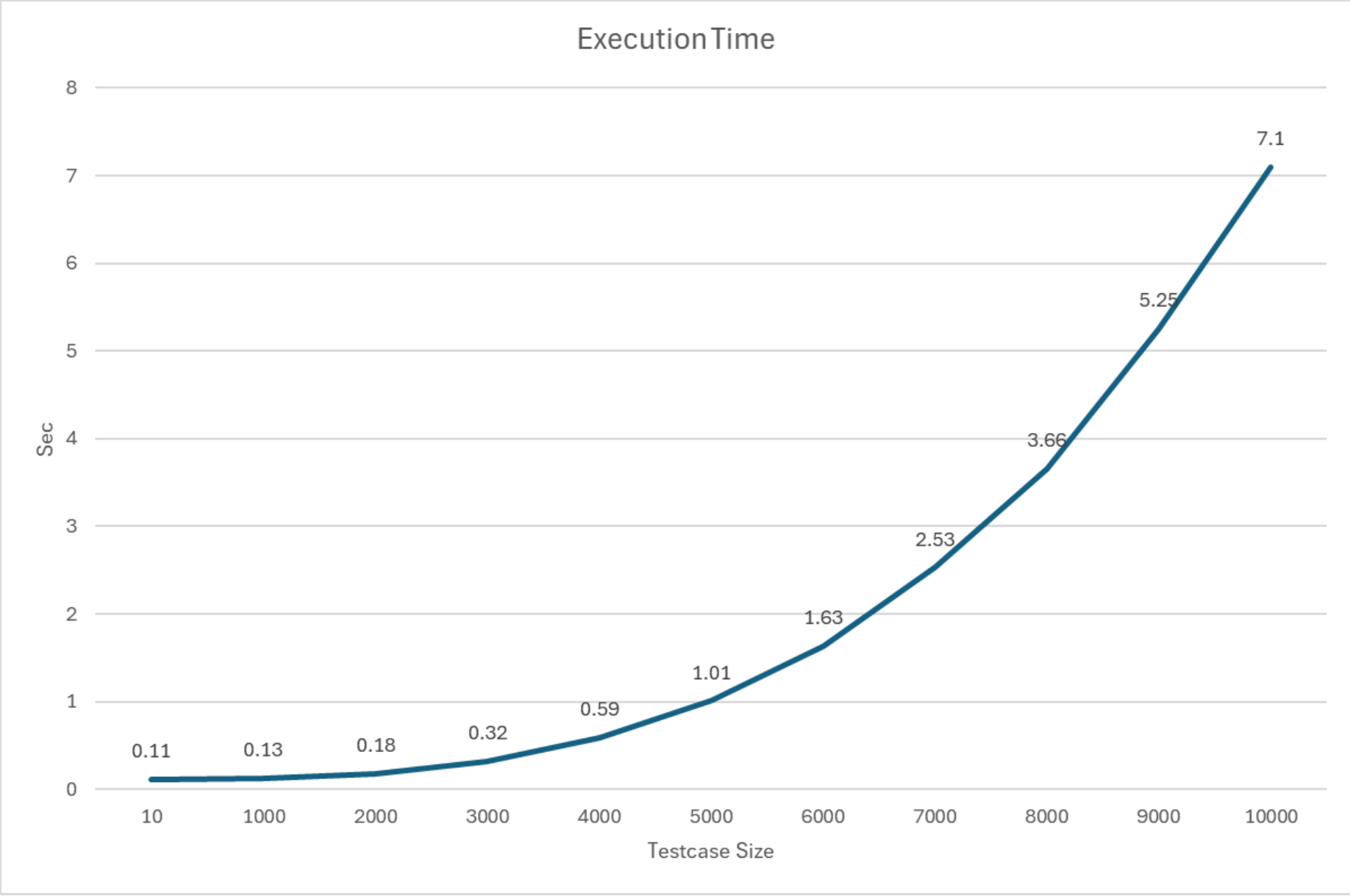


GPU EXECUTION SPEEDUP

Testcase Size: 10000



GPU EXECUTION TIME PER DATA SIZE



OPTIMIZATION

Hybrid: MPI + OpenMP

```
MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&C, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(DATA, C, MPI_INT, 0, MPI_COMM_WORLD);
```

```
// Pre-allocate buffers to avoid repeated allocations
// Each len iteration has (c-len+1) elements to update
vector<int> local_buf;
vector<int> global_buf;
```

```
// Non-blocking collective reduce
MPI_Request req;
MPI_Iallreduce(local_buf.data(), global_buf.data(), total_tasks, MPI_INT, MPI_MIN, MPI_COMM_WORLD, &req);

// Potentially, here we could do some non-dependent computations or preparations
// But since the DP strictly depends on updated results, we must wait.

MPI_Wait(&req, MPI_STATUS_IGNORE);
```

