Parallel and GPU-Accelerated Branch and Bound Solving Algorithms for Sparse Logistic Regression

James Jones, Advisor: Dr. Weijun Xie

December 9, 2025

Contents

1 Executive Summary

Sparse logistic regression plays a central role in scientific and industrial applications where both predictive accuracy and interpretability are required. However, enforcing exact sparsity through an ℓ_0 constraint leads to a challenging combinatorial optimization problem. Branch and bound frameworks provide a principled way to obtain globally optimal sparse models, but their performance is fundamentally limited by repeated subproblem solves and by the size of the search tree.

This project develops new algorithms and computational techniques to improve the scalability of branch and bound for ℓ_0-constrained logistic regression. We introduce two major advancements: (1) a parallel branch and bound architecture that distributes branching decisions across multiple worker processes, and (2) a GPU-accelerated stochastic gradient descent solver tailored for the subproblems encountered during search. Together, these enhancements substantially reduce runtime while preserving solution quality.

Our experiments demonstrate that parallel branching achieves near-ideal speedup up to four workers, and that GPU-accelerated stochastic gradient descent provides more than an order-of-magnitude improvement over CPU-based subsolvers. Furthermore, despite the nonconvex landscape, the GPU-based subsolver maintains high accuracy in lower-bound estimation, enabling effective pruning of the search tree.

Overall, this work shows that principled algorithm design—combined with modern parallel and GPU computing—can dramatically accelerate exact sparse logistic regression, making global feature selection more feasible at scale.

2 Research Motivation

Logistic regression is one of the most widely used statistical models for binary classification across science, engineering, medicine, and finance. Its appeal stems from its interpretability and its ability to quantify how individual features influence an outcome. In many modern applications, however, datasets contain far more potential predictors than are truly relevant. Identifying a small, meaningful subset of features—sparse modeling—is therefore essential both for interpretability and for avoiding overfitting.

Convex optimization techniques, such as ℓ_1 or ℓ_2 regularization, are commonly used to induce sparsity, but provide less clear insights than non-convex measures, especially when dealing with data that is not uniform in scale.

Non-convex methods can provide more precise and more direct insights, but are computationally difficult to solve, especially at scale.

Branch and bound is a promising framework for this task because it can guarantee globally optimal sparse solutions. However, its practical performance is often limited by the size of the search tree, which grows exponentially in the worst case, and the cost of repeatedly solving logistic regression subproblems at each node of the tree.

This project continues the development of a branch and bound solver tailored specifically for sparse logistic regression. Our primary motivation is to close the gap between the theoretical optimality of ℓ_0-based methods and their practical scalability. To do this, we design new algorithms that make far better use of modern hardware. In particular, we introduce parallel mechanisms for the branching procedure and a GPU-accelerated solver for evaluating subproblems.

By leveraging parallel computing and GPU-based optimization, we aim to bring exact sparse logistic regression closer to real-world feasibility on high-dimensional datasets—where interpretability, accuracy, and computational efficiency are all critical.

3 Research Objective

The goal of this project is to develop and implement scalable solvers for ℓ_0-constrained logistic regression. Building upon our existing branch and bound framework, our goal is to significantly reduce runtime by designing solvers that leverage hardware capabilities more effectively.

More specifically, we aim to:

- Develop a parallelized branch and bound architecture that distributes branching operations across multiple worker processes with minimal synchronization overhead.
- Develop a GPU-accelerated subproblem solver to dramatically reduce per-node solve time.
- Evaluate performance on real datasets to determine how solver improvements translate into faster global optimization.

Through these efforts, we aim to expand the practical applicability of exact sparse logistic regression to larger datasets and higher-dimensional feature spaces.

4 Research Methodology

4.1 Problem Formulation

Given feature and target datasets $X \in \mathbb{R}^{m\times n}, y \in \{0,1\}^m$ and an ℓ_0 constraint $0 < k \leq n$, we produce coefficient vector $\theta \in \mathbb{R}^n$ according to the following

$$\min_{\theta} \sum_{i=1}^{n} \left[-y_i \ln\left(\frac{1}{1+e^{-X_i^\top \theta}}\right) - (1-y_i)\ln\left(1-\frac{1}{1+e^{-X_i^\top \theta}}\right)\right] \quad \text{s.t.} \quad \|\theta\|_0 \leq k$$

