Progress Report Proposal

Ahmet Poyraz Güler 38039 Ege Dolmacı 31263 Mert Rodop 37997

Ömer Gölcük 38101 Yiğit Özcelep 29315 Kerem Tufan 32554

December 8, 2025

---

1 Introduction

Due to their interpretability and ability to identify non-linear patterns, decision trees are widely used in machine learning.Scalable heuristics like CART (Classification and Regression Trees), a greedy top-down algorithm, frequently yield suboptimal results because building an Optimal Decision Tree (ODT)-one that minimizes training error under a size constraint is- NP-hard.

ConTree is a new exact ODT algorithm that employs an effective depth-two solver and robust pruning techniques (Neighborhood Pruning, Interval Shrinking, and Sub-interval Pruning) to avoid binarizing continuous data. Although the search space is greatly reduced by these methods, the algorithm is still computationally demanding for deeper trees and larger datasets.The high level search requires evaluating each candidate split independently, and the repeated numerical operations over sorted arrays which are most consuming as the dataset grows, are the main performance bottlenecks.

This project uses parallelization on multicore CPUs and GPUs to speed up ConTree while maintaining its optimality guarantees in order to overcome these constraints. OpenMP reduces branching decision delays by distributing root-level and subtree evaluations among CPU threads. Massively parallel numerical workloads, like split scoring and the depth-two subroutine, are offloaded to the GPU in a CPU as Manager, GPU as Worker model enabling thousands of operations to run simultaneously while the CPU manages irregular pruning logic to prevent warp divergence. The goal of this hybrid design is to greatly increase ConTree's speed and scalability without changing its fundamental structure.

---

2 Roadmap

2.1 Completed Steps

- Read and understand the core DP + BnB formulation and the novel ideas of the paper.

• Analyzed pruning techniques (NB, IS, SP) and identified parts of the program that are embarrassingly parallel.

• Profiled the baseline ConTree implementation with and without the Depth-2 solver parallelized.

2.2 Next Steps

• Implement OpenMP parallelization for the Branch procedure.

• Design a GPU kernel for the Depth-2 specialized solver subroutine offloaded to the GPU.

• Try offloading feature-class misclassification calculations to the GPU and returning all possible values in a memoization fashion.

• Understand heterogeneous design basics and how to utilize CPU–GPU memory transfers.

• Optimize memory layout for sorted feature arrays.

• Perform scalability benchmarks (depth 2–4).

2.3 Timeline

• Week 1 - 2: Implement OpenMP parallelization for the Branch procedure and validate correctness.

• Week 2 - 3: Develop CUDA kernels for the Depth-2 solver and feature-class misclassification, integrate GPU offloading, and debug.

• Week 3 - 4: Optimize memory layout, refine CPU–GPU data transfers, and run scalability benchmarks (depth 2–4).

• Week 5: Write the final report and prepare the presentation.

---

3 Ideas and Approaches

Ideas and Approaches for Parallelizing the Depth-2 (D2) Specialized Solver

The specialized solver for maximum tree depth two (referred to as the *D2 solver*) represents the computationally dominant phase of the overall decision-tree construction pipeline. The source code reveals a clear three-level nested structure:

Root Feature Enumeration:

The solver evaluates every feature as a potential root split

```
void SpecializedSolver::create_optimal_decision_tree(const Dataview& dataview, const Configuration& solution_configuration, std::shared_ptr<Tree>& current_optimal_decision_tree, int upper_bound) {
    for (int feature_index = 0; feature_index < dataview.get_feature_number(); feature_index++) {
        create_optimal_decision_tree(dataview, solution_configuration, feature_index, current_optimal_decision_tree, std::min(upper_bound, current_optimal_decision_tree->misclassification_score));

        if (current_optimal_decision_tree->misclassification_score <= solution_configuration.max_gap) {
            return;
        }
    }
}
```

Figure 1: Relevant Code Section from specialized_solver.cpp

Threshold Exploration for the Selected Root Feature: For each candidate root feature, a series of threshold intervals is explored using pruning techniques (IntervalsPruner) and a queue of Bound objects.

Depth-1 Optimization (Left and Right Subtrees): For each subtree produced by the root split, the solver re-examines all features to identify the optimal depth-1 threshold. This inner loop invokes process_depth_one_feature, which performs a sequential scan over sorted data and maintains cumulative frequency counts.

