D-PDLP: SCALING PDLP TO DISTRIBUTED MULTI-GPU SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

We present a distributed framework of the Primal-Dual Hybrid Gradient (PDHG) algorithm for solving massive-scale linear programming (LP) problems. Although PDHG-based solvers perform strongly on single GPUs, computational throughput can become a performance bottleneck for sufficiently high-workload instances. To overcome this bottleneck, we propose **D-PDLP**, a Distributed PDLP framework and, to our knowledge, the first to accelerate the complete PDLP solver across multiple GPUs. By combining two-dimensional partitioning with our load-balancing strategy, D-PDLP distributes primal–dual computation across GPUs while maintaining efficient GPU execution. To understand how D-PDLP scales across GPUs, we derive a PDLP-specific cost model that relates computation and communication costs to the processor-grid configuration and GPU count. The model guides resource selection, with case studies on contrasting sparse workloads demonstrating its predictive value. Extensive experiments on standard LP benchmarks, including MIPLIB and Mittelmann instances, and huge-scale real-world datasets show that our implementation, built upon cuPDLPx (Lu et al., 2025), achieves approximately $6\times$ speedup on eight GPUs for computationally intensive instances while maintaining numerical accuracy. Our solver is open source and available at https://anonymous.4open.science/r/D-PDLP_SUBMIT-962B.

1 INTRODUCTION

Large-scale linear programming (LP) is central to numerous machine learning (Agarwal et al., 2018; Ben-Tal et al., 2013; Taskar et al., 2005; Joachims et al., 2009), operations research (Ahuja et al., 2014; Gallego & Van Ryzin, 1997; Talluri & Van Ryzin, 2006), and data-driven decision-making problems, including resource allocation (Hibiki, 2006; Kelly et al., 1998), market equilibrium computation Eisenberg & Gale (1959); Orlin (2010), and multi-stage stochastic programming (Hibiki, 2006; Gangammanavar & Sen, 2021). Modern applications, especially in machine learning, often involve millions or even billions of variables and constraints, rendering classical interior-point methods impractical due to their unfavorable memory footprint and limited parallel scalability.

First-order primal–dual methods, such as PDLP (Applegate et al., 2021), cuPDLP.jl (Lu & Yang, 2023a), cuPDLP-C (Lu et al., 2023), cuPDLPx (Lu et al., 2025), and HPR-LP (Chen et al., 2026) have recently emerged as a promising alternative for solving large-scale LPs, owing to their low per-iteration complexity and amenability to parallelization. Among them, the PDLP family based on the primal–dual hybrid gradient (PDHG) framework has demonstrated strong empirical performance and robustness across a wide range of benchmark instances.

We propose **D-PDLP**, a Distributed PDLP framework that, to our knowledge, is the first to accelerate the complete PDLP solver across multiple GPUs for large-scale LPs. While distributed first-order methods historically suffer from prohibitive communication bottlenecks across loosely coupled nodes, D-PDLP is explicitly designed to exploit the massive intra-node bandwidth of modern dense GPU architectures (e.g., NVLink). Our approach spatially decomposes primal and dual updates across a grid of workers, executing partial Sparse Matrix-Vector Multiplications (SpMV) locally before synchronizing via high-speed `AllReduce` collectives. This tightly coupled design effectively mitigates communication bottlenecks, translating raw hardware bandwidth into algorithmic speedups.

Scaling PDHG effectively is non-trivial. Real-world LPs often exhibit irregular sparsity patterns that cause severe load imbalances under naive partitioning. We propose a load-balancing strategy

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

that combines **block-wise random permutation** with **nonzero-aware matrix partitioning**. The permutation redistributes rows and columns in blocks to balance workloads while preserving structure for efficient GPU SpMV, and the subsequent partitioning balances nonzero counts along each grid axis.

Our work makes two main contributions:

- **Multi-GPU acceleration of PDLP.** To our knowledge, D-PDLP is the first framework to accelerate the complete PDLP solver across multiple GPUs, including primal-dual updates, convergence checks, and restart operations. We combine two-dimensional partitioning with block-wise random permutation and nonzero-aware partitioning to balance workloads while maintaining efficient GPU SpMV. Experiments demonstrate approximately $6\times$ speedup on eight GPUs for computationally intensive LP instances. We further extend the framework to convex QPs with diagonal objective matrices, achieving up to $4.7\times$ speedup on eight GPUs over the single-GPU baseline on large-scale Lasso formulations.
- **Theoretical analysis and practical guidance.** We derive a PDLP-specific cost model that accounts for matrix and vector memory traffic, collective communication, and synchronization latency. The analysis explains how these costs shape multi-GPU scaling and guides processor-grid and GPU-count selection. Case studies on contrasting sparse workloads connect the predictions to measured performance. Together, the analysis and empirical evidence provide a theoretical foundation and practical guidance for understanding and developing multi-GPU PDLP algorithms.

Due to page limits, a comprehensive review of the related works is provided in Appendix A.

2 PRELIMINARIES

In this section, we briefly review the general formulation of linear programming in Section 2.1, the primal–dual hybrid gradient algorithm in Section 2.2, and the cuPDLPx solver (Lu et al., 2025), on which our implementation is based, in Section 2.3.

2.1 LINEAR PROGRAMMING

We consider the following linear programming problem

$$\min_{x\in\mathcal{X}} \; c^\top x \quad \text{subject to } Ax \in \mathcal{S} \tag{1}$$

where $A \in \mathbb{R}^{m\times n}$, $c \in \mathbb{R}^n$, the feasible set $\mathcal{X} := \{x \in \mathbb{R}^n : \ell_v \le x \le u_v\}$ with bounds $\ell_v \in (\mathbb{R} \cup \{-\infty\})^n$ and $u_v \in (\mathbb{R} \cup \{\infty\})^n$, and the constraint range $\mathcal{S} := \{s \in \mathbb{R}^m : \ell_c \le s \le u_c\}$ with $\ell_c \in (\mathbb{R}\cup\{-\infty\})^m$ and $u_c \in (\mathbb{R}\cup\{\infty\})^m$. The corresponding Lagrangian yields the equivalent saddle-point formulation

$$\min_{x\in\mathcal{X}} \max_{y\in\mathcal{Y}} \; \mathcal{L}(x,y) \coloneqq c^T x - y^T A x - p(-y; \ell_c, u_c), \tag{2}$$

The minus sign in the coupling term fixes our dual convention: lower-bound constraints have nonnegative multipliers, while upper-bound constraints have nonpositive multipliers. Here $p(\cdot\,; l, u)$ denotes a piecewise linear function defined as

$$p(y; l, u) = u^\top y^+ - l^\top y^-$$

with $y^+ = \max\{y, 0\}$ and $y^- = \max\{-y, 0\}$ being the positive and negative parts of y, respectively, and the dual feasible set $\mathcal{Y} \subseteq \mathbb{R}^m$ is Cartesian product whose i-th components are determined by the boundedness of the primal constraints:

$$\mathcal{Y}_i := \begin{cases} \{0\} & \text{if } (\ell_c)_i = -\infty, (u_c)_i = \infty, \\ \mathbb{R}^- & \text{if } (\ell_c)_i = -\infty, (u_c)_i \in \mathbb{R}, \\ \mathbb{R}^+ & \text{if } (\ell_c)_i \in \mathbb{R}, (u_c)_i = \infty, \\ \mathbb{R} & \text{otherwise.} \end{cases}$$

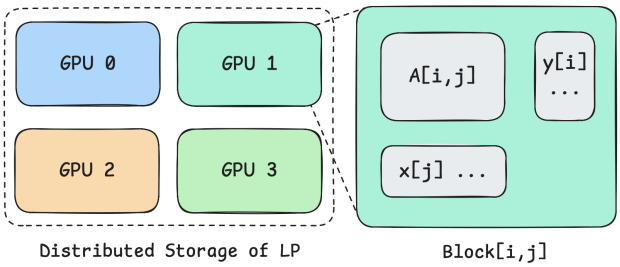

Figure 1: Distributed memory layout under 2D grid partitioning. **Left:** A logical 2×2 device mesh illustrating the global topology. **Right:** Local data stored on the device at grid coordinate (i, j). The constraint matrix is stored as block $A_{[i,j]}$, primal variables $x_{[j]}$, and dual variables $y_{[i]}$. This layout enables local computation of $A_{[i,j]}x_{[j]}$ and $A_{[i,j]}^\top y_{[i]}$ prior to reduction.

2.2 The PDHG Algorithm

Applying PDHG to the saddle-point problem in (2) gives:

$$\begin{aligned} x^{k+1} &= \text{proj}_{\mathcal{X}}\left(x^k - \tau(c - A^\top y^k)\right) \\ y^{k+1} &= y^k - \sigma v^{k+1} - \sigma \cdot \text{proj}_{-\mathcal{S}}\left(\sigma^{-1} y^k - v^{k+1}\right) \end{aligned} \tag{3}$$

where $v^{k+1} := A(2x^{k+1} - x^k)$, and τ and σ denote the primal and dual step sizes, respectively. The step sizes can be reparameterized as $\tau = \frac{\eta}{\omega}$ and $\sigma = \eta\omega$, where η denotes the overall step size and ω is the primal weight that balances the primal and dual updates.

2.3 Algorithmic Enhancements in cuPDLPx

Our distributed implementation builds upon the state-of-the-art GPU-based LP solver cuPDLPx (Lu et al., 2025). To improve convergence behavior and numerical stability, cuPDLPx incorporates several algorithmic enhancements, including reflected Halpern iteration scheme, adaptive restarting strategy, and primal weight updates. Detailed descriptions of these methods are provided in Appendix C.