4.2 Baseline Branch and Bound Solver

4.2.1 Subproblem Representation

Each node in the branch and bound tree corresponds to a subproblem $\mathcal{P}_{\mathcal{F}^+,\mathcal{F}^-}$ defined by:

- $\mathcal{F}^+$: features forced to be included
- $\mathcal{F}^-$: features forced to be excluded

The remaining features are free.
A branch picks a free feature i and creates two children:

- Left child: $\mathcal{F}^+ \cup \{i\}$
- Right child: $\mathcal{F}^- \cup \{i\}$

4.2.2 Relaxation and Bounds

We use a relaxation function $\Phi(\cdot)$ that evaluates the subproblem while respecting fixed inclusions/exclusions. A key monotonicity property holds:

$$\Phi(\mathcal{P}_{\mathcal{F}^+,\mathcal{F}^-}) \leq \Phi(\mathcal{P}_{\mathcal{F}^+\cup\{i\},\mathcal{F}^-}) \quad \text{and} \quad \Phi(\mathcal{P}_{\mathcal{F}^+,\mathcal{F}^-}) \leq \Phi(\mathcal{P}_{\mathcal{F}^+,\mathcal{F}^-\cup\{i\}})$$

i.e., fixing variables never decreases the lower bound, which guarantees correctness of branch and bound.

4.2.3 Algorithm Summary

Algorithm 1 Branch and Bound

Initialize $U = \{\mathcal{P}_{init}\}$, the set of unexplored subproblems
Initialize $F = \emptyset$, the set of feasible subproblems
Compute the initial lower bound $LB = \Phi(\mathcal{P}_{init})$ and an initial $UB = f^{UB}$
while $UB - LB > \varepsilon$ **do**
  Pick the subproblem in U with the smallest relaxation value.
  Branch on a selected feature i.
  Add feasible leaves to F, add active nodes back to U, prune nodes with relaxation values exceeding UB
  Update $UB = \min_{\{\mathcal{P}\in F\}} \Phi(\mathcal{P})$ and $LB = \min_{\{\mathcal{P}\in U\}} \Phi(\mathcal{P})$
end while
Return the best feasible solution $\arg\min_{\{\mathcal{P}\in F\}} \Phi(\mathcal{P})$ within ε.

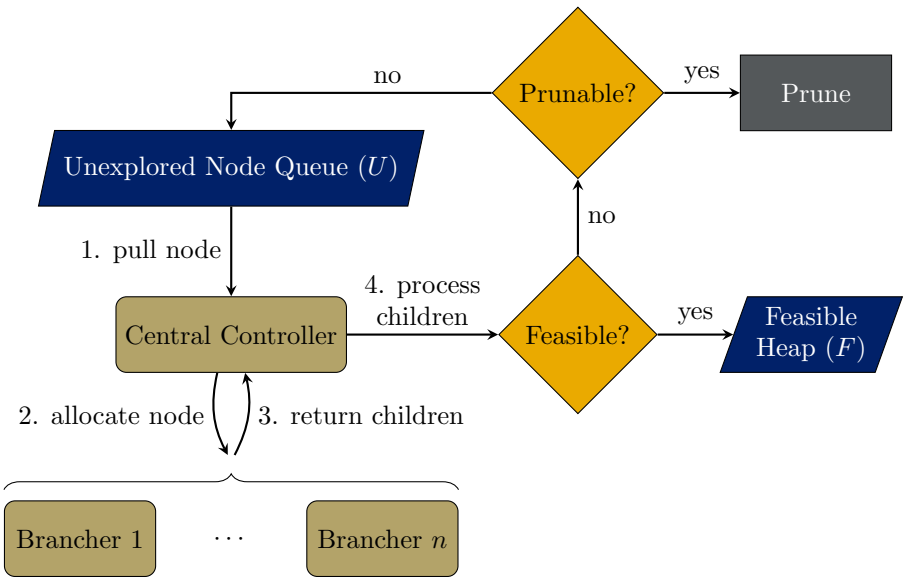
4.3 Parallel Branching

We use a controller-worker model to parallelize the branching step of the branch and bound algorithm. A single controller thread maintains the global unexplored node queue U as well as the the global upper and lower bounds and is responsible for distributing work to a pool of *brancher* workers. Each brancher receives a subproblem, selects the next branching variable, generates its child subproblems, and computes the child subproblem bounds before returning nodes back to the central controller for handling.

