A GPU-accelerated Nonlinear Branch-and-Bound Framework for Sparse Linear Models

Xiang Meng* Ryan Lucas* Rahul Mazumder
*Equal contribution MIT Operations Research Center

Link to paper

Motivation: exact sparse regression is attractive but hard

We consider the $\ell_0\ell_2$-regularized least-squares estimator

$$\min_{\beta\in\mathbb{R}^p} \frac{1}{2}\|y - X\beta\|_2^2 + \lambda_0\|\beta\|_0 + \lambda_2\|\beta\|_2^2.$$

ℓ_0 estimators can achieve near-optimal **support recovery** and **prediction error bounds**, but the optimization problem is NP-hard.
Specialized branch-and-bound solvers can certify optimality on instances orders of magnitude larger than off-the-shelf MIP solvers, yet certification still requires solving many *node relaxations*.
Modern **GPUs** offer massive parallel throughput, but BnB tree traversal is irregular and ill-suited to GPUs.

Perspective MIP relaxation used at each BnB node

Introducing indicators $z_i \in \{0,1\}$ for $\mathbf{1}\{\beta_i \neq 0\}$ and auxiliary variables $s_i \geq \beta_i^2/z_i$:

$$\min_{\beta,z,s} \frac{1}{2}\|y - X\beta\|_2^2 + \lambda_0\sum_i z_i + \lambda_2\sum_i s_i$$
$$\text{s.t. } -Mz_i \leq \beta_i \leq Mz_i, \quad \beta_i^2 \leq s_i z_i, \quad z_i \in \{0,1\}.$$

At a node, fix $z_i = 0$ on F_0 and $z_i = 1$ on F_1, relax the remaining $z_i \in [0,1]$, and eliminate (z,s) analytically:

$$\min_{\|\beta\|_\infty \leq M} \frac{1}{2}\|y - X\beta\|_2^2 + \sum_{i=1}^p \psi_i(\beta_i; F_0, F_1).$$

Here ψ_i is coordinate-separable and piecewise quadratic. On F_1, $\psi_i \equiv 0$; on F_0, it forces $\beta_i = 0$; on free coordinates it interpolates between a reverse-Huber and $|\beta_i|$.
Key property: the relaxation is convex and has the same per-coordinate structure for every node. Only the fixing masks change.

System design: CPU tree, GPU node subproblems

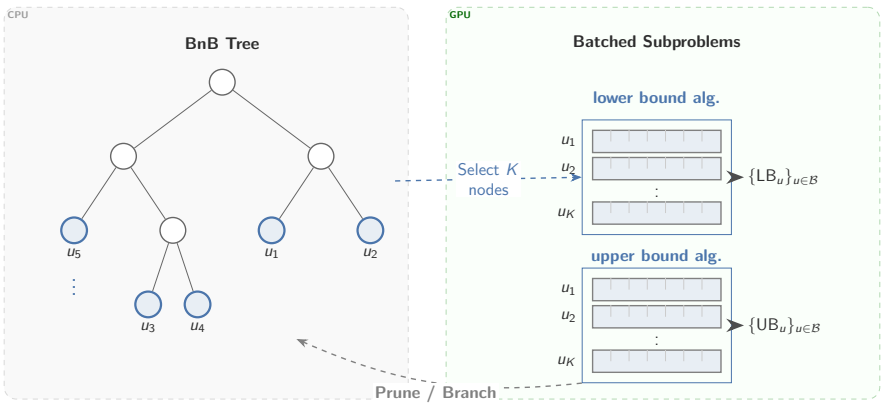


Figure. A batch $\mathcal{B}$ of K active CPU-side nodes is mapped to batched GPU lower- and upper-bound routines, then returned to the CPU for pruning and branching.

Why this split? The CPU handles irregular tree operations: maintaining the open-node queue, applying branching rules, and updating global bounds. The GPU handles the uniform numerical kernels: node lower bounds and feasible refinements.

Why batching preserves correctness

For every open node u, the GPU routine returns a valid lower bound LB_u on the node relaxation and a feasible upper bound UB_u on the original $\ell_0\ell_2$ problem. Therefore nodes with $LB_u \geq UB$ are safely pruned, and at termination the incumbent is globally optimal.

