Parallelization of ConTree: CPU Parallelism and GPU Depth-2 Acceleration

Mert Rodop, Ahmet Poyraz Güler, Kerem Tufan, Ege Dolmacı, Ömer Gölcük, Yiğit Özcelep

Supervised by: Kamer Kaya

mert.rodop@sabanciuniv.edu, poyraz.guler@sabanciuniv.edu, kerem.tufan@sabanciuniv.edu, ege.dolmaci@sabanciuniv.edu, omer.golcuk@sabanciuniv.edu, yigit.ozcelep@sabanciuniv.edu

January 15, 2026

Abstract

This report presents a parallel implementation of ConTree, an exact decision tree construction algorithm based on dynamic programming and branch-and-bound pruning. We accelerate the original solver on multi-core CPUs by exploiting feature-level parallelism at each node and task-level parallelism across independent left/right subproblems, while preserving correctness through synchronized updates of the shared upper bound and best solution. For depth-two subproblems, which form a dominant computational hotspot, we additionally leverage a specialized solver and support hybrid execution by offloading depth-two evaluation to the GPU. Performance is evaluated using Google Benchmark, reporting scalability across thread counts and speedups relative to a single-thread baseline, together with an analysis of dataset- and depth-dependent behavior. We also examine the impact of capping on runtime and accuracy, and observe that capping can improve performance without causing systematic accuracy degradation. Overall, the proposed CPU and GPU parallelization strategies significantly reduce end-to-end runtime while maintaining ConTree's optimization guarantees when the optimality gap is set to zero.

The implementation is publicly available at: `https://github.com/egedolmaci/contree`

1 Introduction

Decision trees are extremely popular in machine learning owing to their interpretability and their ability to capture nonlinear patterns. However, most greedy methods, such as CART, typically produce non-optimal trees. In contrast, Optimal Decision Trees, which minimize training error under a depth or size constraint, are NP-hard to compute and therefore computationally expensive.

ConTree is an exact framework for Optimal Decision Trees that avoids binarization by operating directly on continuous features and applying advanced pruning techniques such as Neighborhood Pruning, Interval Shrinking, and Sub-interval Pruning. It also includes a specialized depth-two solver to accelerate the evaluation of small subtrees. Despite these improvements, ConTree remains slow for large datasets and deeper trees because it must evaluate many candidate split points and repeatedly process sorted arrays.

In our project, we accelerate ConTree using a CPU–GPU hybrid design while preserving its optimality guarantees. CPU parallelism is used for high-level search, feature evaluation, and pruning, while the GPU is used to accelerate compute-intensive tasks such as split scoring and the depth-two subroutine. In addition to this hybrid system, we also evaluate a CPU-only parallelized version and a GPU-accelerated depth-two-only version, allowing us to separately measure the impact of CPU parallelism and GPU offloading on ConTree's performance.

1.1 Motivation

While ConTree calculates an optimal decision tree, its runtime becomes a bottleneck for larger datasets and deeper trees. Even with pruning, the branch-and-bound search must evaluate many candidate splits, and the number of subproblems grows rapidly as tree depth increases.

At the same time, modern CPUs and GPUs provide massive parallel computational power that a sequential ConTree implementation cannot fully utilize. Although many of ConTree's computations are parallelizable, the use of dynamic bounds, pruning, and recursive search makes parallel execution non-trivial.

This gap between ConTree's strong optimality guarantees and its practical runtime motivates this work. Our goal is to make ConTree more efficient by exploiting modern parallel hardware while preserving its correctness and optimality.

1.2 Problem Definition

Let $D = \{(x_i, y_i)\}_{i=1}^{n}$ be a labeled dataset, where each sample $x_i \in \mathbb{R}^m$ consists of m continuous-valued features and $y_i \in \{1, \ldots, C\}$ is its class label. Given a maximum tree depth d, the goal is to construct a binary decision tree T of depth at most d that minimizes the number of misclassified samples in D.

If $\ell(T, D)$ denotes the number of samples in D that are incorrectly classified by T, the optimization problem is

$$T^* = \arg\min_{T \in \mathcal{T}_d} \ \ell(T, D),$$

where $\mathcal{T}_d$ is the set of all binary decision trees of depth at most d, with internal nodes defined by threshold splits on continuous features and leaves predicting a single class.

ConTree solves this problem using dynamic programming with a branch and bound search and pruning techniques. In this project, we keep the same objective and search space, and focus on accelerating the computation of T^* using parallel CPU and GPU implementations while preserving correctness and optimality.

1.3 Contributions

- C1 **Bottleneck Analysis.** We analyze ConTree's branch-and-bound execution and identify feature-evaluation loops and the depth-two subtree solver as the main sources of computational cost.
- C2 **Hybrid CPU–GPU Architecture.** We design a hybrid parallel framework that combines CPU-parallel tree search with GPU-parallel depth-two split evaluation while preserving ConTree's optimality guarantees.
- C3 **CPU-Parallel and GPU-Accelerated Implementations.** We implement a fully CPU-parallelized version of ConTree using OpenMP and a GPU-accelerated depth-two solver based on bitmask kernels for efficient split scoring.
- C4 **Experimental Evaluation.** We evaluate sequential ConTree, CPU-parallel ConTree, GPU-accelerated depth-two ConTree, and the hybrid CPU–GPU system across multiple datasets and tree depths.

2 Background and Related Work

2.1 Optimal Decision Trees Overview

Optimal Decision Trees (ODTs) seek to discover the tree, out of all possible trees of a certain depth or size, that best minimizes the training error. Contrary to greedy approaches such as the popular CART or C4.5 algorithms, which choose their splitting candidates locally, ODTs tackle a global optimization task over the ensemble of possible trees (Breiman et al., 1984; Quinlan, 1993).

The computation of ODTs is NP-hard because the number of candidate trees increases exponentially with depth and the number of features (Bertsimas & Dunn, 2017). Although greedy approaches scale well, they may produce suboptimal trees. In contrast, exact techniques based on mixed-integer programming or SAT solvers guarantee optimality but do not scale to large datasets or deep trees (Narodytska et al., 2018).

More scalable exact solvers combine dynamic programming with branch-and-bound, which allows reusing solutions to subproblems and pruning large portions of the search space (Agli et al., 2022). This class of methods currently represents the most practical approach for computing optimal decision trees.

2.2 Continuous-feature Optimal Trees

Most optimal decision tree algorithms require binary features. When the input features are continuous, they are typically converted into binary form using thresholding or discretization.

Some approaches create a binary variable for every possible threshold of a continuous feature, but this leads to very poor scalability. Among the few methods that operate directly on continuous features, Quant-BnB is a notable example; however, it is limited to small tree depths.

ConTree overcomes these limitations by working directly with continuous feature values and combining dynamic programming with branch-and-bound. By integrating effective pruning techniques and a specialized depth-two solver, ConTree achieves improved scalability while still guaranteeing optimal solutions (Van der Linden et al., 2024).