3 Distributed PDLP

To scale PDHG beyond single-device computation and unlock the computational throughput of modern clusters, we adopt a 2D partitioning strategy for the constraint matrix A. This design enables the flexible distribution of variables and constraints across multiple GPUs, allowing the solver to effectively exploit massive parallelism to accelerate the solving phase.

3.1 Problem Partition

We adopt a variant of the named-axis notation to describe the data layout across hardware resources (Austin et al., 2025). Specifically, we assume a 2D grid of $N_{\text{device}} = |R| \times |C|$ devices, where each axis is assigned a semantic role:

- **Row axis** (R): Corresponds to partitioning along the rows of A, with cardinality $|R|$.
- **Column axis** (C): Corresponds to partitioning along the columns of A, with cardinality $|C|$.

A partition $\mathcal{P}$ maps tensor dimensions to physical grid axes. For the sparse constraint matrix $A \in \mathbb{R}^{m \times n}$, we divide the row and column index sets into $|R|$ and $|C|$ disjoint subsets, respectively, forming a two-dimensional grid of local sparse blocks (Figure 1). Each block $A_{[i,j]}$ contains the entries belonging to the i-th row partition and the j-th column partition and is stored in the local memory of device (i, j).

$$A \to \begin{bmatrix} A_{[1,1]} & \cdots & A_{[1,|C|]} \\ \vdots & \ddots & \vdots \\ A_{[|R|,1]} & \cdots & A_{[|R|,|C|]} \end{bmatrix} \tag{4}$$

Local Shapes and Sparsity. In dense tensor settings, as in LLM model parallelism such as Megatron-LM (Shoeybi et al., 2020), a partition $\mathcal{P}$ typically assigns each device a uniform block of size $(m/|R|,\ n/|C|)$ (Austin et al., 2025). In contrast, our setting features irregular local shapes, where device (i, j) stores submatrix $A_{[i,j]}$ of dimensions (m_i, n_j), with $\sum_i m_i = m$ and $\sum_j n_j = n$. Moreover, the number of nonzero entries may vary significantly across devices. This irregularity motivates the load-balancing heuristics introduced in Section 4. A deeper analysis of the differences between dense model parallelism and our framework is provided in Appendix D.

Vector Partitioning and Replication. To minimize communication during local computations, we align the distribution of all vectors with the two-dimensional partitioning of the constraint matrix. Primal-space vectors in $\mathbb{R}^n$, including the decision variable x and objective coefficients c, are partitioned along the column axis C. The j-th block, denoted by $x_{[j]}, c_{[j]} \in \mathbb{R}^{n_j}$, induces a local primal feasible set

$$\mathcal{X}_{[j]} := \{z \in \mathbb{R}^{n_j} : (\ell_v)_{[j]} \le z \le (u_v)_{[j]}\}.$$

Each block is vertically replicated across all devices in column j, ensuring that any device holding $A_{[i,j]}$ can compute $A_{[i,j]}x_{[j]}$ and $\mathrm{proj}_{\mathcal{X}_{[j]}}$ without communication.

Similarly, dual-space vectors in $\mathbb{R}^m$, including the dual variable y, are partitioned along the row axis R, with blocks $y_{[i]} \in \mathbb{R}^{m_i}$ and corresponding local constraint sets

$$\mathcal{S}_{[i]} := \{b \in \mathbb{R}^{m_i} : (\ell_c)_{[i]} \le b \le (u_c)_{[i]}\}.$$

These blocks are horizontally replicated across all devices in row i, enabling local computation of $A^\top_{[i,j]}y_{[i]}$ and dual projections $\mathrm{proj}_{-\mathcal{S}_{[i]}}$. By orthogonally replicating primal and dual variables, communication is limited to reductions of partial matrix–vector products, maximizing the efficiency of local update kernels.

3.2 Distributed PDHG Algorithm

The iterative updates necessitate global synchronization of partial matrix–vector products computed on local data shards. We implement this synchronization using the NCCL `AllReduce` collective. Formally, for a named grid axis X, the operator $\mathrm{AllReduce}_X(\cdot)$ aggregates partial vectors via summation across all devices along axis X and broadcasts the accumulated result back to them. The complete distributed procedure is outlined in Algorithm 1.

3.2.1 Communication Operators

We formalize the communication pattern by defining the operator $\mathrm{AllReduce}_X$ acting on a distributed tensor T, where $T_{[i,j]}$ denotes the block stored locally on device (i, j). Depending on the aggregation axis, we define three variants:

$$\mathrm{AllReduce}_R(T_{[i,j]}) = \sum_{k=1}^{|R|} T_{[k,j]}, \quad \mathrm{AllReduce}_C(T_{[i,j]}) = \sum_{k=1}^{|C|} T_{[i,k]} \tag{5}$$

$$\mathrm{AllReduce}_G(T_{[i,j]}) = \sum_{k=1}^{|R|}\sum_{l=1}^{|C|} T_{[k,l]} \tag{6}$$

Here, $\mathrm{AllReduce}_R$ aggregates data across the row axis, $\mathrm{AllReduce}_C$ across the column axis, and $\mathrm{AllReduce}_G$ across the entire device grid. In our distributed PDHG implementation, $\mathrm{AllReduce}_R$ aggregates partial products for $A^\top y$, while $\mathrm{AllReduce}_C$ sums partial results for Ax.

3.2.2 Distributed Primal–Dual Updates

For the primal update, the gradient term $A^\top y^k$ is computed via partial products $A^\top_{[i,j]}y^k_{[i]}$, followed by an `AllReduce` across the row axis (R):

$$[A^\top y^k]_{[j]} = \mathrm{AllReduce}_R\big(A^\top_{[i,j]}y^k_{[i]}\big).$$

The primal block update is

$$x^{k+1}_{[j]} = \mathrm{proj}_{\mathcal{X}_{[j]}}\left(x^k_{[j]} - \tau\big(c_{[j]} - [A^\top y^k]_{[j]}\big)\right).$$

For the dual update, let $\bar{x} = 2x^{k+1} - x^k$. The partial products $A_{[i,j]}\bar{x}_{[j]}$ are aggregated via an `AllReduce` across the column axis (C):

$$z_{[i]} = \mathrm{AllReduce}_C\big(A_{[i,j]}\bar{x}_{[j]}\big),$$

followed by the local dual update

$$y_{[i]}^{k+1} = y_{[i]}^k - \sigma z_{[i]} - \sigma \cdot \mathrm{proj}_{-\mathcal{S}_{[i]}}\big(\sigma^{-1}y_{[i]}^k - z_{[i]}\big).$$

This design ensures that the computationally intensive partial matrix-vector products $A_{[i,j]}\bar{x}_{[j]}$ and $A_{[i,j]}^\top y_{[i]}$ are executed entirely within local memory without any inter-device communication. Synchronization is deferred until the aggregation phase, allowing the SpMV kernels to make greater use of the aggregate GPU resources through independent local computation.

The distributed updates are algebraically equivalent to the single-GPU updates in exact arithmetic. Appendix L.4 validates numerical accuracy across GPU configurations and reports matching iteration counts on the tested instances.

KKT evaluations and restart checks run periodically, with primal-weight updates performed at restarts. Their additional synchronization costs are amortized over the intervening core iterations and typically account for a small fraction of the total runtime. Appendix E details the distributed implementation of these operations.

3.3 Theoretical Cost Model

Our theoretical cost model focuses on the primal, dual, and Halpern updates, which are performed at every iteration and dominate the computational and communication costs. Their low arithmetic intensity motivates a memory-bandwidth-based cost model. We assume balanced nonzero counts across devices, motivated by the block-wise random permutation and nonzero-aware partitioning strategies introduced in Section 4. Appendix F provides the detailed derivation.

Let $N_{\mathrm{device}} = |R| \times |C|$ denote the total number of physical GPUs. The theoretical per-iteration time, T_{step}, is:

$$\begin{aligned} T_{\mathrm{step}}(|R|,|C|) = &\frac{\mathcal{C}_{\mathrm{spmv}} \cdot \mathrm{nnz}}{B_{\mathrm{mem}} \cdot N_{\mathrm{device}}} + \frac{\mathcal{C}_{\mathrm{vec}}}{B_{\mathrm{mem}}}\left(\frac{n}{|C|} + \frac{m}{|R|}\right) \\ &+ \frac{\mathcal{C}_{\mathrm{net}}}{B_{\mathrm{net}}}\left(\frac{n|R| + m|C| - (m+n)}{N_{\mathrm{device}}}\right) + n_{\mathrm{hop}}\beta_{\mathrm{sync}}\big(\mathbb{I}(|R|>1) + \mathbb{I}(|C|>1)\big) \end{aligned} \tag{7}$$

where nnz denotes the number of non-zero elements, and m, n represent the dimensions of the constraint matrix. The hardware parameters B_{mem} and B_{net} represent the HBM and interconnect bandwidths used in the model, approximated here by nominal peak values. The parameters n_{hop} and β_{sync} denote an effective topology multiplier and the base latency per active collective, respectively. The coefficients $\mathcal{C}$ represent the bytes transferred per unit of data. Detailed theoretical derivations for each component, along with explicit definitions of all parameters, are deferred to Appendix F.

Equation (7) shows that the matrix-dependent computation time decreases as N_{device}^{-1}. With balanced workloads and a fixed grid aspect ratio, the vector-operation time and bandwidth-dependent communication time scale as $\mathcal{O}(N_{\mathrm{device}}^{-1/2})$, while synchronization latency limits further speedup. This model guides our **AUTO** scheduler, which evaluates candidate configurations before execution and selects the one with the lowest predicted per-iteration time. Experiments in Section 5 demonstrate the effectiveness of this strategy across different problem scales.