Parallelization Opportunity 1: Root Feature Parallelism (Coarse-Grained)

The outermost loop of the depth-2 specialized solver iterates over every feature as a candidate root split. A key observation is that these evaluations are *embarrassingly parallel*: each feature produces an independent temporary tree, computes its own misclassification score, and only at the end competes to update the shared global best solution. Because no intermediate state is shared during feature evaluation, the loop exhibits coarse-grained parallelism and is therefore highly suitable for OpenMP. The root-feature loop is parallelized using a “parallel for” construct, where each thread explores a distinct feature and maintains a private upper bound and local best tree. A final reduction step merges per-thread results to update the global optimum. The simplified OpenMP structure is shown below:

```
#pragma omp parallel
{
    std::shared_ptr<Tree> thread_best_tree;
    int thread_best_score = initial_best_score;

    #pragma omp for schedule(dynamic)
    for (int f = 0; f < feature_number; ++f) {
        auto local_tree = std::make_shared<Tree>(*
            initial_tree);
        int local_ub = upper_bound;

        create_optimal_decision_tree(dataview, config, f,
                                     local_tree, local_ub);

        if (local_tree->misclassification_score <
            thread_best_score) {
            thread_best_score = local_tree->
                misclassification_score;
            thread_best_tree  = local_tree;
        }
    }

    #pragma omp critical
    {
        if (thread_best_tree &&
            thread_best_score <
                current_optimal_decision_tree->
                misclassification_score)
        {
            current_optimal_decision_tree = thread_best_tree
                ;
        }
    }
}
```

Parallelization of the create_optimial_decision_tree

Further Parallelization Opportunities Beyond the D2 solver

While this work focuses on parallelizing the depth-2 (D2) specialized solver, the broader ConTree algorithm contains additional components that can potentially benefit from parallelism. The most evident opportunity arises in the general CT(D,d) recursion, where the algorithm evaluates every feature as a candidate split at deeper levels (d > 2). These feature evaluations are largely independent and could be parallelized using an OpenMP parallel for loop in a manner similar to the D2 root-feature parallelism. This would distribute the per-feature Branch() computations across threads and reduce the time spent in deeper levels of the decision tree search.

A second opportunity involves subtree parallelism. When a candidate split (f,w) is evaluated, the algorithm recursively computes CT(D_left,d-1) and CT(D_right,d-1). Since the left and right subtrees are independent, they could be executed as separate OpenMP tasks. Such task-based parallelism is especially promising at higher depths where subtrees are large, though it requires careful synchronization to avoid contention on shared structures such as the dynamic-programming cache and the global upper bound (UB). A thread-safe caching mechanism and atomic UB updates would be necessary to ensure correctness. Finally, the

interval-pruning logic inside Branch() also exposes fine-grained work units (evaluations of midpoints or candidate split points). These operations are individually small, so parallelizing them may not yield significant gains unless combined with larger-scale parallelism at the CT or subtree level.

In summary, although this report implements parallelism only for the D2 solver, multiple layers of the ConTree algorithm present meaningful opportunities for future parallelization. Extending parallelism into CT(), Branch() and subtree evaluation could further reduce runtime, especially for large datasets and deeper search depths.

4 GPU-CPU codesign

While OpenMP can exploit coarse-grained parallelism over features at the root of the Branch routine, the main numeric bottlenecks of ConTree scanning sorted thresholds and computing misclassification scores are data-parallel operations that map well to GPU architectures. To address scalability limits in deeper trees where sequential control flow dominates, we adopt a hybrid CPU–GPU design.

Core Design: CPU-as-Manager, GPU-as-Worker We retain the dynamic programming recursion, branch-and-bound search, and pruning (Neighborhood Pruning, Interval Shrinking, Sub-interval Pruning)
on the CPU, to avoid warp divergence caused by irregular control flow and dynamic data structures. The GPU is used exclusively as a high-throughput numeric accelerator for regular, linear computations:

GPU-Resident Depth-2 Solver:

ConTree's frequently used depth-2 subroutine is also implemented in a GPU-resident variant. Thread blocks are assigned to either features or (root-feature, root-split) pairs, and shared memory is used to maintain class-count accumulators while traversing the sorted data. The kernel mirrors the CPU logic but keeps all computation on the device, eliminating CPU–GPU transfers for this hotspot. At runtime, the system selects between the OpenMP and GPU depth-2 solvers based on dataset size, batch size, and available hardware.