Intuition. Batching only changes the order in which nodes are processed. The BnB partition of the feasible set and the validity of the lower bounds are independent of the batch size.

ADMM lower-bound solver

Introduce a copy b of β to decouple the quadratic loss from the coordinate-separable nonsmooth term:

$$\min_{b,\beta} \frac{1}{2}\|y - Xb\|_2^2 + \sum_i \psi_i(\beta_i) + I_{\|\beta\|_\infty\leq M}(\beta) \quad \text{s.t. } b = \beta.$$

The ADMM updates are

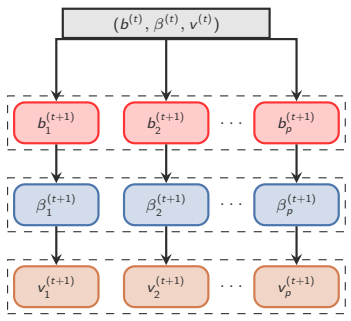
$$b^{t+1} = (X^\top X + \rho I)^{-1}(X^\top y + \rho\beta^t - v^t),$$
$$\beta_i^{t+1} = \mathrm{prox}_{\psi_i + I_{[-M,M]}}\left(b_i^{t+1} + v_i^t/\rho\right),$$
$$v^{t+1} = v^t + \rho(b^{t+1} - \beta^{t+1}).$$

The b-update uses the shared inverse $D = (X^\top X + \rho I)^{-1}$, while the β- and v-updates are coordinate-wise.

ADMM lower-bound solver – SIMD parallelism

Each node update is a regular sequence of dense linear algebra plus elementwise thresholding, rather than an irregular tree operation. **All three updates are SIMD-friendly**, which makes batching tractable.

$$b^{(t+1)} = \arg\min_b L_\rho(b, \beta^{(t)}, v^{(t)}),$$
$$\beta^{(t+1)} = \arg\min_\beta L_\rho(b^{(t+1)}, \beta, v^{(t)}),$$
$$v^{(t+1)} = v^{(t)} + \rho\left(b^{(t+1)} - \beta^{(t+1)}\right)$$



Synthetic benchmark: large speedups over CPU and L0BnB

n	p	**GPUBnB**	CPUBnB	L0BnB	MOSEK
10^3	10^4	23.7	149	46.9	1796
10^3	$3\cdot10^4$	58.4	967	225	-
10^3	10^5	355	9182	847	-
$3\cdot10^3$	$3\cdot10^4$	67.1	1728	177	-
$3\cdot10^3$	10^5	272	8958	504	-
10^4	10^5	257	8911	1514	-

Wall-clock seconds to certified optimality; "-" indicates 80 GB memory limit exceeded.

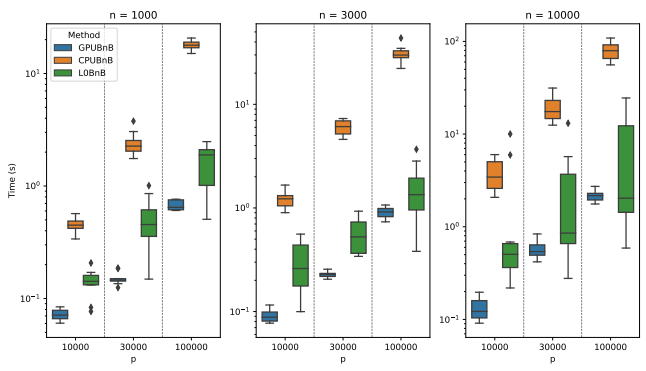


Figure. Per-node relaxation time. ADMM is less specialized than coordinate descent in L0BnB, but it is dense, regular, and GPU-friendly.

