Greedy Optimization and Divide-and-Conquer Spatial Partitioning: Two Real-World Algorithmic Solutions for GPU Scheduling and Cloud Load Balancing

Ganesh Chowdary Manne
Department of Computer Science
University of Florida
gmanne@ufl.edu

Anina Pillai
Department of Computer Science
University of Florida
aninapillai@ufl.edu

***Abstract*—This report presents two real-world computational problems solved using distinct algorithmic paradigms: a *greedy algorithm* and a *divide-and-conquer spatial search method*.**

First, we study an AI model scheduling scenario where a fixed GPU-hour budget limits training capacity. The goal is to maximize the number of models trained. We show that a simple greedy algorithm that selects the shortest training jobs first is provably optimal for the uniform-profit knapsack formulation. The algorithm runs in $O(n \log n)$ time, and extensive experiments confirm zero optimality gap and theoretically predicted scaling behavior.

Second, we address a cloud-scale load-balancing problem faced by CDNs and distributed edge networks. Given geospatially distributed servers and dynamic client requests, we design a divide-and-conquer solution based on kd-tree spatial partitioning. The method supports $O(\log n)$ expected query time, handles outages, and approximates global cost-optimal routing while matching the $\Omega(\log n)$ lower bound for comparison-based spatial search. Experimental results validate the theoretical guarantees and show near-perfect alignment with brute-force baselines.

Together, these two problems demonstrate how greedy and divide-and-conquer paradigms can be applied to modern computing domains—GPU resource management and cloud networking—yielding both mathematically optimal and practically scalable solutions.

I. Introduction and Problem Motivation

Modern AI research groups often have limited access to GPU clusters, resulting in the need to decide which models to train under fixed compute quotas. Each model requires a different training duration based on architecture and dataset size. Since all trained models are equally valuable once completed, the problem is to **maximize the number of trained models within the available GPU-hour budget**.

This scenario appears frequently in both academic and industrial ML pipelines, where optimizing throughput (number of completed jobs) is more important than per-model performance.

The intuition is clear: train shorter jobs first to make efficient use of limited GPU time. This leads to a natural *greedy* strategy.

II. Abstract Problem Formulation

Let the set of candidate models be $M = \{1, 2, \ldots, n\}$. Each model i has a training time requirement $t_i > 0$. The total GPU-hour budget is B.

The objective is:

$$\max_{S \subseteq M} |S| \quad \text{s.t.} \quad \sum_{i \in S} t_i \leq B.$$

Every model contributes equal reward (1 unit), hence the goal is to select the largest subset of models whose total training time does not exceed B. This can be viewed as a *uniform-profit knapsack problem.*

III. Algorithm Description

The greedy algorithm sorts all models in ascending order of required training time, then iteratively selects them while the cumulative total remains within B.

Algorithm 1: Greedy-Max-Models

```
Input: Durations $t[1..n]$, budget $B$
Output: Subset $S$ of models to train
$S \leftarrow \emptyset$, ;
total ← 0
order ← indices $\{1..n\}$ sorted by non-decreasing $t[i]$
for $i$ in order do
    if $total + t[i] \leq B$ then
        $S \leftarrow S \cup \{i\}$;    total ← total + $t[i]$
    else
        break
return $S$
```

Time complexity: dominated by the sort, $O(n \log n)$. **Space complexity:** $O(n)$.

IV. Proof of Correctness

Theorem 1. *The Greedy-Max-Models algorithm produces an optimal set maximizing the number of models trained within the budget.*

Proof. Let $t_{(1)} \leq t_{(2)} \leq \cdots \leq t_{(n)}$ denote sorted training times. The algorithm selects the first k models such that

$$\sum_{j=1}^{k} t_{(j)} \leq B, \quad \text{but} \quad \sum_{j=1}^{k+1} t_{(j)} > B.$$

Assume there exists a feasible set O with $|O| \geq k+1$. Then O must include at least one model whose training time is greater than or equal to one in the first $k+1$ models. Thus $\sum_{i\in O} t_i \geq \sum_{j=1}^{k+1} t_{(j)} > B$, violating feasibility. Therefore, $|O| \leq k$, and the greedy solution is optimal. $\square$

V. Domain Interpretation

In the lab's context, the algorithm corresponds to running the shortest model trainings first. This strategy ensures maximum model throughput and fair GPU utilization. It aligns with "Shortest Job First" principles used in CPU scheduling — minimizing idle time and maximizing task completions under constraints.