Case Studies of Multi-GPU Acceleration. The model explains why LP instances benefit differently from additional GPUs. For `sdm`, matrix-dependent computation dominates, and distributing this work yields substantial speedup. For `zib03`, vector operations account for a larger share of the cost, limiting the benefit of further parallelization. These contrasting cases illustrate how the model identifies the factors governing multi-GPU performance. In practice, it uses problem dimensions, nonzero counts, and hardware parameters to guide grid selection and estimate the benefit of additional GPUs before execution. Appendix F.4 presents the quantitative analysis and comparisons with measured performance.

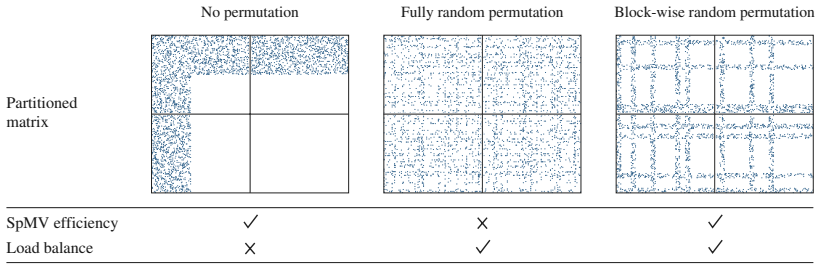

Figure 2: Comparison of permutation strategies. Blue dots show the nonzeros of the same illustrative matrix under natural ordering, fully random permutation, and block-wise random permutation; solid lines indicate device boundaries. The qualitative comparison below summarizes the trade-off between SpMV efficiency and load balance.

4 Load-Balancing Strategies

Achieving high parallel efficiency in distributed sparse optimization requires addressing three interconnected challenges: minimizing communication overhead relative to computation, mitigating load imbalance caused by irregular sparsity, and preserving memory locality for efficient GPU execution. We address these through a three-stage partition strategy: adaptive grid topology selection, block-wise random permutation, and nonzero-aware matrix partitioning. These steps are performed only once before iterative solving, so their cost is amortized over the subsequent iterations. Measurements on four representative instances show that permutation and partitioning together consume only 0.14%–4.47% of the solving-time reduction from one to eight GPUs (Appendix J.2).

4.1 Adaptive Grid Topology Selection

To minimize the per-iteration communication overhead $\mathcal{O}(n/|C| + m/|R|)$, described in the Equation 7, the $R \times C$ processor grid must dynamically adapt to the dimensions of the constraint matrix $A \in \mathbb{R}^{m \times n}$. To reduce primal and dual data traffic, we choose a feasible grid whose aspect ratio is as close as possible to the matrix's on a logarithmic scale. Specifically, for a given device count N_{device}, we configure the grid by selecting the factor pair $(|R|, |C|)$ that satisfies $|R| \times |C| = N_{\text{device}}$ while minimizing $|\log(|R|/|C|) - \log(m/n)|$, equivalently minimizing $n/|C| + m/|R|$ over these pairs. AUTO then evaluates the selected integer grids in the full cost model, including collective latency, to compare candidate device counts.

4.2 Block-Wise Random Permutation

Irregular sparsity patterns in real-world LP instances, such as staircase structures or clustered nonzeros, can lead to severe computational load imbalance under a static 2D partition (Figure 2). While a *Fully Random Permutation* of rows and columns can statistically homogenize the nonzero distribution (Figure 2, Middle), it disrupts local block structure and can reduce the efficiency of local SpMV operations. Our ablation results show that preserving this structure through block-wise permutation improves aggregate solver performance (Appendix J.1).

Block-wise Random Permutation. To reconcile global load balancing with local SpMV efficiency, we introduce a *Block-Wise Random Permutation* strategy (Figure 2, Right). Instead of permuting individual indices, we partition the global rows and columns into contiguous blocks of fixed size B and apply random permutations over these blocks. This approach redistributes fine blocks across the $|R| \times |C|$ device grid to improve load balance, while preserving the dense micro-structures within each $B \times B$ block for efficient GPU SpMV execution. We use $B = 256$ as a default within the broad performance plateau observed in Appendix L.3. Appendix L.2 gives the complete row/column mapping, including boundary blocks and solution recovery.

4.3 NONZERO-AWARE MATRIX PARTITIONING

Following permutation, we partition the matrix independently along each grid axis $X \in \{R, C\}$. We select boundaries $\{p_0, \ldots, p_{|X|}\}$ such that the nonzero count in each slice satisfies $\sum_{k=p_i}^{p_{i+1}-1} \text{nnz}_k \approx \text{nnz}(\tilde{A})/|X|$. Combined with block-wise random permutation, this axis-wise partitioning improves aggregate solving performance (Appendix J.1).

5 NUMERICAL EXPERIMENTS

In this section, we evaluate D-PDLP on standard LP benchmarks and massive-scale instances to assess its multi-GPU performance and the effectiveness of AUTO, followed by ablation studies on the proposed load-balancing strategies.

5.1 EXPERIMENTAL SETUP

We evaluate our distributed solver on a single node with eight NVIDIA H100 (80GB) GPUs fully interconnected via NVLink. All LP experiments in this section use the Julia implementation. We additionally provide a C/CUDA version that is easier to use, requires fewer dependencies, and delivers higher performance. Hardware and implementation details are provided in Appendix G; the C/CUDA comparison with Julia is reported in Appendix I, and supplementary C/CUDA experiments appear in the appendices.

Evaluation Metric. We aggregate performance across instances using the standard Shifted Geometric Mean (SGM) with a shift of $k = 10$ seconds (Mittelmann, 2020), defined as $\text{SGM10} = \exp\left(\frac{1}{N}\sum_{i=1}^{N} \ln(t_i + 10)\right) - 10$, where t_i is the solving time for the i-th instance. We compare all GPU configurations and baselines using solving time, excluding input loading and one-time preprocessing. Permutation and partitioning costs are reported separately in Appendix J.2.

Automatic GPU Count (AUTO). Before execution, AUTO selects the configuration with the lowest model-predicted per-iteration cost among $\{1, 2, 4, 8\}$ GPUs. It evaluates Eq. (7) using the instance dimensions and nonzero count (m, n, nnz), with $B_{\text{mem}} = 3350$ GB/s, $B_{\text{net}} = 450$ GB/s, and $\beta_{\text{sync}} = 0.01$ ms. Appendix F details the parameter estimates; Appendix L.5 explains how fixed GPU counts isolate the effects of load balancing in the ablation study.

Benchmark Datasets We evaluate standard LP benchmarks and huge-scale application instances.

- **Standard Benchmarks.** For initial validation and strong scaling analysis, we employ the MIPLIB 2017 (Gleixner et al., 2021) relaxation set, comprising 383 diverse mixed-integer programming instances solved as linear programs, alongside Mittelmann's LP benchmark (Mittelmann & Spellucci, 2005), which contains 50 challenging public instances. We restrict our focus to feasible and bounded instances, setting the termination tolerance to $\epsilon = 10^{-4}$.
- **Huge-Scale Instances.** Moving beyond standard benchmarks, we probe the architecture using a curated, huge-scale dataset. This collection includes the `zib03` instance from (Koch et al., 2022), large-scale PageRank formulations (Nesterov, 2014) (10^7 nodes), and real-world network datasets (e.g., LiveJournal) (Leskovec & Krevl, 2014). We also incorporate extremely large multicommodity flow (`mcf`) and synthetic design matching (`sdm`) instances. Furthermore, the suite includes Quadratic Assignment Problem (QAP) relaxations based on Adams-Johnson linearization (Adams & Johnson, 1994) (such as `tai50b`, `lipa50b`) and Unit Commitment problems (`ds1`, `ds2`). For these instances, we adopt a tighter accuracy of $\epsilon = 10^{-6}$ (except `mcf` at 10^{-4}).

Appendix L.4 specifies the timing convention and reports 10^{-8} accuracy checks and repeated-run statistics.

5.2 PERFORMANCE ON BENCHMARK DATASETS

Appendix L.5 explains the complementary workload regimes covered by the two tables. Tables 1 and 2 demonstrate the scalability of our distributed, grid-partitioned solver. For the smallest problem set, specifically the *MIPLIB Small* subset with fewer than one million nonzeros, performance peaks

early at just 2 GPUs (5.77s) and noticeably degrades when scaling to 8 GPUs (6.79s). Small problems may underutilize even a single GPU, leaving limited scope for gains from additional devices while introducing inter-device `AllReduce` overhead.

In the *MIPLIB Large* category, the distributed implementation reduces SGM10 from 90.34s on a single GPU to 56.81s on 8 GPUs, demonstrating substantial performance gains from multi-GPU execution. Crucially, our AUTO scheduler successfully navigates these diverse architectural trade-offs without empirical profiling. By circumventing the latency wall on small instances while fully exploiting parallelism for massive problems, AUTO secures the fastest SGM10 times across both benchmarks, achieving 18.38s on Mittelmann and an overall best of 7.31s on MIPLIB, demonstrating the practical effectiveness of our theoretical cost model.

Table 1: Performance on standard benchmarks. 'Solved' indicates instances solved within the time limit (3600s for Mittelmann, 1200s for MIPLIB). Best results are **bold**, second-best are underlined.