- MPI:
  - Use MPI_Bcast
  - Pre-allocate buffers to avoid repeated allocations.
  - Non-blocking collective reduce: MPI_Iallreduce + MPI_Wait
- OpenMP:

```
// Using static scheduling for better performance
#pragma omp parallel for schedule(static)
```

  - Using static scheduling for better performance
- The tasks are divided into chunks as follows:
  - Basic Division: total_tasks is divided by size to determine the chunk_size for each process.
  - Handle Remainder: The remainder is distributed among the first few processes (rank < remainder).
  - Start Index: The starting task index for each process (start_l) is calculated using the rank, chunk size, and whether the process gets an extra task from the remainder.
  - End Index: The ending task index (end_l) is calculated based on the start_l, chunk_size, and whether the process gets an extra task. It is adjusted if it exceeds total_tasks.

```
int chunk_size = total_tasks / size;
int remainder = total_tasks % size;
int start_l = rank * chunk_size + (rank < remainder ? rank : remainder);
int end_l = start_l + chunk_size + (rank < remainder ? 1 : 0);
if (end_l > total_tasks) end_l = total_tasks;
```

OPTIMIZATION

Pthread

- Parallelizing DP Computation:
  - For each segment length len, multiple dp[l][r] entries need to be computed. These computations for different l (left indices) are independent of each other, making them suitable for parallel execution.

```
struct ThreadArgs {
    int len;
    int l_start;
    int l_end;
    int c;
    int *cuts;
    vector<vector<int>> *dp;
};
```

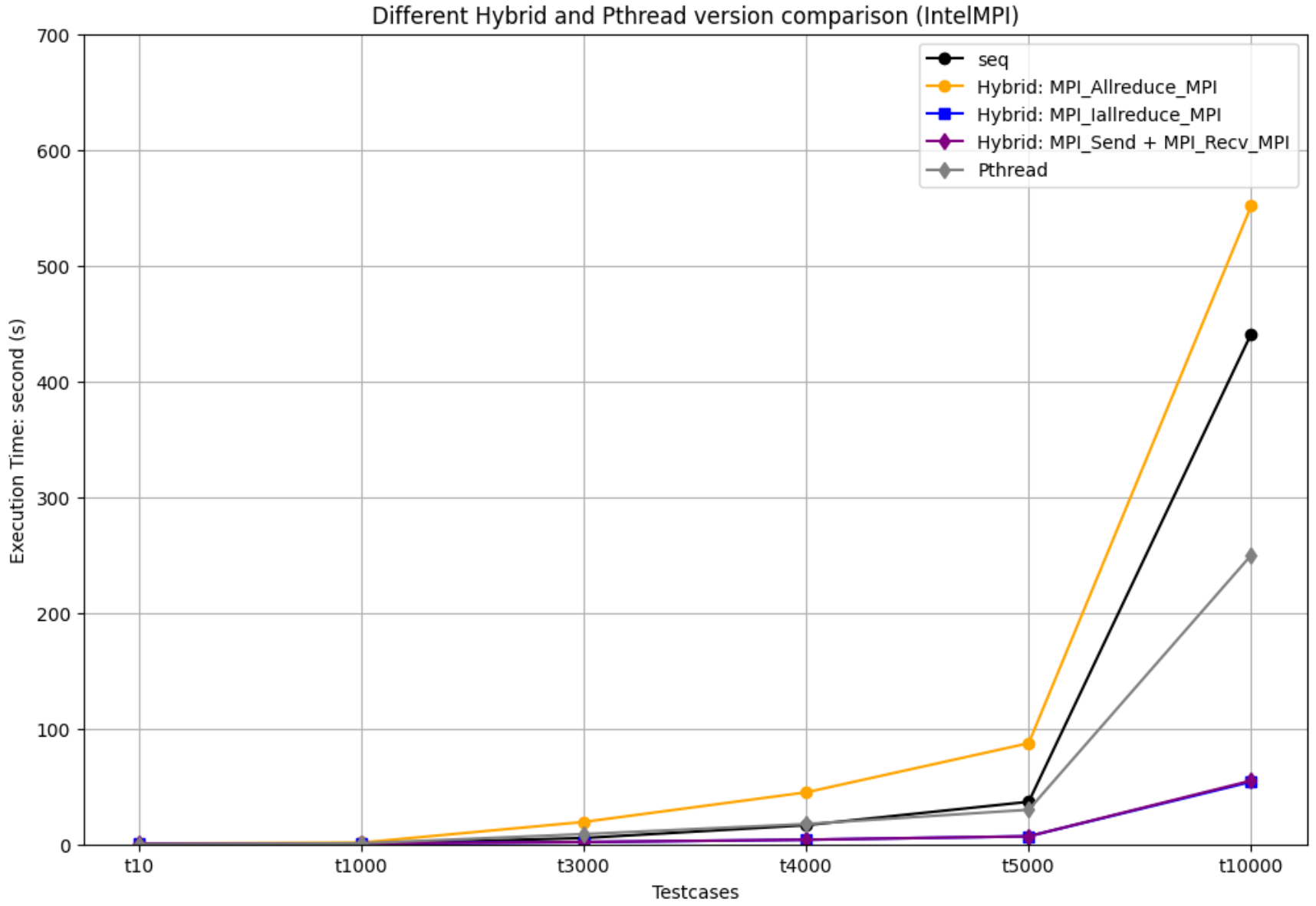
- Thread Management:
  - Thread Creation: pthread_create

```
pthread_create(&threads[i], NULL, compute_dp_range, (void*)&args[i]);
```