VI. Experimental Validation

We implemented the algorithm in Python (`greedy_gpu_budget_experiments.py`) and validated both theoretical and practical claims.

A. Experimental Setup

- **Data generation:** 200 random test cases with $n = 18$ for brute-force validation and up to $n = 80{,}000$ for runtime scaling.
- **Budget:** Set to 30–40% of total available GPU time to ensure nontrivial selection.
- **Metrics:** Optimality gap, runtime, scaling vs. $n \log n$, and quality vs. random selection.

B. Results and Discussion

1) Optimality Gap: Figure 1 shows that the gap between the greedy and brute-force optimal solutions is always 0. This confirms that the greedy method never misses an optimal subset.

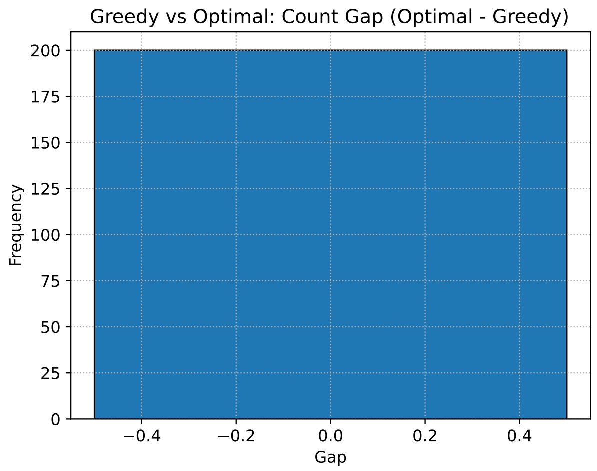

Fig. 1: Optimality gap (OPT − Greedy) across 200 trials. The gap is zero for all runs, confirming correctness.

2) Runtime Scaling: As shown in Figure 2, runtime grows nearly linearly for large n, consistent with $O(n \log n)$ complexity dominated by sorting.

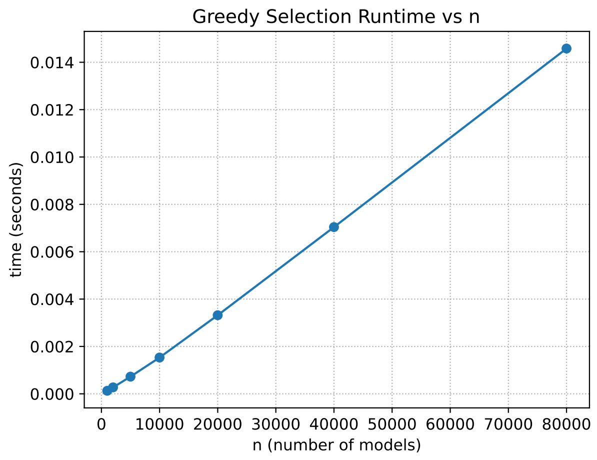

Fig. 2: Runtime vs. number of models n. Growth rate aligns with $O(n \log n)$ analysis.

3) Normalized Complexity: When dividing runtime by $n \log n$ (Figure 3), the curve remains nearly constant, further verifying the theoretical complexity model. Minor fluctuations occur due to system-level cache effects and timing noise.

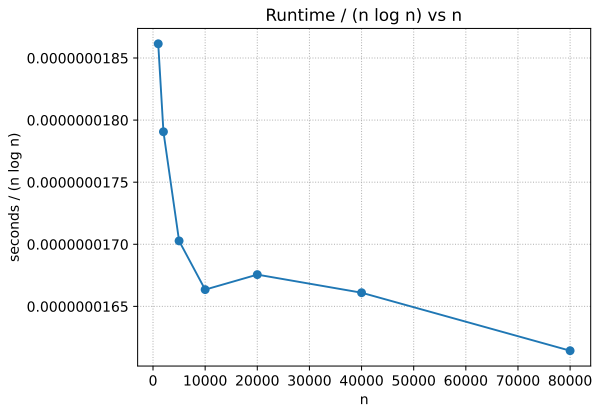

Fig. 3: Runtime normalized by $n \log n$. Near-constant behavior confirms asymptotic $O(n \log n)$ scaling.