3 ConTree Framework

3.1 DP Formulation

ConTree formulates the optimal decision tree problem using dynamic programming over dataset subsets and remaining tree depth. Let $CT(D,d)$ denote the minimum number of misclassified samples achievable by any decision tree of depth at most d on dataset D. The recursion is defined as

$$CT(D,d) = \begin{cases} \min_{\hat{y}\in\mathcal{Y}} \sum_{(x,y)\in D} \mathbf{1}(\hat{y} \neq y), & \text{if } d = 0, \\ \min_{f\in\mathcal{F}} \; \mathit{Branch}(D,d,f), & \text{if } d > 0, \end{cases}$$

where $\mathcal{F}$ is the set of features and $\mathcal{Y}$ is the set of class labels.

For a fixed feature f, the best split is found by evaluating all candidate thresholds:

$$\mathit{Branch}(D,d,f) = \min_{\tau\in\mathcal{S}_f} \; \mathit{Split}(D,d,f,\tau),$$

where $\mathcal{S}_f$ is the set of candidate thresholds for feature f, typically chosen as the midpoints between consecutive sorted feature values.

Each threshold τ divides the dataset into two subsets, and the split cost is computed as

$$\mathit{Split}(D,d,f,\tau) = CT(D(f\leq\tau), d-1) + CT(D(f>\tau), d-1).$$

This dynamic programming formulation exactly represents the optimal decision tree objective. ConTree combines it with branch-and-bound and pruning techniques to avoid evaluating unnecessary splits while preserving optimality.

3.2 Branch-and-Bound Structure

ConTree evaluates candidate splits defined by feature–threshold pairs (f,τ), each of which partitions the data into left and right subsets. The quality of a split is measured by the best trees that can be built on these two subsets:

$$\mathit{Split}(D,d,f,\tau) = CT(D_L, d-1) + CT(D_R, d-1),$$

where $D_L = D(f\leq\tau)$ and $D_R = D(f>\tau)$.

To reduce computation, ConTree uses an upper bound UB (best error found so far) and a lower bound LB (best possible error for a candidate). If $LB \geq UB$, the candidate split is pruned. This branch-and-bound strategy allows ConTree to efficiently explore the search space while still guaranteeing optimality.

3.3 Pruning Techniques (NB / IS / SP)

ConTree employs three pruning strategies: Neighborhood Pruning (NB), Interval Shrinking (IS), and Sub-interval Pruning (SP) to reduce the number of split candidates.

Neighborhood Pruning removes redundant thresholds in the neighborhood of an already evaluated split. Interval Shrinking discards entire ranges of threshold values that cannot improve the current best solution. Sub-interval Pruning further refines this process by eliminating smaller threshold ranges that are also guaranteed to be suboptimal.

For any threshold interval I, ConTree computes a lower bound $LB(I)$ on the best achievable misclassification error within that interval. If

$$LB(I) \geq UB,$$

where UB is the current best (upper bound) error, then all thresholds in I can be safely pruned. These pruning techniques significantly reduce the search space while preserving the optimality of the final decision tree.

3.4 Depth-2 Specialized Subroutine (D2Split)

ConTree has a special depth-two solver, called D2Split, for fast evaluation of small subtrees. Instead of applying the general dynamic programming procedure recursively, D2Split computes the best depth-two tree directly by exhaustively evaluating all possible root and child splits.

For a dataset D, D2Split selects the root split (f_0, τ_0) and the best left and right child splits (f_L, τ_L) and (f_R, τ_R) that jointly minimize the total classification error:

$$D2(D) = \min_{f_0,\tau_0} \Big(\min_{f_L,\tau_L} CT(D_L, 1) \; + \; \min_{f_R,\tau_R} CT(D_R, 1) \Big),$$

where $D_L = D(f_0 \leq \tau_0)$ and $D_R = D(f_0 > \tau_0)$.

By solving this depth-two problem in a single step, D2Split avoids recursive overhead and enables more effective pruning and bounding in the higher-level ConTree search.

3.5 Optimality Gap

ConTree can take advantage of an optimality gap to prematurely terminate the search when a near-optimal solution is sufficient. Even when the optimality gap is set to zero, ConTree still guarantees an optimal decision tree.

The search terminates when the difference between the current upper bound UB and lower bound LB satisfies

$$UB - LB \leq \varepsilon,$$

where ε is the user-defined optimality gap. When $\varepsilon = 0$, the returned tree is guaranteed to be globally optimal.

4 Shortcomings of Baseline ConTree and Our Framework

4.1 Observed Bottlenecks / Shortcomings

Practically speaking, the running time of ConTree is dominated by the evaluation of candidate splits in the branch-and-bound search. At each node, a large number of feature and threshold combinations must be examined, and for each candidate split the misclassification error must be computed over all active data points.

Another major bottleneck is the depth-two subroutine. Although it is faster than evaluating deeper trees recursively, it still requires evaluating all combinations of root and child splits, making it one of the most computationally expensive components of the algorithm.

Finally, the original ConTree implementation is largely sequential. Even though many feature evaluations and subtree computations are independent, they are processed one by one on the CPU, which prevents effective use of modern multicore CPUs and GPUs.

4.2 Design Principles

Our method is a CPU-level parallelization of ConTree that preserves the original branch-and-bound logic. Independent tasks, such as feature scoring and split candidate evaluation, are parallelized, while shared variables including bounds, best solutions, and pruning states are kept consistent.

OpenMP is used to parallelize feature-wise computations, where each thread evaluates candidates locally and reports a local best result. These results are combined through a controlled reduction step, followed by an update of the global upper bound. Pruning always uses the most up-to-date bound.

In addition, the depth-two subproblem is offloaded to the GPU, which efficiently evaluates all root and child split combinations using bitsets. The CPU manages recursion and pruning, while the GPU performs the most compute-intensive parts of the algorithm.

4.3 Algorithm Flow (System-Level)

$$\boxed{\begin{array}{c}\textbf{CPU ConTree recursion (Branch-and-Bound) and DP} \rightarrow \textbf{Check remaining} \\ \textbf{depth} \rightarrow \textbf{If}\ \ \text{d=2:} \begin{cases}\text{CPU-only: run parallelized D2Split (OpenMP)} \\ \text{Hybrid: call GPU depth-2 brute-force solver}\end{cases} \rightarrow \textbf{Decode} \\ \textbf{best split} \rightarrow \textbf{Update bounds \& apply pruning} \rightarrow \textbf{Continue recursion}\end{array}}$$

Figure 1: System-level flow of the framework (CPU-only vs. Hybrid depth-2 execution).

4.4 Implementation Details — CPU Side

4.4.1 Correctness Goal

Our CPU parallelization accelerates ConTree without changing its objective or pruning guarantees. Since pruning depends on a valid non-increasing upper bound (UB) and a consistent "best tree" state, we parallelize only computations that are independent by construction and synchronize every update that affects UB or the global best solution. This prevents race conditions that could otherwise lead to incorrect pruning decisions or unstable accuracy. Consequently, when the optimality gap is $\varepsilon = 0$, the CPU-parallel solver remains exact and returns a globally optimal tree.