This design ensures that branching, which is often one of the most computationally expensive components of the search process, can be parallelized with minimal synchronization overhead. The controller performs only lightweight operations—queue management and node dispatch—while all computationally heavy branching and bounding work is offloaded to the workers.

Our implementation uses Python's Multiprocessing Module, with Process objects as branchers and Pipes for communication.

Figure 1: Parallel Branching Process Flow



4.4 GPU Acceleration

To accelerate the repeated subproblem solves required during branch and bound search, we implemented a stochastic gradient descent (SGD) solver optimized for execution on a GPU. Each subproblem corresponds to fitting a logistic regression model with a restricted feature set determined by the branching decisions made so far. Because these subproblems must be solved thousands of times across the search tree, solver efficiency directly impacts overall performance.

SGD is particularly attractive in this context because its low per-iteration cost and simple update rules map efficiently to GPUs. Our GPU-based solver performs minibatch stochastic gradient updates using CuPy [3] calls to Nvidia's CUDA platform. For each subproblem, the CPU controller assembles the feature-mask for the dataset and transfers it to GPU memory, after which all gradient evaluations, parameter updates, and loss computations are executed on the device. This design minimizes CPU–GPU communication and allows the solver to exploit the high-throughput parallelism of modern GPUs.

4.5 Testing Methodology

All tests were conducted using Georgia Tech's AI Makerspace with select datasets from the UCI Machine Learning Repository [5].

5 Results

5.1 Parallel Branching

Figure 2: Runtimes vs Brancher Count

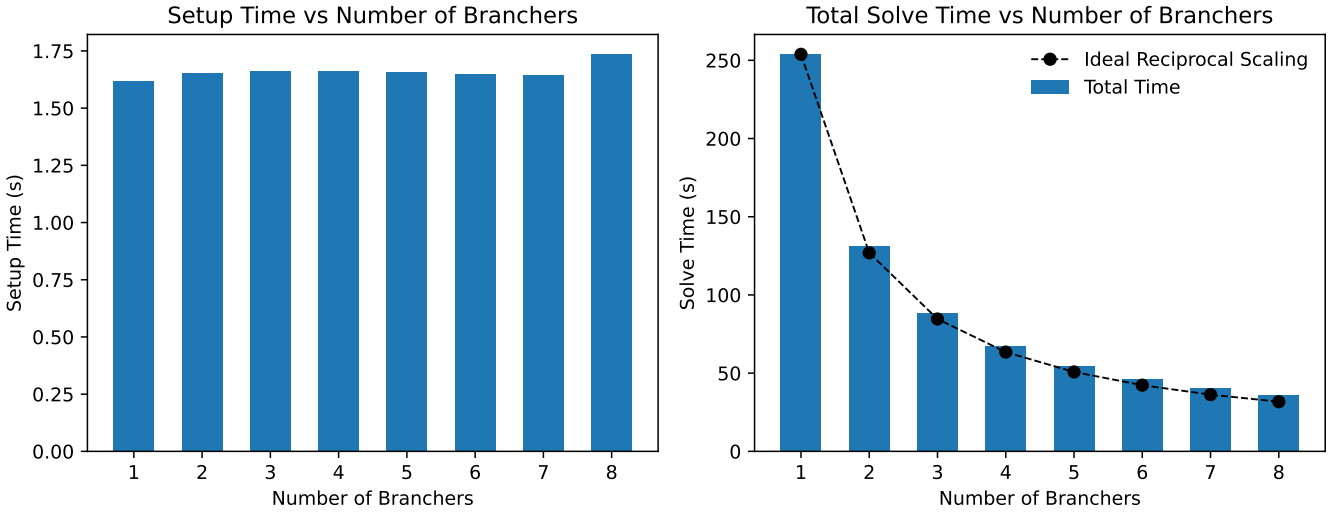


Figure 2 summarizes the performance impact of increasing the number of brancher workers. The left panel shows the mean setup time as the number of branchers increases, while the right panel shows the corresponding mean total solve time for choosing 5 variables from the Ionosphere Dataset [4], along with an idealized reciprocal speedup curve for reference. Setup time increases modestly with additional branchers, reflecting the overhead associated with initializing workers. In contrast, solve time decreases substantially. The observed speedup closely follows the ideal reciprocal trend, indicating with adequate hardware, further time reductions would be attainable.