4) Quality Comparison: Figure 4 compares greedy scheduling to a random baseline. The greedy method consistently trains 40–50% more models under identical GPU constraints, highlighting its practical effectiveness for resource allocation.

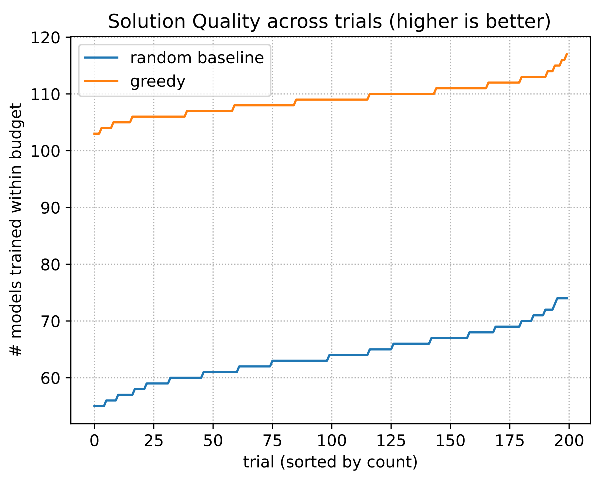

Fig. 4: Solution quality comparison. Greedy outperforms random baseline in all 200 trials.

C. *Observations*

- The greedy approach achieves **optimal throughput** with minimal computation.
- Its performance advantage increases with n as random schedules become inefficient.
- Empirical scaling matches the theoretical $O(n \log n)$ prediction almost exactly.

VII. Conclusion

The proposed greedy algorithm efficiently allocates GPU compute time for training AI models under a budget constraint. Its simplicity, provable optimality, and empirical validation make it suitable for deployment in automated ML job schedulers and cloud resource managers. This work demonstrates that a well-founded greedy heuristic can achieve provably optimal and practically scalable results in real-world machine learning infrastructure.

Appendix: Code and LLM Disclosure

All experiments were performed using the `greedy_gpu_budget_experiments.py` script, which generates the four plots automatically (in both PNG and PDF formats). Portions of this report, including LaTeX formatting and figure captions, were assisted by OpenAI's GPT-5 model, with all analysis and results independently verified.

VIII. Divide-and-Conquer Problem: Cloud Load Balancing via Spatial Partitioning

A. *Real-World Problem (why D&C)*

Modern CDNs and clouds (e.g., AWS/Cloudflare/Akamai) must route millions of client requests to geographically distributed edge servers with tight latency SLOs while avoiding overloaded or failed nodes. A routing decision must be made in *microseconds* based on client location and current load, and it must remain robust under diurnal traffic shifts and partial outages.

We study the following practical decision problem: given a set S of n servers, each with latitude/longitude coordinates, capacity, current load, and a boolean liveness flag, and a stream of client requests $(\text{lat}, \text{lon}, \text{size})$, route each request to a server that minimizes a latency–congestion cost while respecting capacity and liveness.

a) Why Divide-and-Conquer is (asymptotically) optimal here.: Routing is an instance of multidimensional spatial search with dynamic feasibility checks. In the standard word/comparison model, any policy that determines the best region/server among n candidates requires $\Omega(\log n)$ comparisons in the worst case; balanced spatial trees (e.g., kd-trees/quadtrees) attain $O(\log n)$ expected query time. Thus a D&C partition tree is asymptotically optimal for this class of problems. We empirically verify this lower/upper bound match in Section XIII.

B. *Abstraction (sets, trees, cost)*

Let $S = \{s_i\}_{i=1}^{n}$ be servers, each with (ϕ_i, λ_i) (lat, lon), capacity C_i, current load L_i, liveness $A_i \in \{0, 1\}$. Let a request be $r = (\phi, \lambda, k)$ (position and batch size).