4.4.2 DP Cache (Memoization) Integration

ConTree's dynamic programming component is implemented via memoization of subproblems (D, d), where D is the dataset slice at a node and d is the remaining depth. We preserve this behavior under parallel recursion by performing a cache lookup at the beginning of each call and returning immediately if a cached optimal solution exists. After solving a node, we store the result when it satisfies the active bound so future calls can reuse it. This reduces redundant computation across parallel branches while staying consistent with the original DP formulation.

```
if (Cache::global_cache.is_cached(dataview, solution_configuration.max_depth)) {
    current_optimal_decision_tree =
        Cache::global_cache.retrieve(dataview, solution_configuration.max_depth);
    return;
}

...

if (current_optimal_decision_tree->misclassification_score <= upper_bound) {
    Cache::global_cache.store(dataview, solution_configuration.max_depth,
                              current_optimal_decision_tree);
}
```

4.4.3 Feature-Level Parallelism in `GeneralSolver`

At each node (for remaining depth > 2), ConTree evaluates candidate splits across many features, which corresponds to the $\min_f$ portion of the recurrence. These evaluations are independent until they attempt to update the shared best solution, making the feature loop a natural OpenMP parallel region. We use `schedule(dynamic)` because the cost per feature is highly irregular due to pruning and varying recursion depth, which would otherwise cause load imbalance. We also guard the parallel loop with a dataset-size threshold to avoid OpenMP overhead on small nodes. Each thread computes a thread-local best tree for its assigned feature while using the current shared UB to guide pruning.

```
std::atomic<int> node_best_score(current_optimal_decision_tree->misclassification_score)
    ;
int num_features = dataview.get_feature_number();

#pragma omp parallel for schedule(dynamic) if(dataview.get_dataset_size() > 500)
for (int feature_nr = 0; feature_nr < num_features; feature_nr++) {
    int current_best = node_best_score.load(std::memory_order_relaxed);
    if (current_best == 0) continue;

    int feature_index = dataview.gini_values[feature_nr].second;
    std::shared_ptr<Tree> local_tree = std::make_shared<Tree>(-1, INT_MAX);

    create_optimal_decision_tree(dataview, solution_configuration,
                                 feature_index, local_tree,
                                 std::min(upper_bound, current_best),
                                 node_best_score);

    // global update is handled in the next subsection
}
```

4.4.4 Global Best Tree Update (UB Propagation)

Parallel feature evaluation requires a correct reduction from thread-local candidates to a single node-level best solution. A naive update can introduce races on the best-tree pointer and UB value, causing inconsistent bounds and potentially incorrect pruning. We address this by combining (i) an atomic "best score" variable for fast propagation of improved bounds across threads, and (ii) a `critical` section for safely updating the shared best-tree structure. Threads first check whether their local candidate is competitive before entering the critical region, minimizing synchronization overhead while ensuring that UB updates remain monotonic and globally visible.

```
if (local_tree->misclassification_score < INT_MAX &&
    local_tree->misclassification_score <=
        node_best_score.load(std::memory_order_relaxed)) {

    #pragma omp critical(update_tree)
    {
        if (local_tree->misclassification_score <
            current_optimal_decision_tree->misclassification_score) {

            current_optimal_decision_tree = local_tree;
            node_best_score.store(local_tree->misclassification_score,
                                  std::memory_order_relaxed);
        }
    }
}
```

4.4.5 Interval Pruning Uses Sibling Best Bound

Within the per-feature search, ConTree applies interval-based pruning (subinterval pruning and interval shrinking) to eliminate ranges of thresholds that cannot beat the current bound. Under parallel execution, improved bounds discovered by any thread should immediately strengthen pruning for the remaining threads at the same node. We therefore compute the pruning bound using the minimum of the thread-local best score and the shared atomic node-level best score. As soon as one thread finds a better candidate, other threads can prune more aggressively, reducing the number of explored intervals and improving overall scalability.

```
const auto& possible_split_indices = dataview.get_possible_split_indices(feature_index);
IntervalsPruner interval_pruner(possible_split_indices, (solution_configuration.max_gap
    + 1) / 2);

...

int sibling_best = parent_node_best_score.load(std::memory_order_relaxed);
int pruning_bound = std::min(current_optimal_decision_tree->misclassification_score,
                             sibling_best);

if (!solution_configuration.stopwatch.IsWithinTimeLimit()) return;

auto current_interval = unsearched_intervals.front(); unsearched_intervals.pop();

if (interval_pruner.subinterval_pruning(current_interval, pruning_bound)) continue;
interval_pruner.interval_shrinking(current_interval, pruning_bound);
```

4.4.6 Task-Level Parallelism for Left/Right Subtrees

For a fixed candidate split, the left and right child subproblems are independent recursive calls. We exploit this by spawning two OpenMP tasks and synchronizing them with `taskwait` before combining the child results. To keep task overhead bounded, task creation is conditional on subtree sizes; when subtrees are small, a sequential evaluation is faster and also tends to tighten UB earlier if the larger subtree is explored first. This task-level parallelism complements feature-level parallelism and increases concurrency when the recursion generates large subtrees. ize task overhead.

```
bool use_tasks = (left_dataview.get_dataset_size() > 200 &&
                  right_dataview.get_dataset_size() > 200);

if (use_tasks) {
    #pragma omp task shared(left_optimal_dt) \
            firstprivate(left_dataview, left_solution_configuration, left_ub)
    {
        GeneralSolver::create_optimal_decision_tree(left_dataview,
            left_solution_configuration, left_optimal_dt, left_ub);
    }

    #pragma omp task shared(right_optimal_dt) \
        firstprivate(right_dataview, right_solution_configuration, left_ub)
    {
        GeneralSolver::create_optimal_decision_tree(right_dataview,
            right_solution_configuration, right_optimal_dt, left_ub);
    }

    #pragma omp taskwait
} else {
    // sequential fallback for small subtrees
    ...
}
```

4.4.7 Specialized Depth-2 CPU Solver (`SpecializedSolver`)

When remaining depth reaches $d = 2$, the solver switches to the specialized depth-two routine instead of continuing general recursion. This path evaluates depth-2 trees efficiently using class-count-based computations and avoids expensive recursive expansion. We parallelize the outer loop over root features using OpenMP with dynamic scheduling, since different features can lead to different workloads depending on pruning and candidate thresholds. Each thread computes the best depth-2 candidate for a root feature and then competes to update the shared best solution under a critical section, following the same safe reduction pattern used in the general solver.

```
std::atomic<int> atomic_best_score(current_optimal_decision_tree->
    misclassification_score);
int num_features = dataview.get_feature_number();

#pragma omp parallel for schedule(dynamic)
for (int feature_index = 0; feature_index < num_features; feature_index++) {
    if (atomic_best_score.load(std::memory_order_relaxed) == 0) continue;
    if (!solution_configuration.stopwatch.IsWithinTimeLimit()) continue;

    std::shared_ptr<Tree> local_tree = std::make_shared<Tree>();
    int current_best = atomic_best_score.load(std::memory_order_relaxed);
    local_tree->misclassification_score = current_best;

    create_optimal_decision_tree(dataview, solution_configuration,
                                 feature_index, local_tree,
                                 std::min(upper_bound, current_best));

    if (local_tree->misclassification_score <
        atomic_best_score.load(std::memory_order_relaxed)) {
        #pragma omp critical(update_best_tree)
        {
            if (local_tree->misclassification_score <
                current_optimal_decision_tree->misclassification_score) {
                *current_optimal_decision_tree = *local_tree;
                atomic_best_score.store(current_optimal_decision_tree->
                    misclassification_score,
                                    std::memory_order_relaxed);
            }
        }
    }
}
```

4.4.8 Thread-Safe Cache Sharding

Cache operations must remain correct under parallel recursion. The implementation shards the hash-table per depth and uses a per-shard mutex for safe concurrent access.

```
int shard_idx = get_shard_idx(bitset);
CacheShard& shard = *_cache[depth][shard_idx];

std::lock_guard<std::mutex> lock(shard.mtx);
auto it = shard.table.find(bitset);
if (it != shard.table.end()) return it->second.solution;
```

4.5 Implementation Details — GPU Depth-2 Kernel

4.5.1 How it fits ConTree without breaking the objective

The GPU depth-2 kernel integrates into ConTree by accelerating the evaluation of candidate depth-two trees while preserving the exact misclassification objective used by the original CPU brute-force solver.