Figure 3: Brancher Utilization by Brancher Count

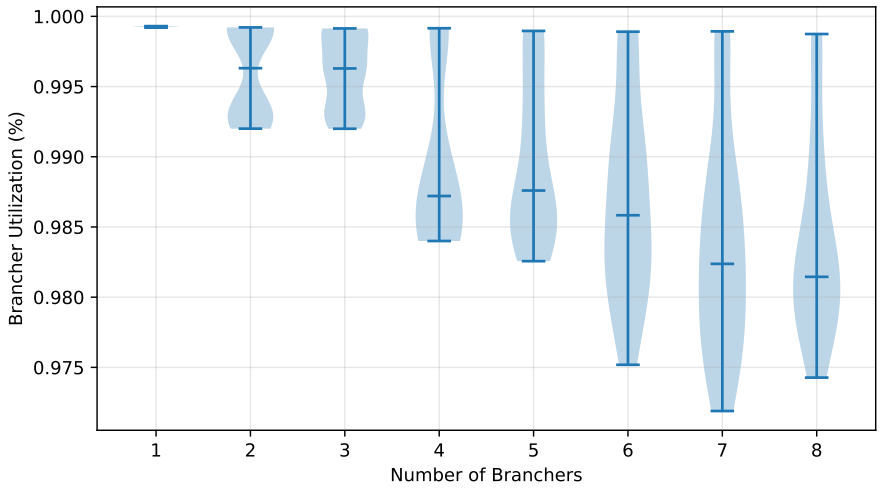
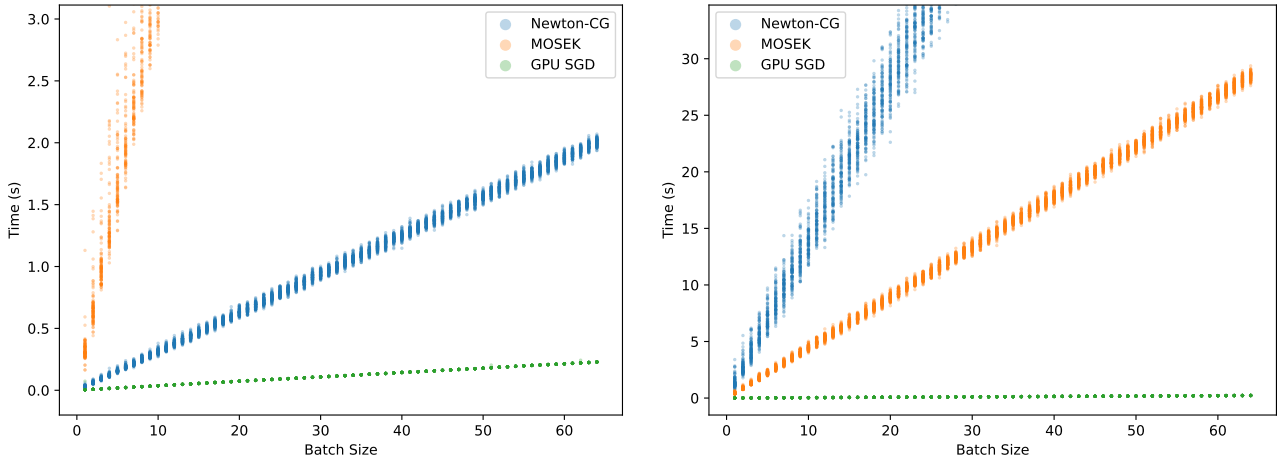


Figure 3 shows violin plots of worker utilization as a function of the number of branchers. Across all configurations, utilization remains extremely high: typically above 97% and frequently exceeding 99%. This indicates that branchers almost always have work available and rarely idle waiting on the controller. High utilization across all tested brancher counts suggests that the system is not bottlenecked by load imbalance or by the pace at which the controller supplies work.