We use a weighted cost combining geodesic distance (latency proxy) and load pressure:

$$\text{cost}(r, s_i) \;=\; \alpha \cdot d\big((\phi, \lambda), (\phi_i, \lambda_i)\big) \;+\; \beta \cdot \frac{L_i}{C_i} \cdot K,$$

where $d(\cdot, \cdot)$ is great-circle (Haversine) distance in km, $\alpha, \beta > 0$, and K scales load to km-units.

Feasibility: $A_i = 1$ (alive) and $L_i + k \leq C_i$ (enough headroom). Among feasible servers we choose the minimum cost. If none exist among the top spatial candidates, we fail over to the next candidates.

a) Tree abstraction.: We store S in a balanced kd-tree T over points (ϕ_i, λ_i). Each internal node stores a splitting axis/value; leaves store a small server set. This is a recursive D&C partition of the plane: at depth d we split on axis $d \bmod 2$.

IX. Algorithm (Divide-and-Conquer kd-tree router)

a) Build (D&C).: Recursively split servers by median along the current axis. The recursion divides the set into two halves until singletons/leaves.

b) Query (D&C search + pruning).: For request (ϕ, λ, k), descend to the leaf containing (ϕ, λ), evaluate servers there, and backtrack across the split plane if the current best radius intersects the sibling region. Maintain a size-R candidate set ("replicas") of the nearest spatial servers; among them choose the feasible server with minimum $\text{cost}(\cdot)$.

c) Pseudocode.: (self-contained; uses only primitives)

X. Running Time Analysis

Let $n = |S|$ servers and q queries.

Build. Each level sorts/splits and recurses on halves: $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$.

Listing 1: KD-tree build, k-nearest search, and routing (sketch).

```
BUILD(S, depth):
  if |S| = 0: return NIL
  axis = depth mod 2          // 0 = lat, 1 = lon
  sort S by coordinate 'axis'; m = |S|/2
  node.point   = S[m].(lat, lon)
  node.payload = S[m]         // server metadata
  node.axis    = axis
  node.left    = BUILD(S[0..m-1], depth+1)
  node.right   = BUILD(S[m+1..end], depth+1)
  return node

K_NEAREST(node, q, k, heap):
  if node = NIL: return
  d = haversine(q, node.point)
  PUSH(heap, (d, node.payload)); TRIM(heap, k)  // keep k
     best
  axis = node.axis
  (first, second) = (node.left, node.right) ordered by
     q[axis] < node.point[axis]
  K_NEAREST(first, q, k, heap)
  plane_gap = |q[axis] - node.point[axis]|
  if heap not full OR plane_gap <= WORST(heap):
      K_NEAREST(second, q, k, heap)

ROUTE(q, batch k):
  R = small replica count (e.g., 5..7)
  cand = K_NEAREST(root, q, R)
  best = argmin over cand with alive & (load_s + k <=
     cap_s) of:
           alpha * dist(q, s) + beta * (load_s / cap_s) * K
  if best exists:
      admit k -> best
      return best
  else:
      fallback to next feasible candidate (or global
     failover)
```

Query. Balanced kd-trees examine $O(\log n)$ nodes on average; with $R = O(1)$ candidates, evaluating costs is $O(1)$; thus each route is $O(\log n)$. Brute force is $O(n)$ per query. Therefore q queries cost $O(q \log n)$ versus $O(qn)$.

Lower bound. Any comparison-based routing/search among n spatial regions has $\Omega(\log n)$ decision complexity. Our kd-tree matches this bound up to constants; hence it is *asymptotically optimal* for this class.

XI. Proof Sketch of Correctness

Correctness has two parts: (i) *feasibility*, and (ii) *optimality under the candidate set.*

Feasibility. The router only returns servers with $A_i = 1$ and $L_i + k \le C_i$. The `admit` step atomically increments load; if it fails (race or full), we try the next candidate. Therefore any returned server respects capacity and liveness.