For a given node, ConTree must identify the depth-two decision tree (one root split and two child splits) that minimizes the total misclassification error over the active data rows. The GPU kernel evaluates all candidate root splits and, for each root, all candidate child splits for both the left and right subtrees. Leaf labels are assigned using majority voting, and errors are computed exactly in the same way as in the CPU implementation.

This behavior is reflected in the kernel launch configuration, where each thread block corresponds to one candidate root split:

The kernel is launched as:

```
solve_depth2_fast_kernel<<<global_num_candidates, 256, 0, g_stream>>>(
    d_candidate_masks, d_class_masks, d_active,
    global_num_candidates, global_num_words, global_num_classes,
    d_err, d_bL, d_bR, d_leafs
);
```

By computing the exact misclassification counts for all depth-two trees and returning the best split to the CPU, the GPU kernel preserves ConTree's objective while significantly reducing the cost of evaluating this critical subproblem.

4.5.2 Data representation

All data is represented using bitsets, which enables efficient parallel computation through bitwise operations. Each bit corresponds to a single data row, allowing many rows to be processed simultaneously within one machine word.

Candidate split masks encode the result of applying a feature threshold to each data row, where a bit value of 1 indicates that the row is sent to the left child and a value of 0 indicates that it is sent to the right child. The mask for a given root split is accessed as:

```
const uint32_t* root_ptr =
    &candidate_masks[root_idx * num_words];
```

Class membership is stored using one bitmask per class. For each word w, class counts are accumulated using population count instructions:

```
uint32_t cls = class_masks[k * num_words + w];
cnt_LL[k] += __popc(m_LL & cls);
```

An active row mask specifies which rows belong to the current node:

```
uint32_t v = active_row_mask[w];
if (v == 0) continue;
```

This representation avoids branching over individual rows, reduces memory usage, and enables fast counting through hardware-supported `__popc` operations.

4.5.3 Kernel mapping (high level)

The depth-two solver kernel maps naturally to the GPU execution model. Each CUDA block is assigned to evaluate one candidate root split, while threads within the block evaluate the candidate child splits in parallel.

This mapping is defined by the block and thread indices:

```
int root_idx = blockIdx.x;
for (int c_idx = threadIdx.x;
     c_idx < num_candidates;
     c_idx += blockDim.x) {
    // evaluate child split c_idx
}
```

For each root–child split pair, data rows are partitioned into four leaf regions using bitwise operations:

```
uint32_t m_LL = r & c & v;
uint32_t m_LR = r & (~c) & v;
uint32_t m_RL = (~r) & c & v;
uint32_t m_RR = (~r) & (~c) & v;
```

This formulation avoids floating-point computation and conditional branching inside the kernel, enabling efficient and fully parallel evaluation of split quality on the GPU.

4.5.4 Reduction strategy

Each thread maintains its own best candidate for the left and right subtrees:

```
int my_best_L = INT_MAX, my_idx_L = -1;
int my_best_R = INT_MAX, my_idx_R = -1;
```

Thread-local best results are first reduced at the warp level using shuffle-based reduction:

```
warpReduceMinWithIndex(my_best_L, my_idx_L, my_LL, my_LR);
warpReduceMinWithIndex(my_best_R, my_idx_R, my_RL, my_RR);
```

The warp-level winners are written to shared memory, and a second reduction step determines the best child split per subtree within each block:

```
if (lane == 0) {
    s_warp_best_L[warp_id] = my_best_L;
    s_warp_idx_L[warp_id] = my_idx_L;
}
```

Finally, two lightweight kernels perform block-level and global reductions to identify the best root split across all candidates. This hierarchical reduction strategy avoids the use of global atomic operations and ensures deterministic results while maintaining high performance.

4.5.5 Host-device synchronization & correctness

The host computes the active row mask and transfers it to the GPU using pinned memory and asynchronous copies:

```
CUDA_CHECK(cudaMemcpyAsync(
    d_active, h_active_pinned, active_bytes,
    cudaMemcpyHostToDevice, g_stream
));
```

After kernel execution and the reduction steps complete, the best split configuration is copied back to the host:

```
CUDA_CHECK(cudaMemcpyAsync(
    h_best_pinned, d_best_one, sizeof(BestPack),
    cudaMemcpyDeviceToHost, g_stream
));
```

Synchronization is performed by waiting on the CUDA stream before the CPU updates global bounds and continues the branch-and-bound search. This ensures that every GPU-evaluated depth-two solution is fully computed and correctly integrated into ConTree's decision process.

4.5.6 Capping / threshold budget

To avoid a combinatorial explosion in the number of candidate thresholds, the host applies a fixed cap on how many thresholds are considered per feature. Thresholds are selected uniformly from the sorted set of unique feature values:

```
int take = std::min(
    std::max(config.max_thresholds_per_feature, 1),
    usable
);
```

Only these selected thresholds are converted into candidate masks and transferred to the GPU. Although this restricts the search space, the GPU kernel still performs an exhaustive and exact evaluation over all provided candidates. As a result, the optimization objective is preserved within the capped candidate set.

4.5.7 CPU-side parallelization and GPU call management (high level)

In the CPU implementation, feature evaluation is parallelized using OpenMP so that multiple threads explore different splits concurrently:

```
#pragma omp parallel
#pragma omp for schedule(dynamic)
for (int feature_nr = 0; feature_nr < dataview.get_feature_number(); ++feature_nr)
     {
    ...
}
```

When a thread reaches a shallow subproblem suitable for GPU acceleration (e.g., depth ≤ 2), it may invoke the GPU solver from inside this parallel region. To keep this safe under multithreading, GPU kernel launches are explicitly controlled. If serialization is enabled, only one CPU thread can launch a kernel at a time:

```
#pragma omp critical(gpu_call)
{
  GPUBruteForceSolver::solve(dataview,
                             solution_configuration,
                                  current_optimal_decision_tree);
}
```

If serialization is disabled, multiple threads may launch GPU kernels concurrently, but each call operates on private (thread-local) GPU buffers and streams. In all cases, GPU results are merged back into the global solution only under CPU-side synchronization:

```
#pragma omp critical(update_tree)
{
  *current_optimal_decision_tree = *thread_local_best;
}
```

This ensures that CPU multithreading and GPU acceleration interact safely, while all shared-state updates (bounds, best tree, and pruning decisions) remain serialized on the CPU.