Batched Split Scoring:

The original Branch procedure evaluates candidate thresholds for each feature one at a time. We replace this with a batched split scorer: after CPU-side pruning (NB, IS, SP) identifies the surviving candidate indices, the CPU collects them into a buffer and launches a single CUDA kernel. Each thread or warp processes one candidate (f, w), computing the left and right class counts and the corresponding misclassification score. This retains the original algorithmic behavior while parallelizing split evaluation across thousands of candidates, substantially reducing the wall-clock time of the Branch step.

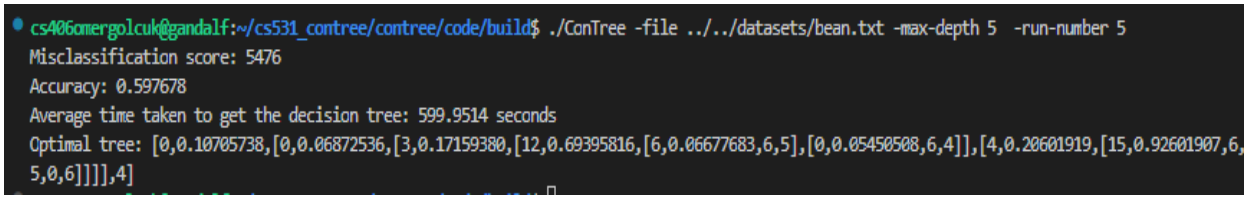
One other thing to take into account while writing a batched split scoring kernel is data layout. The baseline C++ implementation stores the dataset in an array of structures layout, which works well for CPU caches but results in non-coalesced accesses on GPUs. To make the GPU kernels efficient, the dataset is flattened into a structure-of-arrays layout in device memory. feature_values[f][i], sorted_indices[f][i], and labels[i] are stored as contiguous arrays. Each subproblem in the search is described only by index ranges within these arrays, eliminating the need for data copies. This layout enables coalesced global memory reads during split evaluation and requires transferring the dataset to the GPU only once during initialization.

To further reduce per-candidate work, we precompute per-feature, per-class prefix sums on the GPU using scan primitives. The resulting tensor PrefixCounts[f][c][i] records, for each feature, class, and position, how many samples of that class appear in the first i entries when sorted by that feature. During batched split scoring, the kernel obtains left and right class counts for a split index w through constant-time lookups, and subproblems are handled by restricting to the appropriate index ranges. Prefix sums are constructed once per dataset and reused throughout all subsequent Branch calls.

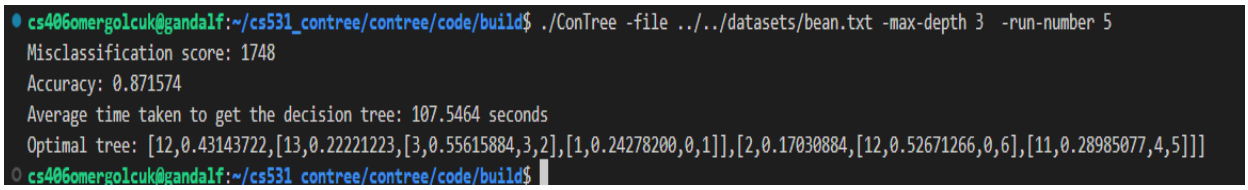
---

5 Preliminary Results

5.1 Baseline Performance



ConTree Performance on bean.txt with no parallelization (max-depth: 5, run number: 5)



ConTree Performance on bean.txt with no parallelization (max-depth: 3, run number: 5)

```
cs406omergolcuk@gandalf:~/cs531_contree/contree/code/build$ ./ConTree -file ../../datasets/bean.txt -max-depth 2  -run-number 5
Misclassification score: 4608
Accuracy: 0.66145
Average time taken to get the decision tree: 0.5246 seconds
Optimal tree: [2,0.17494383,[12,0.52589989,0,6],[11,0.29086977,4,5]]
```