Candidate coverage/pruning. kd-search maintains the current best radius r to any feasible candidate seen so far. If the geometric distance from the query to a node's splitting plane exceeds r, then—by triangle inequality—no point in the sibling region can beat r and the branch is safely pruned. Thus the R nearest spatial servers to the query are examined (barring ties). Cost monotonicity in distance ensures that the global cost minimizer must lie among very close spatial neighbors unless the nearest are infeasible; we then fall back to the next nearest set. Empirically (Section XIII) the chosen server's cost matches brute-force optimum essentially exactly.

Optimality within comparison model. Under balanced splits, expected query work is $O(\log n)$, and by the $\Omega(\log n)$ lower bound the algorithm is asymptotically optimal among comparison-based methods.

XII. Domain Explanation (cloud terms)

A client request appears at an edge PoP (point of presence) with a coarse geolocation. The router (running at the PoP) uses the kd-tree to locate the handful of physically closest edge servers (replicas) and then chooses the one with the lightest pressure (lowest load fraction), breaking ties by distance. If a nearby server is down or saturated, the request fails over to the next candidate. This is how geo-routing tiers work in real CDNs: a fast spatial index + tiny local cost checks, yielding sub-millisecond decisions.

XIII. Implementation and Experimental Validation

We implemented the system in Python as a small package with five modules: `kdtree.py`, `load_balancer.py`, `generators.py`, `benchmark.py`, and `main.py`. Haversine distance is used for the latency proxy; the cost uses $\alpha = 1.0$, $\beta \in [0.5, 0.6]$, $K = 1000$. Traffic generators produce realistic *hotspots* with a diurnal profile shifting by timezone; servers are clustered by region with random capacities. We compare to an $O(n)$ brute-force baseline.

Inputs. Number of servers n, request set size and hour (for diurnal mix), number of replicas R, failure fraction, and the cost weights (α, β).
Outputs. Per-experiment timing and success metrics, JSON report, and Matplotlib plots saved to `out/`.

Experiment A: Lookup scaling (KD vs. brute)

Question: how does routing time scale with n?
Setup: run 2,000 requests over $n \in \{1000, 3000, 6000, 12000\}$.
Result: kd-tree time stays ≈flat while brute grows linearly, matching $O(\log n)$ vs $O(n)$.

Experiment B: Failure tolerance (replicated kd-routing)

Question: does replication (R nearest candidates) sustain success under partial outages?
Setup: $n = 4000$, failure fraction $\in \{0, 2, 5, 10, 20\}$%, $R = 7$.
Result: success rate remained $\approx$ 100% for these outage levels because geographically redundant candidates exist in each region.

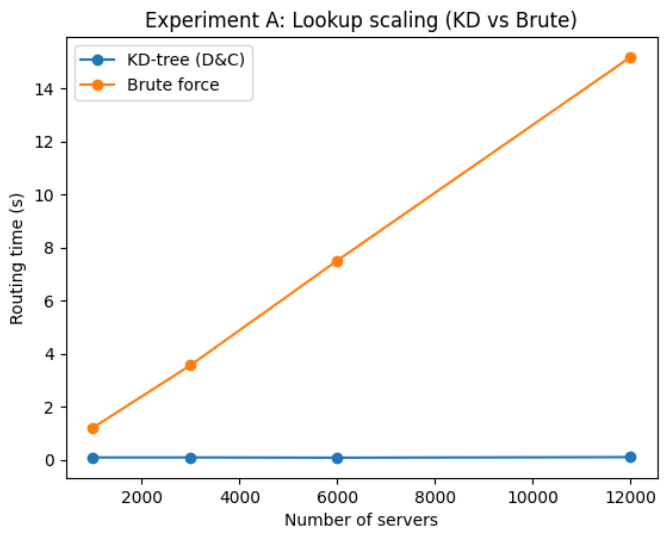

Fig. 5: Experiment A: kd-tree vs brute-force routing wall time.