		Number of GPUs				
Dataset	Metric	1	2	4	8	AUTO
Mittelmann	Solved	**46**	**46**	**46**	**46**	**46**
	Time (s)	21.26	19.00	18.58	19.24	**18.38**
MIPLIB Small (100K - 1M)	Solved	**264**	**264**	**264**	263	**264**
	Time (s)	5.82	**5.77**	6.12	6.79	5.81
MIPLIB Med. (1M - 10M)	Solved	**92**	**92**	**92**	**92**	**92**
	Time (s)	7.21	**6.68**	6.70	7.11	6.76
MIPLIB Large (> 10M)	Solved	13	14	**20**	**20**	**20**
	Time (s)	90.34	75.30	62.07	**56.81**	**56.81**
MIPLIB Overall	Solved	369	370	**376**	375	**376**
	Time (s)	7.80	7.47	7.60	8.14	**7.31**

The advantage is decisive on huge-scale instances (Table 2). We observe **near-linear acceleration** on computationally intensive problems: `mcf_2500` achieves speedups of $3.1\times$ (4 GPUs) and approximately $6\times$ (8 GPUs), while `sdm_50k` attains $6.2\times$ on 8 GPUs. Even the highly structured `zib03` sees a $3.3\times$ boost. Guided by the theoretical cost model, our AUTO scheduler selects eight GPUs as the configuration with the lowest predicted cost for these instances. These results confirm that the grid-partitioned architecture effectively translates additional compute resources into proportional reductions in solving time. A solver-time comparison with representative external solvers is provided in Appendix H, with published CPU-solver context in Appendix L.6.

5.3 Ablation Study on Load-Balancing Strategies

To isolate the performance gains from our load-balancing techniques, we evaluated the individual and combined effects of matrix permutation and partitioning strategies. As detailed in Appendix J.1, the baseline approach (*No Permutation* combined with *No NNZ* uniform partitioning) suffers from severe load imbalance, resulting in an aggregate SGM10 of 220.08s. In contrast, combining our proposed *Block Random* permutation with *Yes NNZ* nonzero-aware partitioning effectively redistributes dense regions while preserving local block structures for efficient SpMV execution. This synergistic configuration reduces the SGM10 to 75.85s, improving on the stronger *Full Random + Yes NNZ* baseline (87.96s) by 13.8%. Adding nonzero-aware cuts to *Block Random* alone yields a 4.6% reduction (79.51s to 75.85s). We use the combined configuration as the default outside this ablation. Appendix L.3 reports the block-size sweep, and Appendix J.2 reports the one-time preprocessing costs.

6 Extensions to Diagonal QPs: D-PDHCG

While D-PDLP is primarily designed for linear programming, the framework naturally extends to convex quadratic programming (QP) problems featuring a diagonal quadratic objective matrix Q. The primal update becomes:

$$x^{k+1} = \arg\min_{x \in \mathcal{X}} \left\{ \frac{1}{2} x^\top Q x + c^\top x - x^\top A^\top y^k + \frac{1}{2\tau_k} \|x - x^k\|_2^2 \right\}. \tag{8}$$

Crucially, because Q is diagonal, this subproblem remains fully separable across the dimensions of x and admits a highly efficient closed-form solution. Under our distributed memory layout, the diagonal entries of Q are partitioned along the column axis in exact alignment with the primal variable $x_{[j]}$. As a result, the primal update can be executed within local device memory, introducing zero additional communication overhead compared to the LP version.

Table 2: Solving time (s) on massive-scale instances. Best results are **bold**; second-best are underlined.

					Number of GPUs		
Source	Instance	m	n	nnz	1	4	8 (AUTO)
Koch	zib03	19,731,970	29,128,799	104,422,573	812	336	**245**
Pagerank	rand-10m-nodes	10,000,001	10,000,000	79,999,982	10	5	**4**
	com-livejournal	3,997,963	3,997,962	77,358,302	3	2	**1**
	soc-livejournal1	4,847,572	4,847,571	78,170,533	2	1	**1**
multicommodity-flow	mcf_2500_100_500	1,512,600	126,250,100	253,750,100	2943	935	**504**
	mcf_5000_50_500	2,775,050	126,250,050	253,750,050	9114	2994	**1677**
	mcf_5000_100_250	1,775,100	127,500,100	257,500,100	9173	3159	**1732**
design-matching	sdm_50k_500k_15_10	5,500,135	10,000,000	690,000,000	377	92	**61**
QAP	wil50	3,437,600	6,252,500	19,125,000	43	31	**25**
	lipa50a				27	20	**17**
	lipa50b				28	21	**17**
	tai50a				30	21	**18**
	tai50b				82	52	**43**
Unit Com.	ds1	641,037	659,145	21,577,566	279	143	**122**
	ds2				1288	691	**591**
SGM10					186.6	101.2	**77.7**

Our D-PDHCG implementation builds on PDHCG-II (Li et al., 2026b). To demonstrate the efficacy of this extension, Table 3 presents a solver-time comparison on large-scale LASSO instances against several state-of-the-art QP solvers, including HPR-QP, PDHCG, PDQP, SCS (GPU), OSQP, and COPT. Several evaluated solvers exceed the time limit (denoted by "t", $>$ 7200s) on the largest instances, particularly those in the `avazu` and `kdd` families. In contrast, D-PDHCG successfully leverages multi-GPU parallelism to achieve the fastest solving times across all evaluated instances. Appendix L.7 gives the componentwise proximal update and describes the compared formulations. Detailed dataset descriptions, solver references, and experimental settings are provided in Appendix K.

Table 3: Solving time (s) on large-scale LASSO problems; Best results are **bold**, second-best are underlined. "t" denotes time limit exceeded ($>$ 7200s).

		D-PDHCG (Ours)								
Problem	PDHCG-II	2 GPUs	4 GPUs	8 GPUs	HPR-QP	PDHCG	PDQP	SCS (GPU)	OSQP	COPT
SLS	1.96	1.10	0.56	**0.50**	2.09	3.35	7.30	345.09	80.32	88.21
rcv1_test	2.37	1.48	1.06	**0.81**	6.21	7.12	19.54	t	t	t
avazu-site.tr	1337.05	953.02	547.28	**376.90**	4911.41	1377.54	5124.82	t	t	t
avazu-app	217.70	116.92	65.86	**46.30**	753.65	1429.55	5557.97	t	t	t
avazu-site	1642.69	993.62	765.46	**503.99**	3213.38	4224.95	t	t	t	t
kddb2010_test	11.59	11.18	8.79	**7.07**	26.87	11.10	46.49	490.66	255.81	69.57
kdda2010_test	9.90	8.50	6.77	**5.61**	29.36	61.00	148.01	t	t	t
kddb2010_train	703.36	466.28	317.69	**233.44**	1971.24	387.00	1715.50	t	t	t
kdda2010_train	341.16	237.19	167.62	**124.89**	842.02	2705.94	t	t	t	t

7 Conclusion, Limitation, and Future Work

We introduced **D-PDLP**, which, to our knowledge, is the first multi-GPU PDLP solver for large-scale LPs, together with a theoretical analysis that guides its distributed execution. By combining two-dimensional grid partitioning with block-wise random permutation and nonzero-aware partitioning, D-PDLP balances irregular sparse workloads while preserving local structures for efficient SpMV. Experiments on large-scale benchmark and industrial instances demonstrate strong scalability, achieving approximately $6\times$ speedups on 8 GPUs. The cost model explains how matrix, vector, and communication costs shape this scaling and guides grid and GPU-count selection. The `sdm` and `zib03` case studies evaluate its predictive accuracy in different workload regimes.

The tightly coupled design of D-PDLP is best suited to a high-bandwidth NVLink domain, where dense AllReduce communication can be efficiently supported. Scaling beyond this domain is limited by interconnect latency and bandwidth. Larger NVLink domains offer a direction for extending this design (Appendix L.8).

Future work will refine the cost model to account for achieved bandwidth and cache behavior, and extend the D-PDLP framework to a broader class of convex optimization problems, including Second-Order Cone Programming (SOCP) and Semidefinite Programming (SDP).

AI USE STATEMENT

Generative AI tools were used for language polishing of the manuscript. All AI-assisted revisions were carefully reviewed and edited by the authors for accuracy and consistency with the intended content. We take responsibility for the final content of this work, including all text and claims produced with the assistance of generative AI.

ETHICS STATEMENT

This work studies computational methods for linear and diagonal quadratic programming. The experiments described in this paper use optimization benchmarks and derived problem instances. The proposed acceleration methods are general-purpose; the ethical implications of downstream use depend on the application, data, and constraints chosen by practitioners.

REPRODUCIBILITY STATEMENT

The source code is available in an anonymous repository at https://anonymous.4open.science/r/D-PDLP_SUBMIT-962B. The distributed updates and partitioning strategies are described in Sections 3 and 4, with pseudocode in Algorithm 1. Appendix F details the cost model and parameter estimation. Section 5 and Appendix G describe the benchmarks, stopping tolerances, metrics, and hardware. Further comparisons and ablations appear in Appendices H, I, and J.1; the QP datasets and settings are described in Appendix K.

REFERENCES

Warren P Adams and Terri A Johnson. Improved linear programming-based lower bounds for the quadratic assignment problem. In *Quadratic Assignment and Related Problems*, volume 16 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pp. 43–75. American Mathematical Society, 1994. doi: 10.1090/dimacs/016/02.

Alekh Agarwal, Alina Beygelzimer, Miroslav Dudík, John Langford, and Hanna Wallach. A reductions approach to fair classification. In *International conference on machine learning*, pp. 60–69. PMLR, 2018.

Jacob M Aguirre, Diego Cifuentes, Vincent Guigues, Renato DC Monteiro, Victor Hugo Nascimento, and Arnesh Sujanani. cuHALLaR: A GPU accelerated low-rank augmented lagrangian method for large-scale semidefinite programming. *arXiv preprint arXiv:2505.13719*, 2025.

Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Always learning. Pearson, 2014. ISBN 9781292042701. URL https://www.pearson.com/en-gb/subject-catalog/p/network-flows-pearson-new-international-edition/P200000003569/9781292042701.