ConTree Performance on bean.txt with no parallelization (max-depth: 2, run number: 5)

```
cs406omergolcuk@gandalf:~/cs531_contree/contree/code/build$ ./ConTree -file ../../datasets/page.txt -max-depth 5  -run-number 1
Misclassification score: 145
Accuracy: 0.973506
Average time taken to get the decision tree: 599.6750 seconds
Optimal tree: [0,0.07293724,[2,0.00349201,[3,0.01175704,[4,0.11189198,[3,0.00471459,3,0],[3,0.00052636,2,0]],[0,0.00072254,[6,0.00015
],3]
```

ConTree Performance on page.txt with no parallelization (max-depth: 5, run number: 1)

```
cs406omergolcuk@gandalf:~/cs531_contree/contree/code/build$ ./ConTree -file ../../datasets/bank.txt -max-depth 5  -run-number 1
Misclassification score: 0
Accuracy: 1
Average time taken to get the decision tree: 0.0060 seconds
Optimal tree: [0,0.54523432,[0,0.37803245,[0,0.29772055,[0,0.15822196,1,[1,0.81267625,1,0]],[0,0.30659604,[0,0.30630040,1,0],[1,0.7
0,0.57816118,[2,0.25990176,[1,0.73595732,1,0],[3,0.85666049,0,1]],[1,0.66253054,[2,0.16961613,1,0],[2,0.04901433,1,0]]],0]]
cs406omergolcuk@gandalf:~/cs531_contree/contree/code/build$
```

ConTree Performance on bank.txt with no parallelization (max-depth: 5, run number: 1)

5.2 Results After Parallelization of The Depth 2 Routine with OpenMP:

First Parallelized Region: SpecializedSolver:create_optimal_decision_tree

8 threads are used for the results below:

```
1   nohup: ignoring input
2   ========== BEAN.TXT | depth=5 | run=5 ==========
3   Misclassification score: 5474
4   Accuracy: 0.597825
5   Average time taken to get the decision tree: 599.9052 seconds
6   Optimal tree: [0,0.10705738,[0,0.07776086,[2,0.13772041,[12,0.55356789,[1
```

ConTree Performance on bean.txt with parallelization of create_optimal_decision_tree (max-depth: 5, run number: 5)

```
10  ========== BEAN.TXT | depth=3 | run=5 ==========
11  Misclassification score: 1757
12  Accuracy: 0.870913
13  Average time taken to get the decision tree: 27.3302 seconds
14  Optimal tree: [12,0.43522245,[13,0.22221223,[3,0.55615884,3,2],[1,0.24278200,0,1]],[2,0.17030884,[12,0.52671266,0,6],[11,0.28985077,4,5]]]
```

ConTree Performance on bean.txt with parallelization of create_optimal_decision_tree
(max-depth: 3, run number: 5)

```
18  ========== BEAN.TXT | depth=2 | run=5 ==========
19  Misclassification score: 4608
20  Accuracy: 0.66145
21  Average time taken to get the decision tree: 0.4472 seconds
22  Optimal tree: [2,0.17030884,[12,0.52671266,0,6],[11,0.29086977,4,5]]
```

ConTree Performance on bean.txt with parallelization of create_optimal_decision_tree
(max-depth: 2, run number: 5)

```
26  ========== PAGE.TXT | depth=5 | run=1 ==========
27  Misclassification score: 144
28  Accuracy: 0.973689
29  Average time taken to get the decision tree: 599.5510 seconds
30  Optimal tree: [0,0.07293724,[3,0.00536810,[4,0.15355031,[9,0.0015011
```

ConTree Performance on page.txt with parallelization of create_optimal_decision_tree
(max-depth: 5, run number: 1)

```
34  ========== BANK.TXT | depth=5 | run=1 ==========
35  Misclassification score: 0
36  Accuracy: 1
37  Average time taken to get the decision tree: 0.0050 seconds
38  Optimal tree: [0,0.54523432,[0,0.37803245,[0,0.29772055,[2,0.26584345,1,[1
```

ConTree Performance on bank.txt with parallelization of create_optimal_decision_tree
(max-depth: 5, run number: 1)

Dataset	Max-Depth	Run-number	Execution Time (base-line) Seconds (s)	Execution Time (parallelized version) Seconds (s)	Speed -up
bean.txt	5	5	599.9514	599.9052	~1.00
bean.txt	3	5	107.5464	27.3302	~3.94
bean.txt	2	5	0.5246	0.4472	~1.17
page.txt	5	1	599.6750	599.5510	~1.00

bank.txt	5	1	0.0060	0.0050	1.20

Interpretation on results:
In the code, All CT/Branch recursion and DP cache logic are still sequential. Total runtime consists of:

$$T_{total} = T_{D2(parallelized)} + T_{rest(sequential)}$$

We evaluated the OpenMP parallelization of the depth-2 specialized solver (D2Split) on three datasets and different maximum depths. For bean.txt with max-depth = 3 the total runtime decreases from 107.5 s to 27.3 s, corresponding to an overall speed-up of ≈3.9×. This indicates that, in this configuration, the D2 solver accounts for the majority of the runtime, so parallelizing it is highly effective.