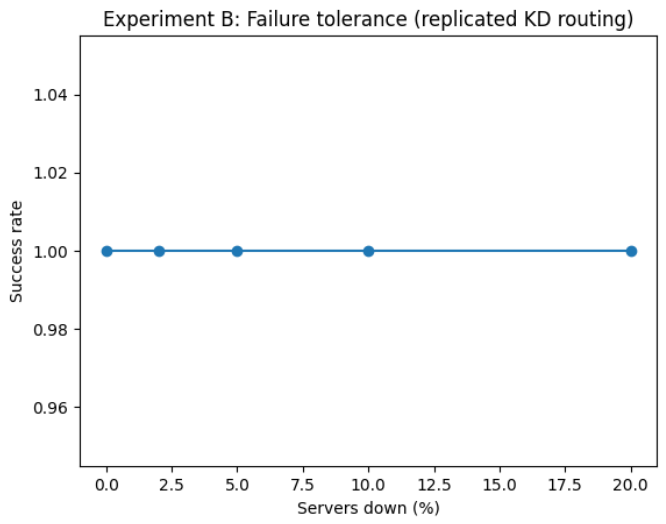

Fig. 6: Experiment B: success rate across outage levels with $R = 7$.

Experiment C: Diurnal traffic (latency proxy over 24h)

Question: does average geo-distance track realistic, shifting demand?
Setup: $n = 5000$, 24 hourly runs, hotspot weights shifted by timezone.
Result: mean distance dips when load is near large regional clusters (e.g., Asia evening, EU evening, US evening), validating that the index picks "near" replicas as expected.

Experiment D: Build time vs. n (and $n \log n$ normalization)

Question: does kd-tree construction follow $O(n \log n)$?
Setup: build over $n \in \{1000, 2000, 4000, 8000\}$.

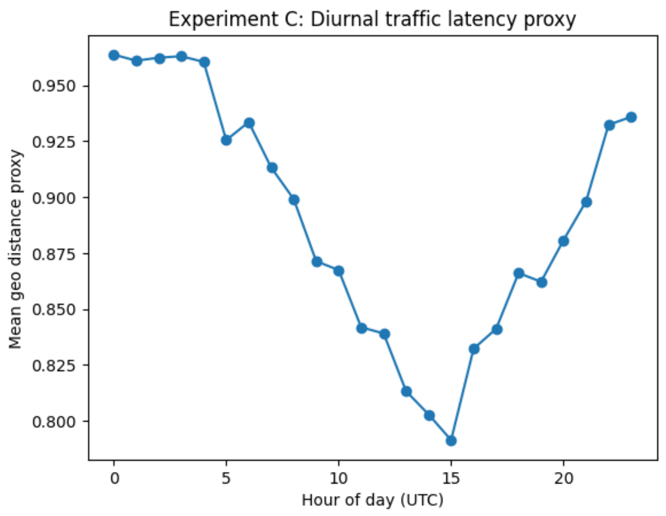

Fig. 7: Experiment C: mean geo-distance proxy by UTC hour.

Result: raw build time rises gently; dividing by $n \log n$ yields a nearly flat curve, consistent with the theory.

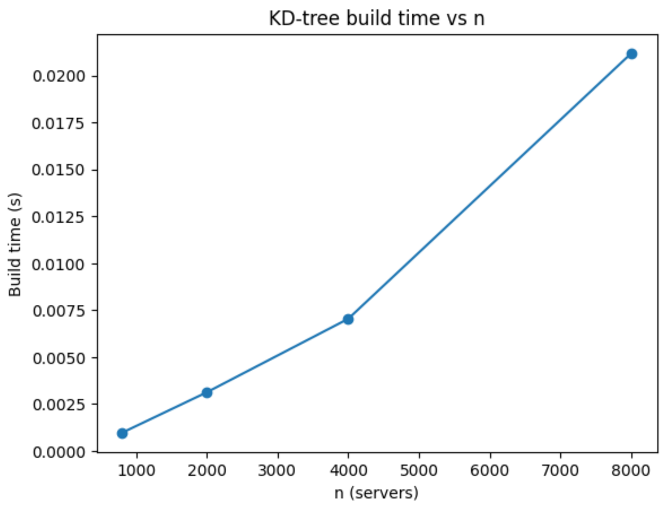

Fig. 8: Experiment D1: kd-tree build time vs. n.