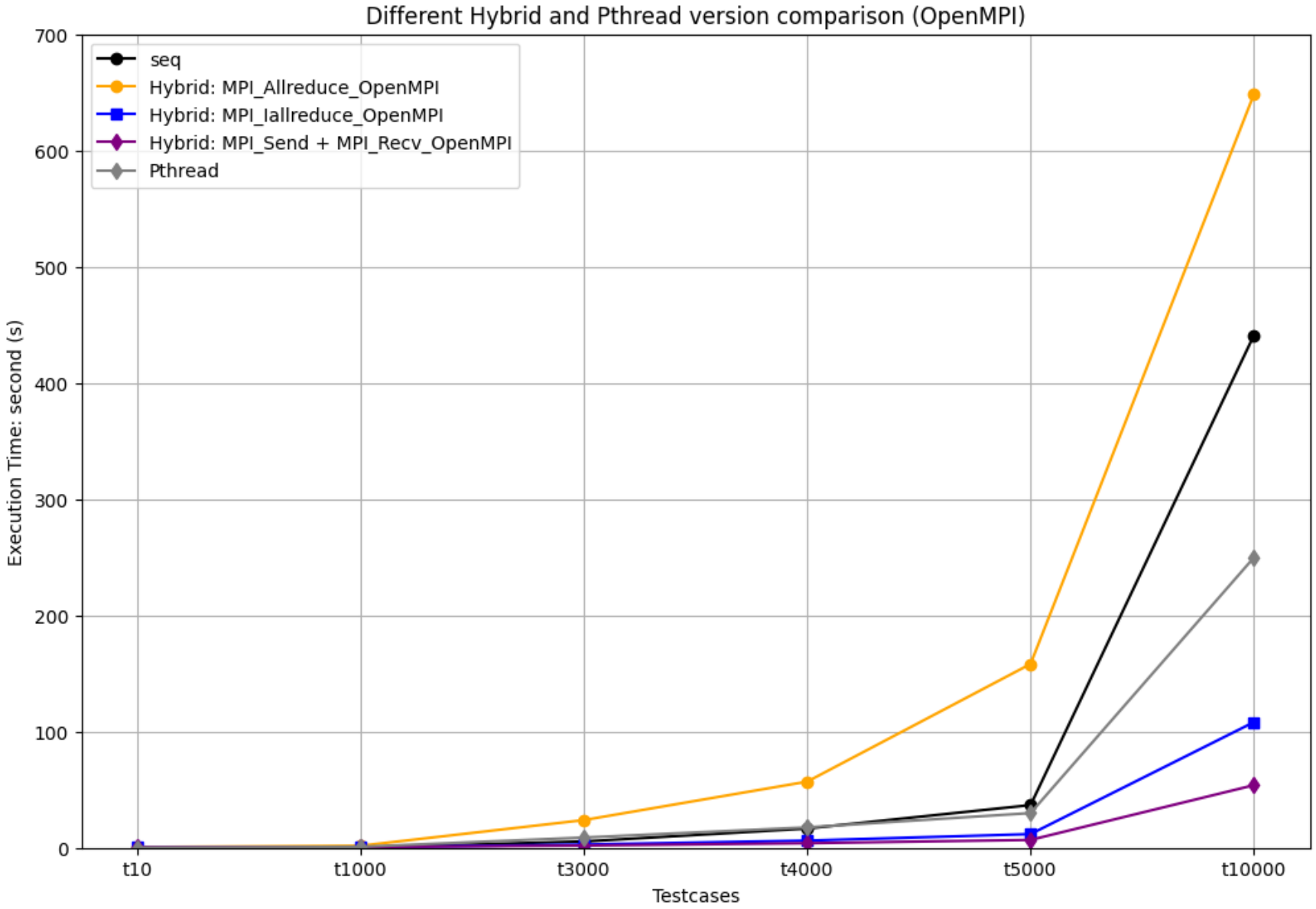
  - Thread Arguments: A ThreadArgs structure is defined to pass necessary information to each thread
  - Synchronization: pthread_join

```
// 等待所有執行緒完成
for (int i = 0; i < T; i++) {
    pthread_join(threads[i], NULL);
}
```