David Applegate, Mateo Díaz, Oliver Hinder, Haihao Lu, Miles Lubin, Brendan O'Donoghue, and Warren Schudy. Practical large-scale linear programming using primal-dual hybrid gradient. *Advances in Neural Information Processing Systems*, 34:20243–20257, 2021.

Jacob Austin, Sholto Douglas, Roy Frostig, Anselm Levskaya, Charlie Chen, Sharad Vikram, Federico Lebron, Peter Choy, Vinay Ramasesh, Albert Webson, and Reiner Pope. How to scale your model. Google DeepMind, online, 2025. URL https://jax-ml.github.io/scaling-book/.

Aharon Ben-Tal, Dick Den Hertog, Anja De Waegenaere, Bertrand Melenberg, and Gijs Rennen. Robust solutions of optimization problems affected by uncertain probabilities. *Management Science*, 59(2):341–357, 2013.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. doi: 10.1137/141000671. URL https://epubs.siam.org/doi/10.1137/141000671.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27, 2011.

Kaihuang Chen, Defeng Sun, Yancheng Yuan, Guojun Zhang, and Xinyuan Zhao. HPR-QP: A dual Halpern Peaceman-Rachford method for solving large-scale convex composite quadratic programming. *arXiv preprint arXiv:2507.02470*, 2025.

Kaihuang Chen, Defeng Sun, Yancheng Yuan, Guojun Zhang, and Xinyuan Zhao. HPR-LP: An implementation of an HPR method for solving linear programming. *Mathematical Programming Computation*, 18(1):183–210, 2026. doi: 10.1007/s12532-025-00292-0.

Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

Qi Deng, Qing Feng, Wenzhi Gao, Dongdong Ge, Bo Jiang, Yuntian Jiang, Jingsong Liu, Tianhao Liu, Chenyu Xue, Yinyu Ye, and Chuwen Zhang. New developments of ADMM-based interior point methods for linear programming and conic programming. *arXiv preprint arXiv:2209.01793*, 2022. URL https://arxiv.org/abs/2209.01793v1.

Arnaud Deza, Santanu Dey, and Pascal Van Hentenryck. Distributed linear programming on GPU clusters at extreme scale. *arXiv preprint arXiv:2609.09108*, 2026. URL https://arxiv.org/abs/2609.09108v1.

Lijun Ding, Haihao Lu, and Jinwen Yang. New understandings and computation on augmented lagrangian methods for low-rank semidefinite programming. *arXiv preprint arXiv:2505.15775*, 2025.

Edmund Eisenberg and David Gale. Consensus of subjective probabilities: The pari-mutuel method. *The Annals of Mathematical Statistics*, 30(1):165–168, 1959.

Guillermo Gallego and Garrett Van Ryzin. A multiproduct dynamic pricing problem and its applications to network yield management. *Operations research*, 45(1):24–41, 1997.

Harsha Gangammanavar and Suvrajeet Sen. Stochastic dynamic linear programming: A sequential sampling algorithm for multistage stochastic linear programming. *SIAM Journal on Optimization*, 31(3):2111–2140, 2021.

Wenzhi Gao, Dongdong Ge, Chunlin Sun, and Yinyu Ye. Solving linear programs with fast online learning algorithms. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10649–10675. PMLR, 2023. URL https://proceedings.mlr.press/v202/gao23a.html.

Dongdong Ge, Qi Huangfu, Zizhuo Wang, Jian Wu, and Yinyu Ye. Cardinal Optimizer (COPT) user guide. *arXiv preprint arXiv:2208.14314*, 2022.

Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, et al. MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 13(3):443–490, 2021.

Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*, 2021. URL https://www.gurobi.com.

Qiushi Han, Chenxi Li, Zhenwei Lin, Caihua Chen, Qi Deng, Dongdong Ge, Huikang Liu, and Yinyu Ye. A low-rank ADMM splitting approach for semidefinite programming. *arXiv preprint arXiv:2403.09133*, 2024a.

Qiushi Han, Zhenwei Lin, Hanwen Liu, Caihua Chen, Qi Deng, Dongdong Ge, and Yinyu Ye. Accelerating low-rank factorization-based semidefinite programming algorithms on GPU. *arXiv preprint arXiv:2407.15049*, 2024b.

Norio Hibiki. Multi-period stochastic optimization models for dynamic asset allocation. *Journal of Banking & Finance*, 30(2):365–390, 2006.

Yicheng Huang, Wanyu Zhang, Hongpei Li, Dongdong Ge, Huikang Liu, and Yinyu Ye. A restarted primal-dual hybrid conjugate gradient method for large-scale quadratic programming. *INFORMS Journal on Computing*, 2025. doi: 10.1287/ijoc.2024.0983.

Thorsten Joachims, Thomas Finley, and Chun-Nam John Yu. Cutting-plane training of structural SVMs. *Machine learning*, 77(1):27–59, 2009.

Frank P Kelly, Aman K Maulloo, and David Kim Hong Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, 49(3):237–252, 1998.

Thorsten Koch, Timo Berthold, Jaap Pedersen, and Charlie Vanaret. Progress in mathematical programming solvers from 2001 to 2020. *EURO Journal on Computational Optimization*, 10: 100031, 2022. ISSN 21924406. doi: 10.1016/j.ejco.2022.100031. URL https://linkinghub.elsevier.com/retrieve/pii/S2192440622000077.

Nikita Kornilov, Petr Mokrov, Alexander Gasnikov, and Alexander Korotin. Optimal flow matching: Learning straight trajectories in just one step. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/bc8f76d9caadd48f77025b1c889d2e2d-Abstract-Conference.html.

Jure Leskovec and Andrej Krevl. SNAP datasets: Stanford large network dataset collection. Stanford University, 2014. URL https://snap.stanford.edu/data/.

Bingheng Li, Linxin Yang, Yupeng Chen, Senmiao Wang, Haitao Mao, Qian Chen, Yao Ma, Akang Wang, Tian Ding, Jiliang Tang, and Ruoyu Sun. PDHG-unrolled learning-to-optimize method for large-scale linear programming. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 29164–29180. PMLR, 2024. URL https://proceedings.mlr.press/v235/li24ce.html.

Hongpei Li, Yicheng Huang, Huikang Liu, Dongdong Ge, and Yinyu Ye. Gpu-accelerated conic quadratic programming with local linear convergence under strict complementarity. *arXiv preprint arXiv:2608.09159*, 2026a.

Hongpei Li, Yicheng Huang, Huikang Liu, Dongdong Ge, and Yinyu Ye. PDHCG-II: An enhanced version of PDHCG for large-scale convex QP, 2026b. URL https://arxiv.org/abs/2602.23967.

Hongpei Li, Huikang Liu, Dongdong Ge, and Yinyu Ye. A curvature-aware rank-adaptive distributed augmented-lagrangian solver for large-scale sdps. *arXiv preprint arXiv:2607.17933*, 2026c.

Tianyi Lin, Shiqian Ma, Yinyu Ye, and Shuzhong Zhang. An ADMM-based interior-point method for large-scale linear programming. *Optimization Methods and Software*, 36(2-3):389–424, 2021.

Zhenwei Lin, Zikai Xiong, Dongdong Ge, and Yinyu Ye. PDCS: A primal-dual large-scale conic programming solver with GPU enhancements. *arXiv preprint arXiv:2505.00311*, 2025. URL https://arxiv.org/abs/2505.00311v1.

Huikang Liu, Yicheng Huang, Hongpei Li, Dongdong Ge, and Yinyu Ye. PDHCG: A scalable first-order method for large-scale competitive market equilibrium computation. *arXiv preprint arXiv:2506.06258*, 2025.

Haihao Lu and Jinwen Yang. cuPDLP.jl: A GPU implementation of restarted primal-dual hybrid gradient for linear programming in Julia. *arXiv preprint arXiv:2311.12180*, 2023a.

Haihao Lu and Jinwen Yang. A practical and optimal first-order method for large-scale convex quadratic programming. *arXiv preprint arXiv:2311.07710*, 2023b.

Haihao Lu and Jinwen Yang. PDOT: A practical primal-dual algorithm and a gpu-based solver for optimal transport. *arXiv preprint arXiv:2407.19689*, 2024a.

Haihao Lu and Jinwen Yang. Restarted Halpern PDHG for linear programming. *arXiv preprint arXiv:2407.16144*, 2024b.

Haihao Lu, Jinwen Yang, Haodong Hu, Qi Huangfu, Jinsong Liu, Tianhao Liu, Yinyu Ye, Chuwen Zhang, and Dongdong Ge. cuPDLP-C: A strengthened implementation of cuPDLP for linear programming by c language. *arXiv preprint arXiv:2312.14832*, 2023.

Haihao Lu, Zedong Peng, and Jinwen Yang. MPAX: Mathematical programming in JAX. *arXiv preprint arXiv:2412.09734*, 2024. URL https://arxiv.org/abs/2412.09734v3. Version 3, revised January 2026.

Haihao Lu, Zedong Peng, and Jinwen Yang. cuPDLPx: A further enhanced gpu-based first-order solver for linear programming. *arXiv preprint arXiv:2507.14051*, 2025.

Alan Malek, Yasin Abbasi-Yadkori, and Peter Bartlett. Linear programming for large-scale Markov decision problems. In *Proceedings of the 31st International Conference on Machine Learning*, volume 32(2) of *Proceedings of Machine Learning Research*, pp. 496–504. PMLR, 2014. URL https://proceedings.mlr.press/v32/malek14.html.

Hans D. Mittelmann. Benchmarks for optimization software, 2020. URL https://plato.asu.edu/bench.html. Accessed 2026-09-24.

Hans D Mittelmann and P Spellucci. Decision tree for optimization software, 2005. URL https://plato.asu.edu/guide.html.