5 Experimental Evaluation

5.1 Experimental Setup

Goal. We evaluate the effectiveness of our OpenMP-based CPU parallelization and GPU offloading strategy for ConTree. The evaluation focuses on runtime scalability, speedup, and correctness in terms of classification accuracy and misclassification count. Our primary goal is to understand how different forms of parallelism affect feasibility and performance across tree depths.

Datasets and depths. We use 16 datasets (avila, bank, bean, bidding, eeg, fault, htru, magic, occupancy, page, raisin, rice, room, segment, skin, wilt), covering a wide range of instance counts, feature dimensions, and class distributions. Experiments are conducted for tree depths $d \in \{2, 3, 4\}$. Depth-5 runs are excluded from quantitative evaluation due to insufficient coverage and widespread timeouts caused by the exponential search space.

Platforms

- **Gandalf (CPU)**: OpenMP-based CPU parallelization results obtained from `openmp_findings.csv`.
- **Typhoon (GPU + CPU)**: Hybrid CPU–GPU runs obtained from `gpu_typhoon_capped_results.csv` and uncapped runs from `gpu_typhoon_uncapped_results.csv`.
- **Nebula (GPU)**: Limited snapshot GPU-only results from `gpu_nebula_capped_result_for_4_datasets.csv`, used for qualitative comparison.

Measurement methodology. All experiments are executed using **Google Benchmark**, which repeatedly runs each configuration and reports stable runtime statistics. Reported times correspond to the benchmark's measured runtime field. Measurements are consistent within each platform, and a global timeout of approximately 600 seconds is enforced.

Speedup definition. For CPU scaling experiments on Gandalf, speedup is defined relative to the single-thread baseline:

$$S(p) = \frac{T(1)}{T(p)}.$$

For CPU vs. GPU comparisons on Typhoon, we compare single-thread CPU execution against single-thread hybrid GPU execution to isolate architectural benefits.

5.2 Results

5.2.1 CPU Results (OpenMP on Gandalf)

Overall scaling behavior. Table 1 presents a high-level summary of OpenMP scalability across depths. At depth-2, parallelism yields limited speedup because the specialized solver already performs efficiently and the workload is too small to amortize OpenMP overhead. Depth-3 exhibits the strongest scalability due to the emergence of both feature-level and task-level parallelism in the recursive search. At depth-4, exponential growth dominates runtime behavior; parallelism often improves feasibility rather than raw scalability, and the optimal thread count decreases.

Table 1: Gandalf (CPU/OpenMP): scaling summary across depths.

Depth	Datasets	Geometric Mean Speedup	Median Best Threads
2	16	2.23	5
3	16	10.99	9
4	16	5.01	5

Depth-2 CPU performance. At depth-2, the search space consists of evaluating the root split and its two child splits. For small datasets, execution time is already in the millisecond range, making thread creation and synchronization overhead dominant. As a result, speedups are modest and saturate quickly. Medium-sized datasets benefit more from feature-level parallelism, but scaling stops once the number of threads exceeds the available independent work.

Table 2: Strongest CPU speedups at depth-2 (Gandalf).

Dataset	1T (s)	Best Threads	Best (s)	Speedup
fault	0.210	10	0.042	4.95
occupancy	0.218	4	0.053	4.08
segment	0.076	13	0.019	4.00
eeg	0.429	8	0.131	3.28
raisin	0.012	5	0.004	3.25

Depth-3 CPU performance. Depth-3 exposes substantial parallelism because each candidate root split spawns independent depth-2 subproblems. In addition to feature-level parallelism, task-level parallelism allows recursive branches to be explored concurrently. Datasets with large, balanced search trees (e.g., bidding, fault, bean) achieve order-of-magnitude speedups. Scaling eventually saturates due to shared bound updates, memory contention, and diminishing task granularity.

Table 3: Strongest CPU speedups at depth-3 (Gandalf).

Dataset	1T (s)	Best Threads	Best (s)	Speedup
bidding	14.22	10	0.39	36.66
fault	52.47	15	1.80	29.18
bean	124.15	8	6.11	20.33
eeg	214.72	9	11.54	18.60
segment	9.69	9	0.68	14.29

Depth-4 CPU performance. At depth-4, many datasets time out under single-thread execution due to exponential complexity. OpenMP parallelization can drastically reduce runtime or enable completion within the timeout budget. However, performance is highly sensitive to thread count: excessive parallelism increases synchronization overhead, cache thrashing, and load imbalance. Thus, small thread counts (4–9) often provide the best results.

Table 4: Depth-4 CPU results (datasets that finish).

Dataset	1T (s)	Best Threads	Best (s)	Speedup
bidding	417.29	9	2.74	152.1
room	296.58	5	9.79	30.3
wilt	179.65	4	7.01	25.6
raisin	232.88	4	24.05	9.68

5.2.2 GPU Results (Typhoon and Nebula)

Depth-2 GPU acceleration. Depth-2 evaluation is highly suitable for GPU execution because all candidate split combinations can be evaluated independently. Each GPU block processes a root split, while threads evaluate child splits using bitset operations and hardware population count. This leads to large speedups for medium and large datasets. For very small datasets, GPU kernel launch overhead dominates and can negate benefits.

Table 5: Typhoon: CPU vs GPU at depth-2 (threads=1).

Dataset	CPU (s)	GPU (s)	Speedup
bidding	0.216	0.008	27.0
occupancy	0.250	0.011	22.7
magic	0.702	0.036	19.5
htru	0.475	0.027	17.6
eeg	0.472	0.034	13.9

Depth-3 hybrid CPU–GPU performance. At depth-3, the CPU manages the branch-and-bound recursion while depth-2 subtrees are offloaded to the GPU. This hybrid strategy combines CPU flexibility with GPU throughput. The resulting speedups are substantial, particularly for datasets where depth-2 evaluation dominates runtime. In several cases, the GPU also discovers more accurate trees by exhaustively evaluating candidate splits that CPU pruning may discard.

Table 6: Typhoon: CPU vs Hybrid GPU at depth-3 (threads=1).

Dataset	CPU (s)	Hybrid (s)	Speedup
occupancy	21.89	0.46	47.5
htru	181.02	6.98	25.9
rice	19.22	0.84	22.9
magic	168.28	11.79	14.3
eeg	134.24	11.22	12.0

GPU capping effect. Capping is our way of implementing optimality gap. where we sacrifice accuracy to gain performence.

Table 7: Typhoon GPU capping effect (threads=1).

Depth	Median speedup	Mean speedup	Max speedup
2	25.3×	541×	3971×
3	42.2×	188×	957×

Nebula GPU snapshot. Nebula results are consistent with Typhoon trends but are limited in scope. They serve as qualitative confirmation rather than a primary evaluation source.

5.3 Discussion

Overall, CPU parallelization is most effective at depth-3, where recursive parallelism is abundant. GPU acceleration excels at depth-2 and significantly improves hybrid depth-3 execution. However, exponential complexity ultimately dominates at large depths, limiting scalability.

5.4 Limitations

- Depth-4 results are often timeout-limited; reported speedups may reflect feasibility rather than pure scaling.
- Depth-5 coverage is insufficient for quantitative analysis.
- Cross-platform absolute runtime comparisons are avoided.