In contrast, for max-depth = 5 on both bean.txt and page.txt the total runtime remains around 600 s in both the baseline and parallel versions (speed-up ≈1×). At this depth the search is dominated by the general CT/Branch recursion and pruning logic, which remain sequential in our implementation. The fraction of time spent in the parallel D2 solver is therefore small, and Amdahl's law limits the overall speed-up. For very shallow trees and very small datasets (e.g., bean.txt with max-depth = 2 or bank.txt with max-depth = 5), the absolute runtimes are below one second or a few milliseconds. In these cases, OpenMP startup and synchronization overheads become comparable with the useful work, so the observed speed-ups (~1.1 -- 1.2×) are mostly noise and not indicative of scalable parallelism.

Overall, our results show that parallelizing only the D2Split routine can provide substantial gains when depth-2 computations dominate the runtime (bean, depth 3), but has negligible impact when the workload is controlled by deeper CT/Branch recursion. To obtain speed-up for deeper trees (max-depth ≥ 5), further parallelization of the CT/Branch layer and/or subtree evaluation would be necessary.

5.3 Results After Parallelization of the General Solver with OpenMP

Second Parallelized Region: GeneralSolver::create_optimal_decision_tree (Feature-Level Parallelism)

Threads	Execution Time (s)	Speed-up	Efficiency
1	120.26	1.00×	100%

2	89.61	1.34×	67%
4	66.48	1.81×	45%
8	63.35	1.90×	24%

Scalability Analysis: bean.txt at Depth 3

Configuration: 5 runs per test, averaged

Observations:

- Moderate speedup of 1.90× with 8 threads at depth 3
- Diminishing returns beyond 4 threads due to limited parallelism (bean.txt has 16 features) and synchronization overhead.

Threads	**Execution Time (s)**	**Speed-up**
1	599.50 (near timeout)	1.00×
8	600.30 (timeout)	1.00×

Depth 4 Results: bean.txt

Configuration: 1 run per test (depth 4 is significantly slower)

Interpretation:

At depth 4, the search hits the 600-second timeout in both serial and parallel versions. The workload involves exponentially growing search space. To achieve speedup at depth ≥ 4, further parallelization strategies (e.g., subtree-level task parallelism) would be needed.

Additional Datasets at Depth 3

Dataset	**Threads**	**Execution Time (s)**	**Speed-up**
page.txt	1	7.67	1.00×
page.txt	8	4.18	1.84×
bank.txt	1	0.093	1.00×
bank.txt	8	0.065	1.43×

Observations:

- page.txt (medium dataset) shows comparable speedup (1.84×) to bean.txt, confirming that general solver parallelization is effective for depth 3.
- bank.txt (tiny dataset) has minimal absolute runtime (~0.09s), so parallelization overhead reduces efficiency.

Interpretation: Where Does General Solver Parallelization Help?

- Depth = 2: General solver NOT invoked. Expected speedup: ~1.0×.
- Depth = 3: General solver called ONCE at root. D2 solver handles subtrees. Result: 1.90× speedup.
- Depth = 4: General solver called MULTIPLE times. Problem is too large; both serial and parallel hit 600s limit.
- Depth ≥ 5: General solver dominates. Prohibitively slow without massive parallelization.

Amdahl's Law Analysis

For depth = 3, let f_general denote the fraction of total runtime spent in the general solver. Assuming perfect 8× speedup on the parallelized portion:

```
S_max = 1 / ((1 - f_general) + f_general / 8)
From our observed speedup of 1.90×:
1.90 = 1 / ((1 - f_general) + f_general / 8)
0.526 = 1 - 0.875 * f_general
f_general ≈ 0.54 (54%)
```

This suggests that at depth 3, the general solver accounts for roughly 54% of the total runtime

Load Balancing and Scheduling Strategy

The use of schedule(dynamic) is critical because features have heterogeneous costs due to variable split points, pruning variance, and data distribution. Dynamic scheduling ensures threads remain busy, whereas static scheduling would lead to idle time if one thread received expensive features.