MOSEK ApS. *MOSEK Optimization Toolbox for MATLAB*, 2019. URL https://www.mosek.com/documentation/.

Yu Nesterov. Subgradient methods for huge-scale optimization problems. *Mathematical Programming*, 146(1–2):275–297, 2014. doi: 10.1007/s10107-013-0686-4.

NVIDIA. GB200 NVL72. Product documentation, 2026. URL https://www.nvidia.com/en-us/data-center/gb200-nvl72/. Accessed 2026-09-24.

Brendan O'Donoghue, Eric Chu, Neal Parikh, and Stephen Boyd. SCS: Splitting conic solver, version 3.2.4. https://github.com/cvxgrp/scs, November 2023.

Andrew T. Ogielski and William Aiello. Sparse matrix computations on parallel processor arrays. *SIAM Journal on Scientific Computing*, 14(3):519–530, 1993. doi: 10.1137/0914033.

James B Orlin. Improved algorithms for computing Fisher's market clearing prices. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pp. 291–300, 2010. doi: 10.1145/1806689.1806731.

Aida Rahmattalabi, Gregory Dexter, Sanjana Garg, Qinquan Song, Shenyinying Tu, Yuan Gao, Zhipeng Wang, and Rahul Mazumder. Large-scale regularized matching on GPU clusters. *arXiv preprint arXiv:2606.07777*, 2026. URL https://arxiv.org/abs/2606.07777v1.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

Bartolomeo Stellato, Goran Banjac, Paul Goulart, Alberto Bemporad, and Stephen Boyd. OSQP: An operator splitting solver for quadratic programs. *Mathematical Programming Computation*, 12(4):637–672, 2020.

Kalyan T Talluri and Garrett J Van Ryzin. *The theory and practice of revenue management*, volume 68. Springer Science & Business Media, 2006.

Ben Taskar, Vassil Chatalbashev, Daphne Koller, and Carlos Guestrin. Learning structured prediction models: A large margin approach. In *Proceedings of the 22nd international conference on Machine learning*, pp. 896–903, 2005.

Richard W. Vuduc and Hyun-Jin Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *High Performance Computing and Communications*, volume 3726 of *Lecture Notes in Computer Science*, pp. 807–816. Springer, 2005. doi: 10.1007/11557654_91.

Fangzhao Zhang and Stephen Boyd. Solving large multicommodity network flow problems on GPUs. *arXiv preprint arXiv:2501.17996*, 2025.

Guojun Zhang, Zhexuan Gu, Yancheng Yuan, and Defeng Sun. HOT: An efficient Halpern accelerating algorithm for optimal transport problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 47(8):6703–6714, 2025. doi: 10.1109/TPAMI.2025.3564353.

A Related Work

Traditional linear programming solvers, including the simplex method and interior-point methods (IPMs), are highly effective for small- to medium-scale problems. These methods deliver reliable, high-accuracy solutions across various applications and are supported by commercial solvers such as MOSEK (MOSEK ApS, 2019), GUROBI (Gurobi Optimization, LLC, 2021), and COPT (Ge et al., 2022). However, their applicability to large-scale instances is limited: the simplex method suffers from exponential worst-case complexity, while IPMs rely on sequential matrix factorizations that are difficult to scale and parallelize. As a result, first-order methods (FOMs) have emerged as a compelling alternative for solving large-scale LPs.

Among FOM-based approaches, ADMM-based solvers such as ABIP (Lin et al., 2021) and ABIP+ (Deng et al., 2022) demonstrate the potential of first-order methods for large-scale LPs by requiring only a one-time matrix factorization. Building on this progress, PDHG-based solvers, most notably PDLP (Applegate et al., 2021) and its GPU implementation cuPDLP.jl (Lu & Yang, 2023a), eliminate the need for matrix factorization altogether and enable efficient solution of truly large-scale LPs. To further reduce runtime overhead, the C/CUDA-based implementation cuPDLP-C (Lu et al., 2023) was introduced. More recently, cuPDLPx (Lu et al., 2025) incorporates additional algorithmic heuristics and engineering optimizations to further improve convergence behavior and computational efficiency.

Beyond GPU-based first-order primal-dual methods developed specifically for LPs, a growing body of work has proposed GPU-accelerated solvers for a wide range of large-scale convex optimization problems. For quadratic programming, representative methods include rAPDHG (Lu & Yang, 2023b), PDHCG (Huang et al., 2025), and HPR-QP (Chen et al., 2025). For semidefinite programming, notable GPU-based solvers include LoRADS (Han et al., 2024a;b), ALORA (Ding et al., 2025), cuHALLaR (Aguirre et al., 2025), and CARDAL (Li et al., 2026c). For broader conic formulations, PDCS (Lin et al., 2025) addresses large-scale conic programming, while PDHCG-CQP (Li et al., 2026a) handles conic quadratic programming. GPU-enabled primal-dual methods have also been successfully applied to optimal transport (Lu & Yang, 2024a; Zhang et al., 2025), market equilibrium computation (Liu et al., 2025), and large-scale network flow problems (Zhang & Boyd, 2025). Collectively, these GPU-accelerated primal-dual solvers demonstrate substantial advantages over traditional approaches, particularly in terms of scalability and computational efficiency on modern parallel hardware.

For multi-GPU LP solving, MPAX (Lu et al., 2024) supports distributed execution through JAX (Bradbury et al., 2018) sharding, with experiments on dense LPs. Subsequent work has adopted and evaluated D-PDLP: SHARDLP (Deza et al., 2026) adopts D-PDLP's distributed PDHG design, including its two-dimensional partitioning, primal-dual variable replication, and communication pattern, and extends it to multi-node execution. Meanwhile, Rahmattalabi et al. (2026) evaluate D-PDLP on large-scale matching LPs.

B Algorithm of Distributed PDLP

We summarize the full initialization process and the distributed iteration logic in Algorithm 1.

The Halpern iteration scheme computes a linear combination of the current PDHG output, the previous iterate z^k, and the anchor point z^0. Since z^0 follows the identical partitioning scheme as the current iterate, this update is executed locally on each device without communication. Let $z^k_{[i,j]} = (x^k_{[j]}, y^k_{[i]})$ denote the local portion of the primal-dual pair on device (i, j). The update is:

$$z^{k+1}_{[i,j]} = \frac{k+1}{k+2}\left((1+\gamma)\,\mathrm{PDHG}(z^k_{[i,j]}) - \gamma z^k_{[i,j]}\right) + \frac{1}{k+2} z^0_{[i,j]} \tag{9}$$

C Algorithm Enhancement and Evaluation in cuPDLPx

The convergence of PDHG is analyzed using the canonical norm $\|\cdot\|_P$, where the metric matrix P is defined as:

$$P := P_{\eta,\omega} = \begin{bmatrix} \frac{\omega}{\eta} I & A^\top \\ A & \frac{1}{\eta\omega} I \end{bmatrix} \tag{10}$$

The iterate update is denoted as $z^{k+1} = \mathrm{PDHG}(z^k)$, where $z^k = [x^k; y^k]^\top$.

Algorithm 1 D-PDLP

Input: Problem instance (A, b, c), device count $N_{\text{device}} \geq 2$, permutation Π, partition $\mathcal{P}$, initial point (x^0, y^0), step sizes τ, σ, and accuracy ϵ.
Initialization: Determine grid topology $|R| \times |C| = N_{\text{device}}$. Apply permutation Π and partition $\mathcal{P}$ to get

$$A \to \left\{A_{[i,j]}\right\},\ b \to \{b_{[i]}\},\ c \to \{c_{[j]}\},\ y^0 \to \{y^0_{[i]}\},\ x^0 \to \{x^0_{[j]}\}.$$

Store local blocks $(A_{[i,j]}, b_{[i]}, c_{[j]}, x^0_{[j]}, y^0_{[i]})$ in each (i,j)-th GPU.
for $k = 0, 1, \cdots$, on each (i,j)-th GPU **do**
1. Distributed Primal Update
Compute local partial gradient: $g_{[i,j]} = A^\top_{[i,j]} y^k_{[i]}$
Communication: Aggregate $g_{[i,j]}$ across row axis R: $[A^\top y^k]_{[j]} = \text{AllReduce}_R\left(g_{[i,j]}\right)$
Update local primal variable: $x^{k+1}_{[j]} = \text{proj}_{\mathcal{X}_{[j]}}\left(x^k_{[j]} - \tau(c_{[j]} - [A^\top y^k]_{[j]})\right)$
2. Distributed Dual Update
Compute local partial constraint: $v_{[i,j]} = A_{[i,j]}(2x^{k+1}_{[j]} - x^k_{[j]})$
Communication: Aggregate $v_{[i,j]}$ across column axis C: $z_{[i]} = \text{AllReduce}_C\left(v_{[i,j]}\right)$
Update local dual variable: $y^{k+1}_{[i]} = y^k_{[i]} - \sigma z_{[i]} - \sigma \cdot \text{proj}_{-S_{[i]}}\left(\sigma^{-1} y^k_{[i]} - z_{[i]}\right)$
3. Distributed Halpern Iteration Update
4. Termination criterion: Stop if the KKT error is less than ϵ.
end for

C.1 Reflected Halpern Iteration Scheme

Lu & Yang (2024b) develop the restarted Halpern PDHG (rHPDHG) scheme. This scheme interpolates between the standard PDHG iterate and an initial anchor point z^0:

$$z^{k+1} = \frac{k+1}{k+2}\left((1+\gamma)\,\text{PDHG}(z^k) - \gamma z^k\right) + \frac{1}{k+2} z^0 \tag{11}$$

Here $\gamma \in [0,1]$, and k counts iterations within the current restart epoch; a restart resets the anchor and this counter. We use the reflected form from Lu et al. (2025).