6 Conclusion

This study demonstrates how the optimality of ConTree can be achieved through both CPU and GPU parallelization strategies. The CPU parallelization based on OpenMP helps in accelerating the branch-and-bound search process. The depth-two solver on the GPU helps in efficiently handling expensive split calculations. Both strategies are very effective in reducing the runtime and ensuring that the optimality of the ConTree solution for the optimization goal remains unaffected.

References

[1] Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). *Classification and Regression Trees.* Wadsworth.

[2] Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

[3] Bertsimas, D., & Dunn, J. (2017). Optimal classification trees. *Machine Learning*, 106(7), 1039–1082.

[4] Narodytska, N., Ignatiev, A., Pereira, F., & Marques-Silva, J. (2018). Learning optimal decision trees with SAT. In *Proceedings of IJCAI* (pp. 1362–1368).

[5] Agli, J.-B., Nijssen, S., & Schaus, P. (2022). Scalable optimal decision trees using dynamic programming and branch-and-bound. *Artificial Intelligence*, 304, 103665.

[6] Van der Linden, W., Nijssen, S., & Aho, T. (2024). ConTree: Optimal decision tree construction with continuous features using dynamic programming and branch-and-bound. *Artificial Intelligence*, 330, 103981.

7 Important AI Model Conversations Used in the Project

During the development of this project, several large language model (LLM)–based AI assistants were used as interactive research and engineering support tools. These models were primarily used to assist with algorithm understanding, CUDA kernel design, correctness verification, and LaTeX-based documentation.

To ensure transparency and reproducibility of the design process, we provide links to selected conversations that capture key stages of technical reasoning and system development:

- ChatGPT session (GPU kernel design, ConTree integration, and reduction strategies):
`https://chatgpt.com/share/69690852-1e90-8011-99a4-8442dc8b22bb`
- Claude session (algorithm structure, pruning, and high-level system reasoning):
`https://claude.ai/share/53b24e28-2d5e-4719-a2be-e1adbbb23dac`
- ChatGPT session (documentation, LaTeX, and algorithm presentation):
`https://chatgpt.com/share/69690f39-aad8-800b-9a4f-35bc24b7fa80`

These conversations document how the CPU parallelization strategy, GPU depth-two solver design, correctness preservation, and system-level integration were iteratively developed. They serve as supporting evidence of the engineering and reasoning process behind the final implementation.

Appendix

A Algorithms

Algorithm 1 GPU Depth-2 Solver Kernel (`solve_depth2_fast_kernel`)

```
1: procedure SolveDepth2FastKernel(candidate_masks, class_masks, active_mask,
   num_cands, num_words, num_classes)
2:     root_idx ← blockIdx.x
3:     root_ptr ← candidate_masks[root_idx]
   Initialize thread-local bests:
4:         bestL ← $\infty$, idxL ← $-1$, (clsLL, clsLR) ← $(-1, -1)$
5:         bestR ← $\infty$, idxR ← $-1$, (clsRL, clsRR) ← $(-1, -1)$
6:     for c_idx ← threadIdx.x; c_idx < num_cands; c_idx += blockDim.x do
7:         child_ptr ← candidate_masks[c_idx]
   Zero counters:
8:             cnt_LL, cnt_LR, cnt_RL, cnt_RR ∈ $\mathbb{Z}^{\texttt{num\_classes}}$
9:         for $w$ ← 0 to num_words − 1 do
10:            v ← active_mask[$w$]; if v = 0 continue
11:            r ← root_ptr[$w$], c ← child_ptr[$w$]
   Leaf masks:
12:                m_LL ← r & c & v,   m_LR ← r & ∼ c & v
13:                m_RL ←∼ r & c & v,   m_RR ←∼ r & ∼ c & v
14:            for $k$ ← 0 to num_classes − 1 do
15:                cls ← class_masks[$k, w$]
16:                cnt_LL[$k$] += popc(m_LL & cls)                    ▷ similarly LR/RL/RR
17:            end for
18:        end for
   Majority-vote errors:
19:            errL ← leafErr(cnt_LL) + leafErr(cnt_LR)
20:            errR ← leafErr(cnt_RL) + leafErr(cnt_RR)
   Update local bests for left/right (tie-break by smaller index)
21:    end for
   Warp-level reduction for left and right bests; write warp winners to shared memory
   Block-level reduction selects best child split per subtree
22:    if threadIdx.x = 0 then
23:        out_errors[root_idx] ← bestL + bestR
24:        out_best_L[root_idx] ← idxL;  out_best_R[root_idx] ← idxR
25:        out_leaf_classes[4 · root_idx + {0, 1, 2, 3}] ← (clsLL, clsLR, clsRL, clsRR)
26:    end if
27: end procedure

   Helper: leafErr(cnt) = $\sum_k$ cnt[$k$] − $\max_k$ cnt[$k$]; the argmax is the leaf label.
```

Algorithm 2 OpenMP-Parallel ConTree General Solver (Feature-Parallel + Task-Parallel Recursion)

```
Require: Dataset slice $D$, remaining depth $d$, configuration (gap $\varepsilon$, time limit), shared best tree $T^\star$, shared upper bound $UB$
Ensure: Updates $T^\star$ and $UB$ (node-level best solution)
 1: if $|D| = 0$ or $UB = 0$ then
 2:     return
 3: end if
 4: Compute best leaf for $D$ (majority label) and update local node score
 5: if $d = 0$ or stopping condition holds (gap/time limit/singleton) then
 6:     return
 7: end if
 8: if $d = 2$ then
 9:     Call specialized depth-2 solver (Algorithm 3)
10:     return
11: end if
12: Initialize shared node-level best score $S^\star \leftarrow UB$ (atomic)
13: for all features $f$ in parallel (#pragma omp parallel for schedule(dynamic)) do
14:     if time limit exceeded or $S^\star = 0$ then
15:         continue
16:     end if
17:     Compute best split for feature $f$ using interval pruning with bound $\min(UB, S^\star)$
18:     Let best split for $f$ be $(\tau, D_L, D_R)$ with candidate score $S_f$
19:     if $S_f$ improves bound then
20:         critical: update shared $T^\star$ and atomic $S^\star$
21:     end if
22: end for
23: return
```

Algorithm 3 OpenMP-Parallel Specialized Depth-2 Solver (CPU D2Split)

```
Require: Dataset slice $D$, configuration, shared best tree $T^\star$, shared upper bound $UB$
Ensure: Best depth-2 tree parameters and updates to $T^\star$, $UB$
 1: Initialize shared best score $S^\star \leftarrow UB$ (atomic)
 2: for all root features $f_1$ in parallel (#pragma omp parallel for schedule(dynamic)) do
 3:     if time limit exceeded or $S^\star = 0$ then
 4:         continue
 5:     end if
 6:     For each candidate root threshold $\tau_1$ on $f_1$, partition $D$ into $D_L, D_R$
 7:     Evaluate the best depth-2 completion by scanning candidate child splits and maintaining class counts
 8:     Compute misclassification score
            $$S = \sum_{Q \in \{LL, LR, RL, RR\}} \Big( |D_Q| - \max_c \mathrm{count}_{D_Q}(c) \Big)$$
 9:     if $S < S^\star$ then
10:         critical: update $T^\star$ and atomic $S^\star$
11:     end if
12: end for
13: return
```