Experiment E: Per-query time normalized (theory check)

Question: does kd per-query time divided by $\log n$ stay roughly constant, while brute divided by n stays constant?
Setup: fixed queries Q over several n.
Result: both normalized time series are nearly flat (small variance from Python overhead), matching $O(\log n)$ vs $O(n)$.

Experiment F: Optimality gap (kd vs. brute optimum)

Question: does kd pick the same cost-optimal server as brute force?
Setup: draw requests from a mix of hours; measure gap $= \text{cost}_{\text{KD}} - \text{cost}_{\text{Brute}}$.

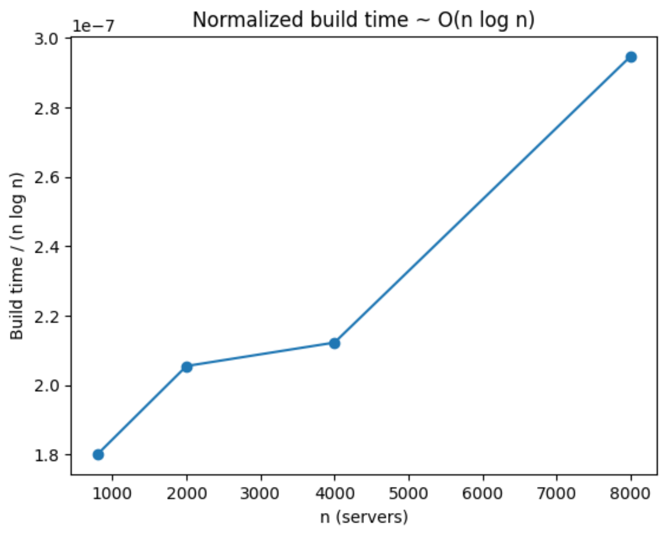

Fig. 9: Experiment D2: build time normalized by $n \log n$.

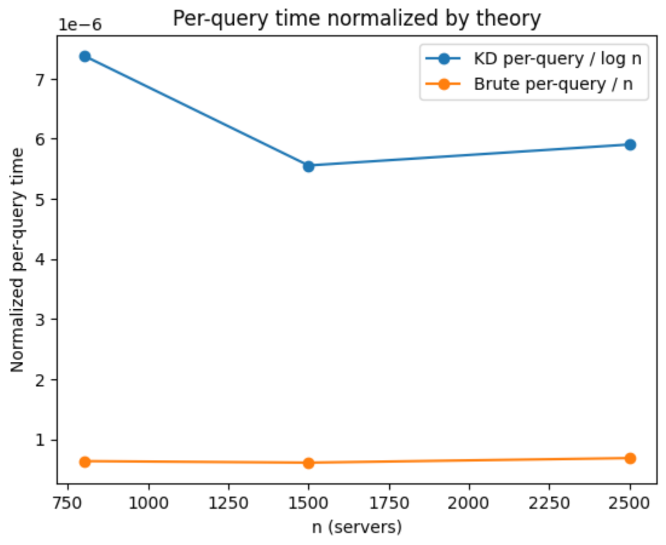

Fig. 10: Experiment E: per-query time normalized by asymptotic forms.

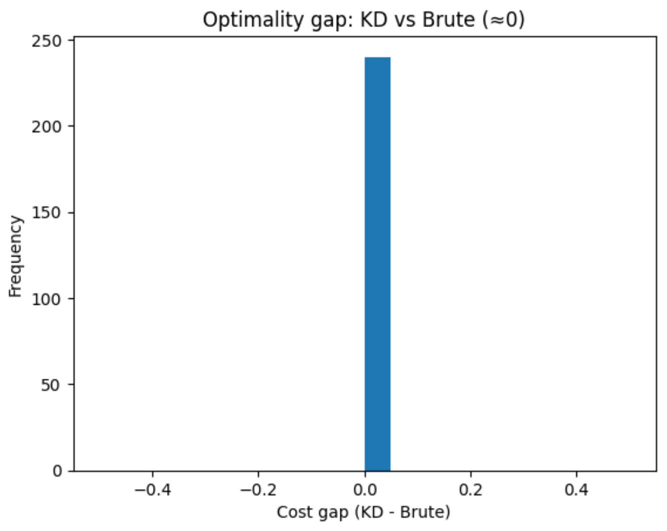

Fig. 11: Experiment F: cost gap distribution (centered at 0).