C.2 Adaptive Restarting Strategy

The adaptive restart strategy is designed to ensure the anchor point remains relevant to the local geometry. A restart is triggered based on a fixed-point error metric $r(z) = \|z - \text{PDHG}(z)\|_P$. The criteria for triggering a restart include:

- **Sufficient Decay:** $r(z^{n,k}) \leq \beta_{\text{sufficient}} r(z^{n,0})$.
- **Necessary Decay:** $r(z^{n,k}) \leq \beta_{\text{necessary}} r(z^{n,0})$ and $r(z^{n,k}) > r(z^{n,k-1})$.
- **Artificial Restart:** $k \geq \beta_{\text{artificial}} T$, where T is the total iterations.

C.3 PID-Controlled Primal Weight Update

To balance progress between the primal and dual spaces, the primal weight ω is dynamically adjusted using a Proportional-Integral-Derivative (PID) controller at each restart Lu & Yang (2024b). Logarithmic error e^n is defined as the distance between the primal and dual to the anchor:

$$e^n = \log\left(\frac{\sqrt{\omega^n}\|x^{n,t} - x^{n,0}\|_2}{\frac{1}{\sqrt{\omega^n}}\|y^{n,t} - y^{n,0}\|_2}\right) \tag{12}$$

The weight for the next epoch is updated as:

$$\log \omega^{n+1} = \log \omega^n - \left[K_P \cdot e^n + K_I \sum_{i=1}^{n} e^i + K_D(e^n - e^{n-1})\right] \tag{13}$$

where K_P, K_I, and K_D are the controller coefficients.

C.4 KKT Evaluation

The dual form of the problem is given by

$$\max_{y\in\mathcal{Y},r\in\mathcal{R}} -p(-y;\ell_c,u_c) - p(-r;\ell_v,u_v) \quad \text{subject to } c - A^\top y = r, \tag{14}$$

where $\mathcal{R} \subseteq \mathbb{R}^n$ is the Cartesian product whose i-th components are determined by the boundedness of the primal constraints:

$$\mathcal{R}_i := \begin{cases} \{0\} & \text{if } (\ell_v)_i = -\infty, (u_v)_i = \infty, \\ \mathbb{R}^- & \text{if } (\ell_v)_i = -\infty, (u_v)_i \in \mathbb{R}, \\ \mathbb{R}^+ & \text{if } (\ell_v)_i \in \mathbb{R}, (u_v)_i = \infty, \\ \mathbb{R} & \text{otherwise.} \end{cases} \tag{15}$$

The solver terminates when the iterates satisfy a specified tolerance ϵ across three relative KKT error metrics: primal residual (r_{primal}), dual residual (r_{dual}), and the duality gap (r_{gap}).

$$r_{\text{primal}} = \frac{\|Ax - \text{proj}_{\mathcal{S}}(Ax)\|_2}{1 + \|[\ell_c, u_c]\|_2},$$

$$r_{\text{dual}} = \frac{\left\|\text{proj}_{\mathcal{X}}(x - \tau(c - A^\top y)) - x\right\|_2 / \tau}{1 + \|c\|_2},$$

$$r_{\text{gap}} = \frac{|c^\top x + p(-y;\ell_c,u_c) - s^\top x|}{1 + \max\{|c^\top x|, |-p(-y;\ell_c,u_c) + s^\top x|\}}.$$

where $s = (\text{proj}_{\mathcal{X}}(x - \tau(c - A^T y)) - (x - \tau(c - A^T y)))/\tau$. The overall residual is defined as $\max\{r_{\text{primal}}, r_{\text{dual}}, r_{\text{gap}}\}$.

D Differences from Dense Model-Parallel Systems

Dense model-parallel training systems, exemplified by Megatron-LM (Shoeybi et al., 2020), partition large weight tensors across devices to scale deep-learning workloads. D-PDLP uses a two-dimensional device grid for sparse matrix–vector operations with irregular workloads.

Dense vs. Sparse Workloads. In dense model parallelism, each local block has a perfectly regular shape (e.g., $m/|R| \times n/|C|$) and the workload is statically predictable. In contrast, LP constraint matrices are highly sparse and structurally irregular. A naive row–column split can produce orders-of-magnitude imbalance in nonzero counts across devices, which directly translates to stragglers and under-utilized GPUs. Our block-wise random permutation together with nonzero-aware partitioning, illustrated in Section 4 is introduced precisely to address this irregularity.

Solver-Specific Design for PDLP. D-PDLP aligns its data layout and communication with the PDLP iteration. In each iteration, the two SpMV operations ($A\bar{x}$ and $A^\top y$) can be performed locally on each device once the corresponding primal and dual slices are available, while all element-wise operations (primal/dual updates, projections, and Halpern steps) require no inter-device communication. To support this execution pattern, the 2D layout is co-designed with the algorithm: (i) primal slices $x_{[j]}$ are replicated across row communicators so that each device in column j can compute its local $A_{[i,j]}\bar{x}_{[j]}$ independently; (ii) dual slices $y_{[i]}$ are replicated across column communicators so that each device in row i can compute its local $A^\top_{[i,j]}y_{[i]}$ independently; and (iii) two orthogonal AllReduce phases synchronize the partial results—one along columns for $A\bar{x}$ and one along rows for $A^\top y$. Consequently, the data layout, variable replication pattern, and communication schedule are all dictated by the PDLP algorithmic structure.

E Distributed Infrequent Operations and Communication Analysis

The core loop performs primal, dual, and Halpern updates at each iteration. KKT evaluations and restart checks run periodically, with primal-weight updates performed at restarts. Their communication

costs are amortized over the intervening core iterations. Table 4 summarizes the communication patterns; the following subsections describe their distributed implementation.

Table 4: Communication Analysis

Operation	Comm. Type	Axis	Data Size	Frequency
Main Loop				
Primal Step ($A^\top y$)	Vector Sum	R	$\mathcal{O}(n/\lvert C\rvert)$	1 per iter
Dual Step (Ax)	Vector Sum	C	$\mathcal{O}(m/\lvert R\rvert)$	1 per iter
Halpern Update	None	-	0	1 per iter
KKT Evaluation				
Matrix Products	Vector Sum	R, C	$\mathcal{O}(n/\lvert C\rvert + m/\lvert R\rvert)$	$1/K$ iters
Residual Norms	Scalar Sum	G	$\mathcal{O}(1)$	$1/K$ iters
Gap Calculation	Scalar Sum	G	$\mathcal{O}(1)$	$1/K$ iters
Restart Logic				
PDHG map for restart	Vector Sum	R, C	$\mathcal{O}(n/\lvert C\rvert + m/\lvert R\rvert)$	$1/K$ iters
Restart metric	Scalar Sum	G	$\mathcal{O}(1)$	$1/K$ iters
Apply Restart	None	-	0	-
Weight Update	Scalar Sum	G	$\mathcal{O}(1)$	On Restart

E.1 Distributed KKT Error Computation

We assemble the KKT residuals from Appendix C.4 at a primal-dual point (x, y) using distributed products and reductions. First compute

$$\begin{aligned} a_{[i]} &= \mathrm{AllReduce}_C(A_{[i,j]}x_{[j]}), \\ g_{[j]} &= c_{[j]} - \mathrm{AllReduce}_R(A_{[i,j]}^\top y_{[i]}), \\ \widehat{x}_{[j]} &= \mathrm{proj}_{\mathcal{X}_{[j]}}(x_{[j]} - \tau g_{[j]}). \end{aligned}$$

The local primal and dual residual blocks and the bound-slack blocks are

$$\begin{aligned} \rho_{\mathrm{p},[i]} &= a_{[i]} - \mathrm{proj}_{\mathcal{S}_{[i]}}(a_{[i]}), \\ \rho_{\mathrm{d},[j]} &= (\widehat{x}_{[j]} - x_{[j]})/\tau, \\ s_{[j]} &= \big(\widehat{x}_{[j]} - (x_{[j]} - \tau g_{[j]})\big)/\tau. \end{aligned}$$

In particular, $g_{[j]} - s_{[j]} = -\rho_{\mathrm{d},[j]}$, so the projected dual residual includes the contribution of variable bounds. The relative residuals are

$$\begin{aligned} r_{\mathrm{primal}} &= \frac{\sqrt{\mathrm{AllReduce}_R(\|\rho_{\mathrm{p},[i]}\|_2^2)}}{1 + \|[\ell_c, u_c]\|_2}, \\ r_{\mathrm{dual}} &= \frac{\sqrt{\mathrm{AllReduce}_C(\|\rho_{\mathrm{d},[j]}\|_2^2)}}{1 + \|c\|_2}. \end{aligned}$$

The normalization constants are the same global data norms as in the centralized definitions and can be assembled once from their local squared-norm contributions. Primal residual blocks are summed only over R, and dual residual blocks only over C: each distinct block is counted once.

The primal and dual objective terms and the relative gap are assembled as

$$\begin{aligned} \mathrm{Obj}_P &= \mathrm{AllReduce}_C(c_{[j]}^\top x_{[j]}), \\ \mathrm{Obj}_D &= -\mathrm{AllReduce}_R\big(p(-y_{[i]}; (\ell_c)_{[i]}, (u_c)_{[i]})\big) + \mathrm{AllReduce}_C(s_{[j]}^\top x_{[j]}), \\ r_{\mathrm{gap}} &= \frac{|\mathrm{Obj}_P - \mathrm{Obj}_D|}{1 + \max\{|\mathrm{Obj}_P|, |\mathrm{Obj}_D|\}}. \end{aligned}$$

The stopping metric is $\max\{r_{\mathrm{primal}}, r_{\mathrm{dual}}, r_{\mathrm{gap}}\}$.