B How to Run the Code

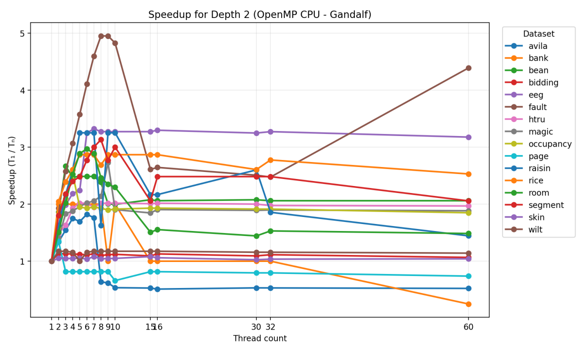
B.1 CPU Build and Run

```
# Clone CPU-only branch
git clone -b final-cpu --single-branch https://github.com/egedolmaci/contree.git
cd contree/code

# Build
mkdir build
cd build
cmake ..
make -j

# Run (example: Bean dataset, depth 3)
OMP_NUM_THREADS=8 ./build/ConTree \
  -file ../datasets/bean.txt \
  -max-depth 3
```

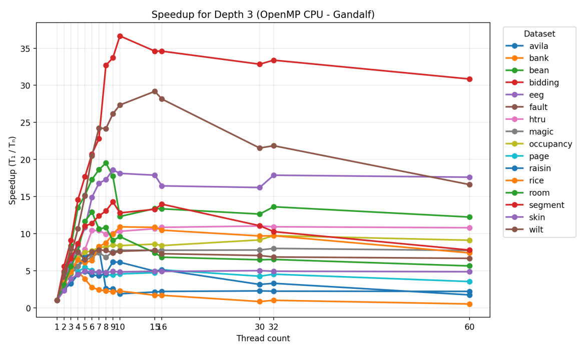
B.2 GPU Build and Run

```
# Clone GPU-enabled branch
git clone -b final-gpu --single-branch https://github.com/egedolmaci/contree.git
cd contree/code

# Build with CUDA enabled
mkdir build
cd build
cmake .. -DENABLE_CUDA=1
make -j

# Run with GPU depth-2 solver enabled
OMP_NUM_THREADS=1 ./build/ConTree \
  -file ../datasets/bean.txt \
  -max-depth 2 \
  -use-gpu-bruteforce 1
```

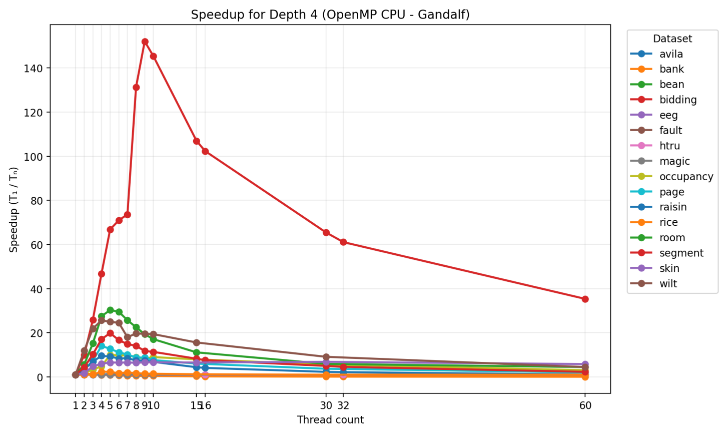
C Extra Tables / Figures



(a) Depth = 2

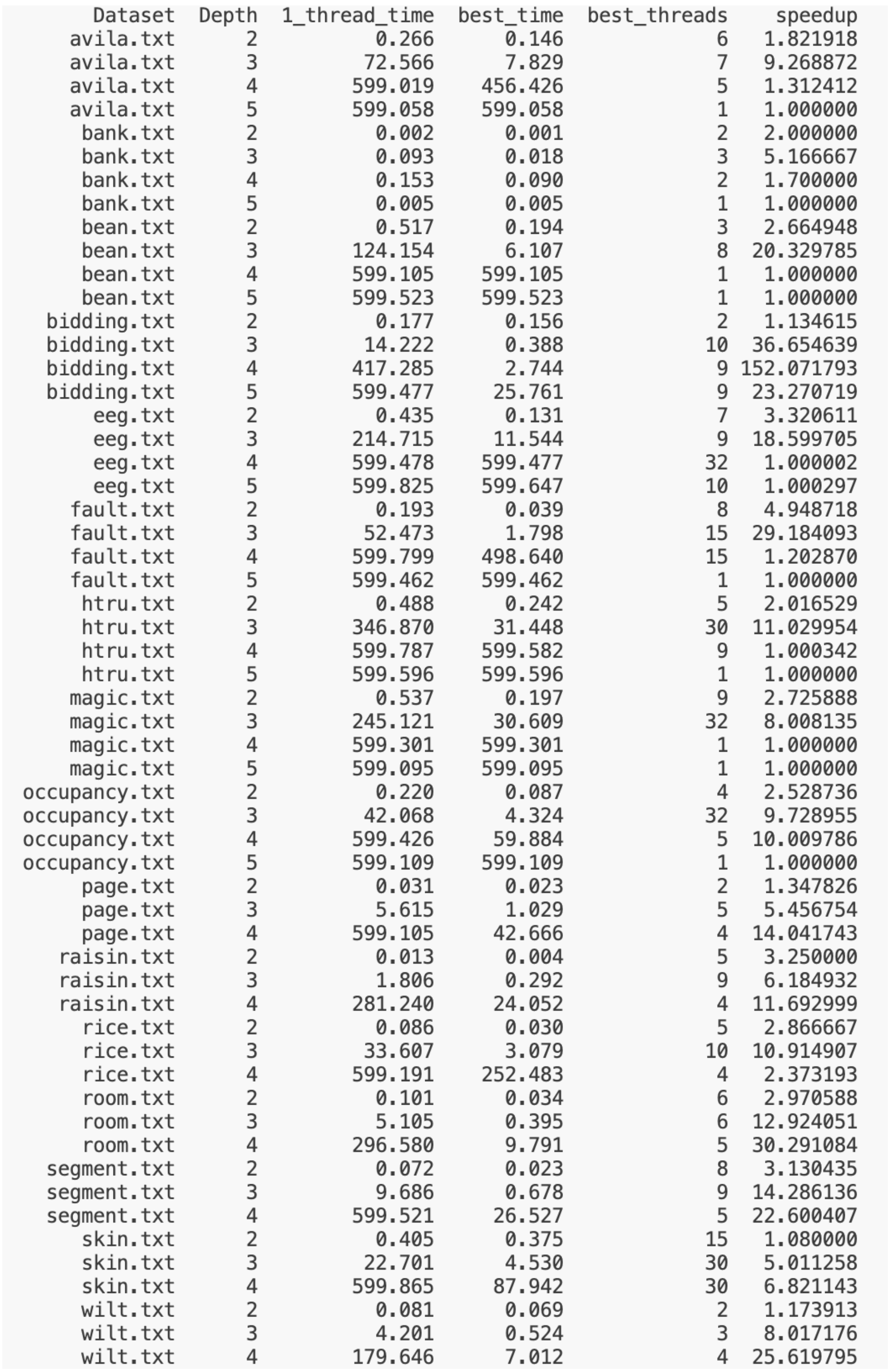


(b) Depth = 3



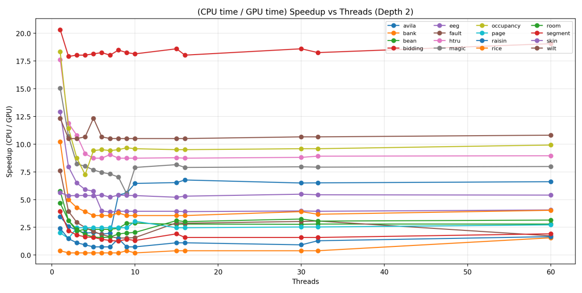
(c) Depth = 4

Figure 2: OpenMP CPU speedup results on Gandalf across different tree depths. As the depth increases, the workload characteristics and speedup variance change.