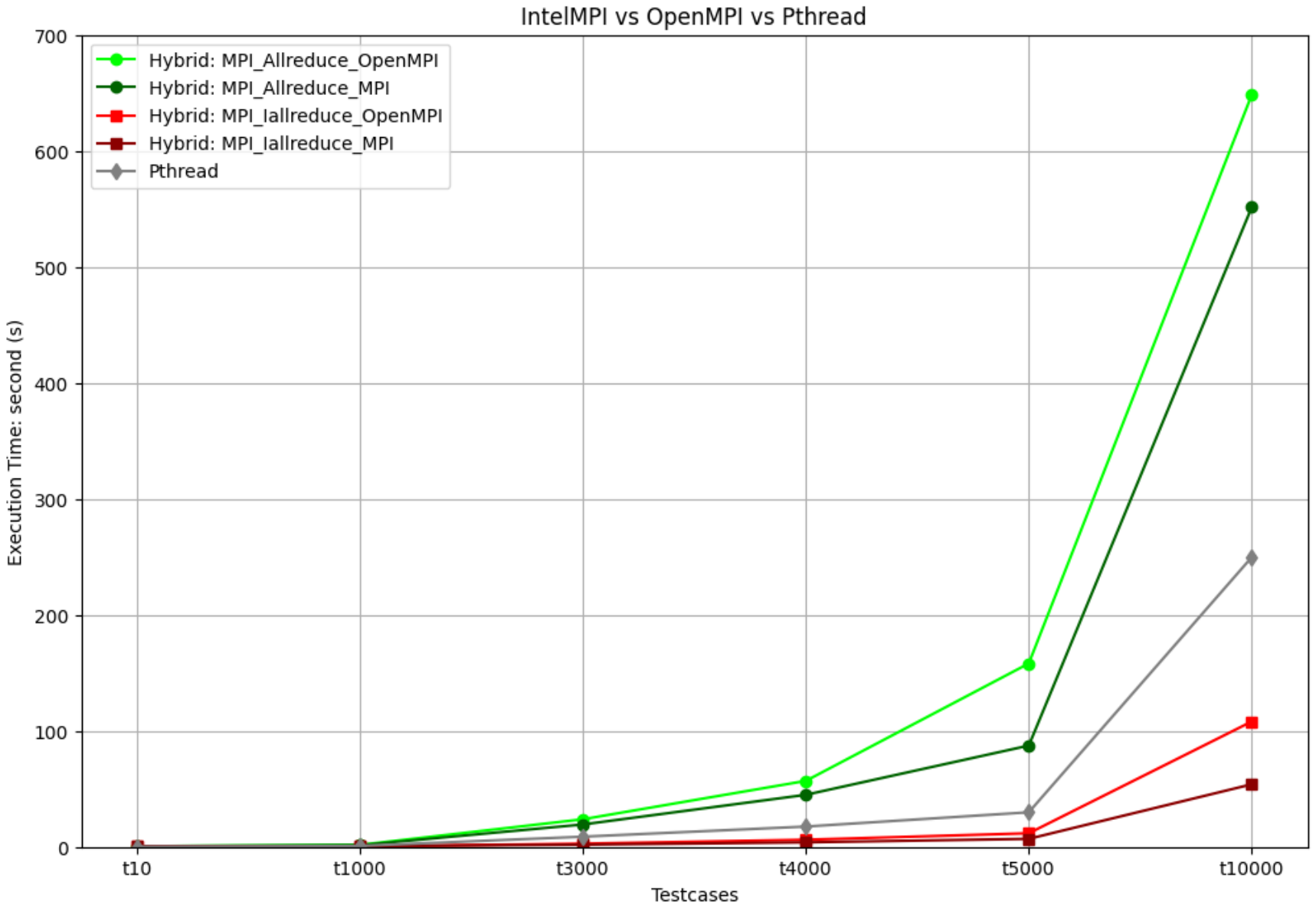
Hybrid vs Pthread: IntelMPI



Hybrid vs Pthread: OpenMPI



IntelMPI vs OpenMPI vs Pthread



CUDA vs Hybrid vs Pthread