5.2 GPU Acceleration

Figure 4: Runtime per Batch Size Across Subsolvers

Dataset	Myocardial Infarction Complications			Madelon		
Subsolver	Slope (s/problem)	Intercept (s)	R^2	Slope (s/problem)	Intercept (s)	R^2
Newton-CG	0.0313	0.0311	0.9994	1.4081	1.3880	0.9981
MOSEK	0.3828	0.4183	0.9852	0.4452	0.4490	0.9996
GPU SGD	0.0036	0.0049	0.9997	0.0036	0.0049	0.9997



(a) Myocardial Infarction Complications Dataset

(b) Madelon Dataset

Figure 4 compares the runtime of the GPU-accelerated SGD subproblem solver with CPU based approaches. These batches are warm started random selections from the midsized Myocardial Infarction Complications dataset

[1] and the large Madelon dataset [2]. We see that not only does SGD achieve more than an order-of-magnitude speedup over CPU-based methods, but it also scales far more efficiently.

Figure 5 shows results for a full solve with a 1000 second cutoff. We see improvements in iteration counts, solve time, and bound convergence that are maintained as the problem scales.

Overall, the GPU-accelerated SGD solver provides dramatic reductions in subproblem solve time. When amortized across the full search, these speedups substantially decrease total branch and bound runtime while preserving solution quality. Future enhancements such as selective restarts or multi-start refinement may further mitigate the small number of inaccurate prunes without sacrificing the solver's performance advantage.

Figure 5: Runtime per Batch Size Across Solvers

Dataset	Instances (m)	Features (n)	ℓ_0 Constraint (k)	Solver	Iterations	Bound Gap Percent	Time (seconds)
Myocardial Infarction Complications	1700	111	4	CPU B&B	1389	11.52%	1000
				MOSEK	2161	7.89%	1000
				GPU SGD B&B	1954	0.0%	18.04
			6	CPU B&B	1489	10.13%	1000
				MOSEK	1939	8.36%	1000
				GPU SGD B&B	33714	0%	303.72
			8	CPU B&B	1660	8.70%	1000
				MOSEK	1916	7.35%	1000
				GPU SGD B&B	66143	2.04%	1000
			10	CPU B&B	1630	7.82%	1000
				MOSEK	1906	6.84%	1000
				GPU SGD B&B	66549	3.13%	1000
Madelon	2000	500	4	CPU B&B	1926	24.54%	1000
				MOSEK	1261	22.28%	1000
				GPU SGD B&B	61071	16.31%	1000
			6	CPU B&B	1912	24.86%	1000
				MOSEK	1283	22.26%	1000
				GPU SGD B&B	43011	20.39%	1000
			8	CPU B&B	1930	24.96%	1000
				MOSEK	1264	21.40%	1000
				GPU SGD B&B	49106	20.41%	1000
			10	CPU B&B	1959	24.98%	1000
				MOSEK	1295	21.83%	1000
				GPU SGD B&B	58478	20.29%	1000

6 Conclusions and Future Work

This work demonstrates that significant improvements to convergence rates can be made in branch and bound methods for sparse logistic regression through parallelism and GPU acceleration. Our parallel branching architecture achieves high utilization and near-ideal scaling for moderate numbers of workers, while our GPU-accelerated stochastic gradient solver reduces subproblem runtimes by more than an order of magnitude. Together, these improvements help quicken convergence with minimal loss in pruning accuracy.

The results highlight the potential of combining principled optimization frameworks with modern hardware to solve large-scale nonconvex problems. However, several opportunities remain for further improvement. Future work may include:

- Adaptive strategies to improve the robustness of SGD-based lower bounds.
- Dual-informed bounding techniques to provide tighter relaxations during search.
- Distributed branch-and-bound, extending parallelism across multiple machines.
- Enhanced branching rules to further reduce search tree size.

- Integration into a production-quality solver to support applied scientific and industrial workloads.

Overall, this project demonstrates that exact sparse logistic regression—long viewed as computationally prohibitive—can be made significantly more scalable through thoughtful algorithm–hardware co-design.

References

[1] S. Golovenkin, V. Shulman, D. Rossiev, P. Shesternya, S. Nikulina, Y. Orlova, and V. Voino-Yasenetsky. Myocardial infarction complications [dataset]. UCI Machine Learning Repository, 2020.

[2] Isabelle Guyon. Madelon. UCI Machine Learning Repository, 2004. DOI: https://doi.org/10.24432/C5602H.

[3] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[4] Vincent Sigillito, Sharon Wing, Laurance Hutton, and Kile Baker. Ionosphere [dataset]. UCI Machine Learning Repository, 1989.

[5] University of California, Irvine. Uci machine learning repository. `https://archive.ics.uci.edu/`, 1987. Accessed: 2025-08-13.