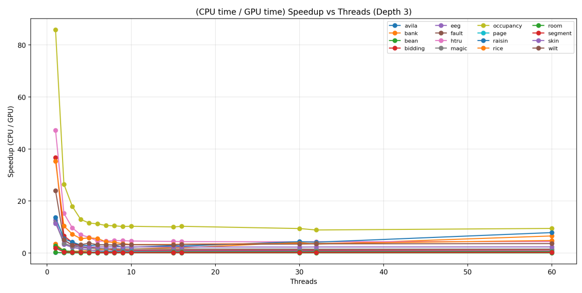


```
        Dataset  Depth  1_thread_time  best_time  best_threads     speedup
      avila.txt      2          0.266      0.146             6    1.821918
      avila.txt      3         72.566      7.829             7    9.268872
      avila.txt      4        599.019    456.426             5    1.312412
      avila.txt      5        599.058    599.058             1    1.000000
       bank.txt      2          0.002      0.001             2    2.000000
       bank.txt      3          0.093      0.018             3    5.166667
       bank.txt      4          0.153      0.090             2    1.700000
       bank.txt      5          0.005      0.005             1    1.000000
       bean.txt      2          0.517      0.194             3    2.664948
       bean.txt      3        124.154      6.107             8   20.329785
       bean.txt      4        599.105    599.105             1    1.000000
       bean.txt      5        599.523    599.523             1    1.000000
    bidding.txt      2          0.177      0.156             2    1.134615
    bidding.txt      3         14.222      0.388            10   36.654639
    bidding.txt      4        417.285      2.744             9  152.071793
    bidding.txt      5        599.477     25.761             9   23.270719
        eeg.txt      2          0.435      0.131             7    3.320611
        eeg.txt      3        214.715     11.544             9   18.599705
        eeg.txt      4        599.478    599.477            32    1.000002
        eeg.txt      5        599.825    599.647            10    1.000297
      fault.txt      2          0.193      0.039             8    4.948718
      fault.txt      3         52.473      1.798            15   29.184093
      fault.txt      4        599.799    498.640            15    1.202870
      fault.txt      5        599.462    599.462             1    1.000000
       htru.txt      2          0.488      0.242             5    2.016529
       htru.txt      3        346.870     31.448            30   11.029954
       htru.txt      4        599.787    599.582             9    1.000342
       htru.txt      5        599.596    599.596             1    1.000000
      magic.txt      2          0.537      0.197             9    2.725888
      magic.txt      3        245.121     30.609            32    8.008135
      magic.txt      4        599.301    599.301             1    1.000000
      magic.txt      5        599.095    599.095             1    1.000000
  occupancy.txt      2          0.220      0.087             4    2.528736
  occupancy.txt      3         42.068      4.324            32    9.728955
  occupancy.txt      4        599.426     59.884             5   10.009786
  occupancy.txt      5        599.109    599.109             1    1.000000
       page.txt      2          0.031      0.023             2    1.347826
       page.txt      3          5.615      1.029             5    5.456754
       page.txt      4        599.105     42.666             4   14.041743
     raisin.txt      2          0.013      0.004             5    3.250000
     raisin.txt      3          1.806      0.292             9    6.184932
     raisin.txt      4        281.240     24.052             4   11.692999
       rice.txt      2          0.086      0.030             5    2.866667
       rice.txt      3         33.607      3.079            10   10.914907
       rice.txt      4        599.191    252.483             4    2.373193
       room.txt      2          0.101      0.034             6    2.970588
       room.txt      3          5.105      0.395             6   12.924051
       room.txt      4        296.580      9.791             5   30.291084
    segment.txt      2          0.072      0.023             8    3.130435
    segment.txt      3          9.686      0.678             9   14.286136
    segment.txt      4        599.521     26.527             5   22.600407
       skin.txt      2          0.405      0.375            15    1.080000
       skin.txt      3         22.701      4.530            30    5.011258
       skin.txt      4        599.865     87.942            30    6.821143
       wilt.txt      2          0.081      0.069             2    1.173913
       wilt.txt      3          4.201      0.524             3    8.017176
       wilt.txt      4        179.646      7.012             4   25.619795
```

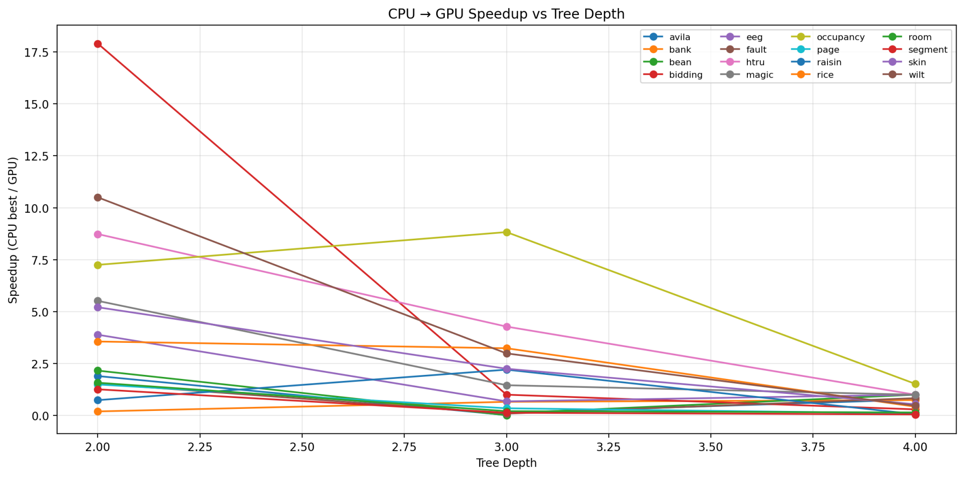
Table 8: Summary of single-thread execution times, best execution times, optimal thread counts, and achieved speedups for all datasets and tree depths on the OpenMP CPU (Gandalf).



(a) Speedup vs. Threads (Depth 2)



(b) Speedup vs. Threads (Depth 3)



(c) Impact of Tree Depth on Speedup

Figure 3: Performance comparison between CPU and GPU implementations. (a) and (b) illustrate the speedup factor relative to the number of threads for tree depths of 2 and 3, respectively. (c) shows the decline in GPU speedup advantage as the tree depth increases from 2 to 4 across various datasets.