Batched node relaxations on GPU

For K open BnB nodes, stack the ADMM variables row-wise:

$$B,\ \mathcal{B},\ V \in \mathbb{R}^{K\times p}.$$

Each row is one node relaxation and each column is one coefficient. The batched lower-bound solve is the same three ADMM updates applied to matrices:

$$B^{t+1} = \left(C + \rho\mathcal{B}^t - V^t\right)D \qquad \text{dense auxiliary update}$$

$$\mathcal{B}^{t+1} = \delta^{(0)}\odot 0 + \delta^{(1)}\odot\Pi_{[-M,M]}\left(\frac{\rho}{\rho+2\lambda_2}\widetilde{B}\right) + \delta^{(f)}\odot T(\widetilde{B}; a, M) \qquad \text{masked primal update}$$

$$V^{t+1} = V^t + \rho(B^{t+1} - \mathcal{B}^{t+1}) \qquad \text{dual residual update}$$

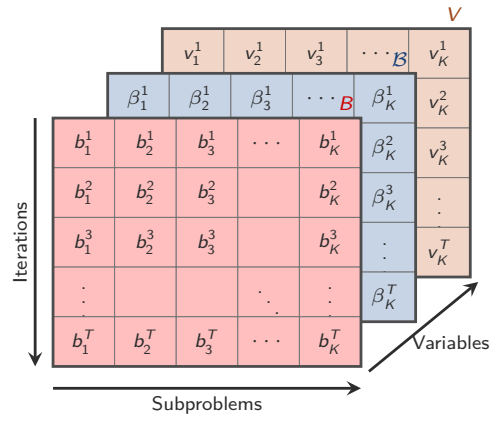
where:

$$D = (X^\top X + \rho I)^{-1}, \qquad C = \mathbf{1}_K(X^\top y)^\top, \qquad \widetilde{B} = B^{t+1} + V^t/\rho.$$

The only node-specific objects are the fixing masks

$$\delta^{(0)}, \delta^{(1)}, \delta^{(f)} \in \{0,1\}^{K\times p},$$

which encode forced-zero, forced-active, and free coordinates. Thus the GPU sees one large tensor program: a shared matrix multiply for B, a masked thresholding step for $\mathcal{B}$, and a matrix update for V.



Upper-bound solver: batched feasible solutions

For each open node $u = 1,\ldots,K$, round the relaxed indicator $\hat{z}_u$ to a support S_u. We solve:

$$\min_{\beta_{S_u}} \frac{1}{2}\|y - X_{S_u}\beta_{S_u}\|_2^2 + \lambda_2\|\beta_{S_u}\|_2^2 \quad \text{s.t.} \quad |\beta_i| \leq M,\ i \in S_u.$$

Introduce:

$$B,\ B^-,\ \widetilde{B},\ \nabla G \in \mathbb{R}^{K\times p}, \qquad \mathcal{M} \in \{0,1\}^{K\times p},$$

where $\mathcal{M}_{u,i} = 1$ iff $i \in S_u$. One projected-gradient iteration is applied to all K nodes at once:

$$\widetilde{B}^t = B^t + \frac{t}{t+3}\left(B^t - B^{t-1}\right) \qquad \text{shared momentum}$$

$$\nabla G(\widetilde{B}^t) = \left(X^\top(X(\widetilde{B}^t)^\top - y\mathbf{1}_K^\top)\right)^\top + 2\lambda_2\widetilde{B}^t,$$
$$\nabla G \leftarrow \nabla G \odot \mathcal{M}$$

$$B^{t+1} = \Pi_{[-M,M]}\left(\widetilde{B}^t - \alpha\odot\nabla G(\widetilde{B}^t)\right)\odot\mathcal{M}$$

Here $\alpha \in \mathbb{R}^K$ stores one backtracking stepsize per node and is broadcast across columns. Each GPU pass returns:

$$\{UB_u, \beta_u^{\text{feas}}\}_{u=1}^K.$$

Node parallelism improves convergence and throughput

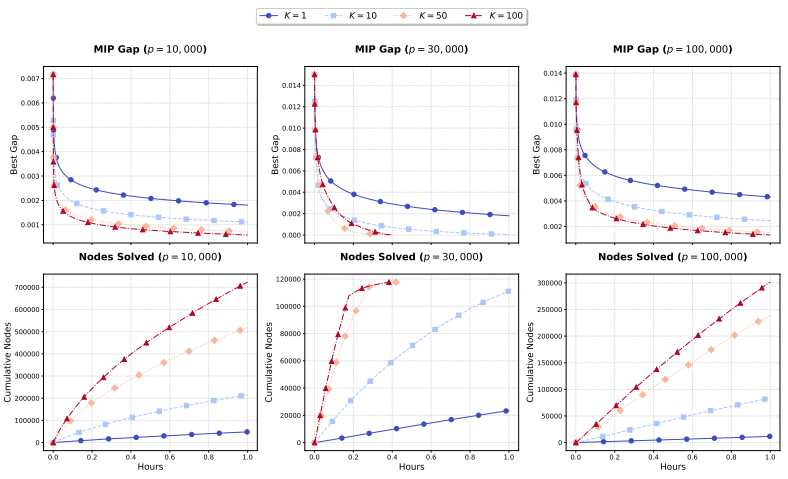


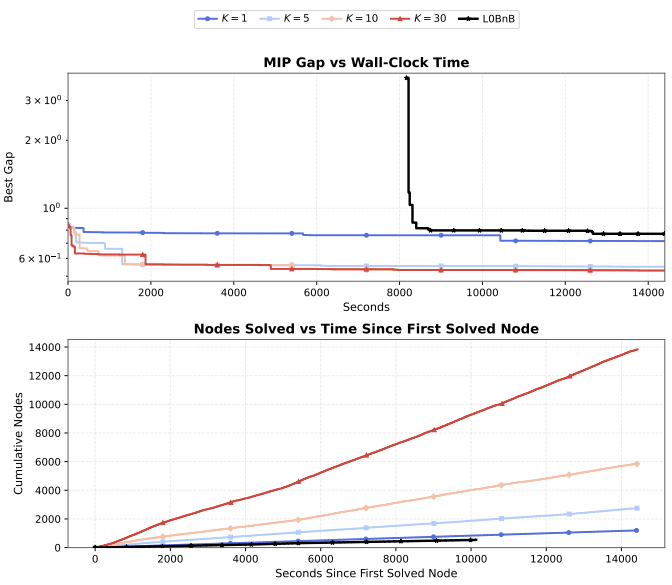
Figure. Increasing batch size K reduces the MIP gap at a given wall-clock budget and increases nodes-per-second throughput.

Memory cost is modest. Because X, D, and y are stored once and shared across all nodes in the batch, going from $K = 1$ to $K = 100$ inflates GPU memory by far less than $100\times$.

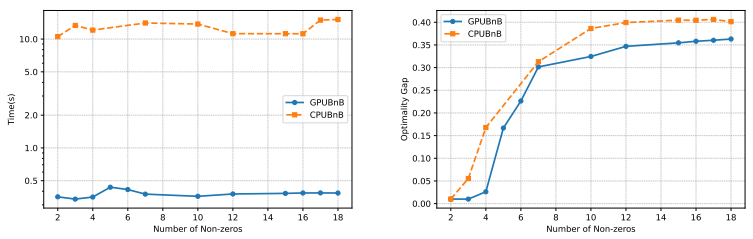
K	$p = 10^4$ GB	$p = 10^4$ Inc.	$p = 3\cdot10^4$ GB	$p = 3\cdot10^4$ Inc.	$p = 10^5$ GB	$p = 10^5$ Inc.
1	0.636	1.00×	1.285	1.00×	2.824	1.00×
10	0.652	1.03×	1.310	1.02×	2.983	1.06×
50	0.726	1.14×	1.459	1.14×	3.693	1.31×
100	0.816	1.28×	1.696	1.32×	4.580	1.62×

Figure. GPU memory in GB and multiplicative increase relative to $K = 1$.

Real-world datasets



Census (US Census tracts, n, $p \approx 2\cdot10^5, 10^5$): batching solves over $10\times$ more nodes per unit time and maintains substantially lower MIP gaps throughout the run.



UJIIndoorLoc (indoor positioning, n, $p \approx 2\cdot10^4, 5\cdot10^2$): GPU node solves are roughly $30\times$ faster than the CPU baseline.