Scalability and Efficiency

Threads	**Speedup**	**Efficiency**
1	1.00×	100%
2	1.34×	67%

4	1.81×	45%
8	1.90×	24%

Efficiency decreases at 8 threads due to Amdahl's Law (sequential D2 solver), synchronization overhead, and limited parallelism (number of features is small compared to threads).

Conclusion

Parallelizing the general solver addresses a key limitation identified in Section 5.2: at depths ≥ 3, the general solver begins to dominate runtime. By targeting the feature enumeration loop, we achieve meaningful speedup at depth 3 (1.90× with 8 threads for bean.txt).

---

6 Expected Contributions

6.1 OpenMP Parallelization of the Depth-2 Specialized Solver

Coarse-grained parallelism for the root feature enumeration loop of the D2 solver has already been implemented and verified. This parallelization achieves about 3.94× speedup on configurations where depth-2 computations dominate runtime (e.g., bean.txt with max-depth 3), as shown in Section 5.2. With a final reduction step to update the global optimum, each thread independently assesses a unique root feature while preserving private upper bounds and local best trees.

6.2 OpenMP Parallelization of the Branch Procedure

We intend to extend OpenMP parallelism to the general CT(D,d) recursion, as described in Sections 2.2 and 3. Similar to the D2 root-feature approach, this entails parallelizing per-feature evaluations within the Branch procedure using a parallel for construct. The performance bottleneck at deeper tree depths (max-depth ≥ 5), where the current sequential CT/Branch recursion dominates total runtime, is the focus of this contribution.

6.3 GPU-Resident Depth-2 Solver

We will create a CUDA implementation of the depth-2 specialized solver, as explained in Section 4. During sorted data traversal, thread blocks will be allocated to features or (root-feature, root-split) pairs, and shared memory will be utilized for class-count accumulators. This GPU-resident version eliminates CPU–GPU transfers for the D2 hotspot and offers a different execution path that can be chosen at runtime according to hardware availability and dataset size.

6.4 Batched Split Scoring Kernel

We will implement a CUDA kernel for batched evaluation of candidate split points in accordance with the design in Section 4. The kernel evaluates all (feature, threshold) pairs in parallel, calculating left/right class counts and misclassification scores after CPU-side pruning (Neighborhood Pruning, Interval Shrinking, Sub-interval Pruning) finds surviving candidates. This takes the place of the Branch procedure's sequential per-threshold evaluation.

6.5 Optimized GPU Data Layout and Prefix Sum Precomputation

For coalesced GPU memory access, we will convert the dataset from array-of-structures to structure-of-arrays layout, as explained in Section 4. Furthermore, we will use GPU scan primitives to precalculate per-feature, per-class prefix sums (PrefixCounts[f][c][i]). By enabling constant-time class count lookups during split scoring, these prefix sums minimize per-candidate computation and necessitate a single dataset transfer at initialization.

6.6 Scalability Benchmarks

We will perform thorough benchmarks across tree depths 2-4 on the given datasets (bean.txt, page.txt, bank.txt), as detailed in Section 2.2 and the timeline in Section 2.3. Our hybrid CPU–GPU design will be empirically validated by these benchmarks, which will measure speedup, parallel efficiency, and the impact of each parallelization strategy.

---

7 References

References
[1] Catalin E. Brita et al., Optimal Classification Trees for Continuous Feature Data Using Dynamic Programming with Branch-and-Bound. AAAI 2025. Available in project description (CS406/531 Course Project).

---

8 Appendix

A. Generative AI Usage Statement

We utilized Large Language Models (ChatGPT-5) to assist with the optimization and implementation strategies of the project. The usage was limited to brainstorming and debugging specific modules. The details are as follows:

1. CUDA Implementation Strategy & Brainstorming To accelerate the decision tree construction, we consulted the AI model for high-level architectural ideas and potential parallelization strategies suitable for our specific data structure.
* Link: https://chatgpt.com/share/6931e52d-c588-8011-8a15-923b3ec5324a