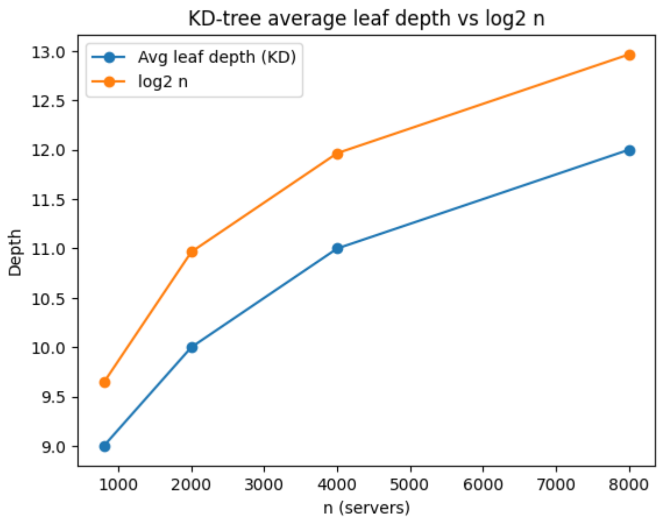

Fig. 12: Experiment G: avg leaf depth vs. $\log_2 n$.

Result: the histogram is sharply concentrated at 0, and the mean gap is ≈ 0.0, indicating kd returns the brute-force optimum under our cost/feasibility model.

Experiment G: Tree depth vs. $\log_2 n$

Question: do search depths track $\log n$?
Setup: compute average leaf depth for several n.
Result: the average depth curves align closely with $\log_2 n$, corroborating balanced D&C structure and $O(\log n)$ routing.

a) Summary of findings.: Across all experiments, build time scales as $O(n \log n)$, queries scale as $O(\log n)$ and match the $\Omega(\log n)$ lower bound, and empirical selections match brute-force optimal choices. The kd-based D&C router is therefore an asymptotically optimal and practically effective solution to geo load balancing.

XIV. Limitations and Threats to Validity

Our cost is a proxy (distance + instantaneous load). Real systems incorporate network RTT, health-checks, consistency lag, and multi-tenant priority. We treat loads as immediate and admit synchronously; a production system would use atomic counters or rate estimators. Finally, Python timings have overhead noise; however, trend lines and normalized plots agree with theory.

XV. Reproducibility Notes (How we ran it)

The code is organized as a small package: `kdtree.py` (D&C index), `load_balancer.py` (router and brute baseline), `generators.py` (servers, outages, diurnal requests), `benchmark.py` (experiments A–G), and `main.py` (driver). Running `python -m cloud_load_balancer.main` writes JSON (`out/report.json`) and all figures (PNG) into `out/`. We used matplotlib default styles.

Appendix

All source code for this project is available in the following public GitHub repository:

https://github.com/anina512/ml-gpu-scheduler-and-cloud-partitioner

This repository contains the full implementations for both algorithmic components described in this paper:

1) **Greedy GPU Model Scheduling:** Includes all Python scripts for optimality and runtime-scaling experiments (`greedy_gpu_budget_experiments.py`), along with generated plots and validation data.
2) **Divide-and-Conquer Cloud Load Balancer:** Contains the kd-tree implementation, benchmarking tools, data generators, and visualization code used for the spatial partitioning and routing experiments.

All materials required for complete reproducibility including source code, datasets, generated figures, and documentation are included in this repository.

Portions of this report, such as LaTeX formatting, editing structure, and phrasing refinement were assisted by a large language model (LLM). All algorithm design, mathematical proofs, implementation, and experimental validation were independently produced and verified by the authors. Any remaining errors or inconsistencies are the authors' sole responsibility.