E.2 Distributed Operators for Algorithm Enhancements

The advanced heuristics—namely Adaptive Restarting and PID-controlled weight updates—require specific distributed operations to function correctly on the $R \times C$ grid. We define these operations using the partition notation where the primal vector x is partitioned along the column axis C into blocks $x_{[j]}$, and the dual vector y is partitioned along the row axis R into blocks $y_{[i]}$.

E.2.1 Distributed Fixed-Point Error

The restarting strategy relies on the fixed-point error metric $r(z) = \|z - \text{PDHG}(z)\|_P$. The computation of the squared norm $\|\Delta z\|_P^2$ (where $\Delta z = z - \text{PDHG}(z)$) involves a coupling term due to the off-diagonal blocks of the metric matrix P:

$$\|\Delta z\|_P^2 = \frac{\omega}{\eta}\|\Delta x\|^2 + \frac{1}{\eta\omega}\|\Delta y\|^2 + 2\langle A\Delta x, \Delta y\rangle \tag{16}$$

To compute this efficiently without reconstructing global vectors, we decompose the calculation into distributed scalar reductions:

Primal and Dual Norms: The squared Euclidean norms are computed by summing the local squared norms across the respective partition axes and aggregating globally. The factors $1/|R|$ and $1/|C|$ account for the replication of variables across the orthogonal grid axes:

$$\|\Delta x\|^2 = \text{AllReduce}_G\left(\frac{1}{|R|}\|\Delta x_{[j]}\|^2\right) \tag{17}$$

$$\|\Delta y\|^2 = \text{AllReduce}_G\left(\frac{1}{|C|}\|\Delta y_{[i]}\|^2\right) \tag{18}$$

Interaction Term ($\langle A\Delta x, \Delta y\rangle$): Directly computing $A\Delta x$ would require an expensive vector `AllReduce`. Instead, we utilize the linearity of the inner product. Each device computes the local dot product between its partial matrix-vector result ($A_{[i,j]}\Delta x_{[j]}$) and its local dual block ($\Delta y_{[i]}$). We then aggregate these scalars globally:

$$\langle A\Delta x, \Delta y\rangle = \text{AllReduce}_G\left((A_{[i,j]}\Delta x_{[j]})^\top \Delta y_{[i]}\right) \tag{19}$$

This approach reduces the communication volume from $\mathcal{O}(m+n)$ (vector reduction) to $\mathcal{O}(1)$ (scalar reduction). The entire metric can be aggregated in one scalar collective by first combining the local terms:

$$\|\Delta z\|_P^2 = \text{AllReduce}_G\left(\frac{\omega}{\eta|R|}\|\Delta x_{[j]}\|^2 + \frac{1}{\eta\omega|C|}\|\Delta y_{[i]}\|^2 + 2(A_{[i,j]}\Delta x_{[j]})^\top \Delta y_{[i]}\right).$$

This scalar aggregation is separate from any vector collectives needed to evaluate the PDHG map that defines Δz.

E.2.2 Distributed PID Weight Update

The PID controller adjusts the primal weight ω based on the logarithmic error e^n, which depends on the global distances to the anchor point:

$$d_x^2 = \|x^{n,t} - x^{n,0}\|_2^2, \quad d_y^2 = \|y^{n,t} - y^{n,0}\|_2^2 \tag{20}$$

These distances are computed using global scalar reductions analogous to the norm calculations above. Once the scalars d_x and d_y are synchronized via AllReduce_G, the PID update for ω^{n+1} is performed redundantly on every device. This guarantees that the scalar parameter ω remains consistent across the entire grid without requiring a separate broadcast step.

F Detailed Derivations of the Theoretical Cost Model

We derive the components of the per-iteration cost model from Section 3.3:

$$
\begin{aligned}
T_{\text{step}}(|R|,|C|) = & \underbrace{\frac{\mathcal{C}_{\text{spmv}} \cdot \text{nnz}}{B_{\text{mem}} \cdot N_{\text{device}}}}_{\text{SpMV I/O}} + \underbrace{\frac{\mathcal{C}_{\text{vec}}}{B_{\text{mem}}}\left(\frac{n}{|C|} + \frac{m}{|R|}\right)}_{\text{Vector I/O}} \\
& + \underbrace{\frac{\mathcal{C}_{\text{net}}}{B_{\text{net}}}\left(\frac{n|R| + m|C| - (m+n)}{N_{\text{device}}}\right)}_{\text{Network Payload}} \\
& + \underbrace{n_{\text{hop}}\beta_{\text{sync}}\big(\mathbb{I}(|R|>1) + \mathbb{I}(|C|>1)\big)}_{\text{Topology Latency}}
\end{aligned} \tag{21}
$$

The model approximates execution time by dividing data-traffic estimates by bandwidths. It assumes balanced, sufficiently large sparse workloads. Each iteration streams both $A\bar{x}$ and $A^\top y$. With FP64 values and INT32 indices, these two passes read $2(8+4) = 24$ bytes per logical nonzero and perform approximately four floating-point operations. The resulting arithmetic intensity is at most $4/24 \approx 0.167$ FLOP/byte, even before vector gathers, pointers, output writes, and cache-line effects. Values, indices, and vector entries move from HBM into on-chip caches/registers, so data movement can dominate arithmetic. Small shards, cache behavior, and kernel launches can depart from this approximation.

F.1 Hardware and Algorithmic Parameters

To formalize the derivation, we first define the system and algorithmic parameters utilized in the cost model:

- B_{mem}: Single-GPU High Bandwidth Memory (HBM) bandwidth used in the model.
- B_{net}: Interconnect bandwidth used in the model (e.g., NVLink or InfiniBand).
- $\mathcal{C}_{\text{spmv}}$: Matrix values and indices streamed per logical nonzero across both SpMV operations (24 bytes for FP64/INT32).
- $\mathcal{C}_{\text{vec}}$: Effective coefficient for dimension-dependent pointer, vector-gather, and element-wise update traffic.
- $\mathcal{C}_{\text{net}}$: Network transfer byte coefficient per payload element.
- β_{sync}: Base latency per active collective, aggregating protocol and synchronization overhead.
- n_{hop}: Effective topology multiplier applied to the base collective latency.

Model assumptions. The model uses average shard traffic to estimate relative scaling and guide resource selection. The vector coefficient aggregates dimension-dependent traffic; actual gathers depend on sparsity and cache reuse. Kernel launches and communication overlap contribute additional variation in measured execution time.

F.2 Derivation of the Cost Components

The total execution time T_{step} is additively decomposed into local computation (memory I/O) and global communication (network payload and topology latency).

Computational Memory Traffic. The local computation phase is dominated by Sparse Matrix-Vector Multiplication (SpMV) for the forward ($A\bar{x}$) and transposed ($A^\top y$) mappings, as well as element-wise vector operations for the primal, dual, and Halpern updates. We separate the local memory traffic into $O(\text{nnz})$ and $O(m,n)$ components. Under the balanced-workload approximation, each device stores $\text{nnz}/N_{\text{device}}$ non-zeros. By maintaining both CSR and CSC representations in device memory to improve efficiency, the pure matrix-dependent traffic (streaming values and indices) scales with the non-zero count. Consequently, we define the coefficient $\mathcal{C}_{\text{spmv}}$ to exclusively account for this dense array data, yielding an SpMV memory I/O latency of $\frac{\mathcal{C}_{\text{spmv}} \cdot \text{nnz}}{B_{\text{mem}} N_{\text{device}}}$, which scales as N_{device}^{-1}.

Dimension-dependent memory traffic is aggregated into the vector coefficient $\mathcal{C}_{\text{vec}}$. This aggregated term encompasses not only the element-wise additions and projections on the partitioned variables, but also the SpMV-induced dimensional I/O, specifically fetching row/column pointers and streaming the local dense vectors. In the 2D grid topology, these operations act on evenly partitioned chunks of size $n/|C|$ for the primal variables $x \in \mathbb{R}^n$ and $m/|R|$ for the dual variables $y \in \mathbb{R}^m$. The corresponding local memory-traffic term is $\frac{\mathcal{C}_{\text{vec}}}{B_{\text{mem}}}\left(\frac{n}{|C|}+\frac{m}{|R|}\right)$.

Inter-Device Network Payload. A ring `AllReduce` has a reduce-scatter phase and an all-gather phase. For a payload of L elements on K devices, each phase sends $L(K-1)/K$ elements per device. Thus the full collective sends $2L(K-1)/K$ elements per device, and receives the same amount. With concurrent sending and receiving, we model transfer time using sent bytes divided by unidirectional bandwidth.

The primal collective has $L = n/|C|$, $K = |R|$; the dual collective has $L = m/|R|$, $K = |C|$. Including both phases and 8 bytes per FP64 element gives

$$V_{\text{row}}^{\text{bytes}} = 16\,\frac{n(|R|-1)}{N_{\text{device}}}, \qquad V_{\text{col}}^{\text{bytes}} = 16\,\frac{m(|C|-1)}{N_{\text{device}}}, \tag{22}$$

$$T_{\text{net}} = \frac{16[n(|R|-1)+m(|C|-1)]}{N_{\text{device}}B_{\text{net}}}. \tag{23}$$

Accordingly, $\mathcal{C}_{\text{net}} = 16$ bytes in Eqs. (7) and (21).

Collective Latency. For the measured single-NVLink-domain setting, we use $n_{\text{hop}} = 1$ as a model parameter and a fixed $10\,\mu$s per active collective. Thus

$$T_{\text{lat}} = \lambda_R(|R|) + \lambda_C(|C|), \qquad \lambda_X(K) = 10\,\mu\text{s}\,\mathbb{I}(K>1).$$

A single device has no collective, a one-dimensional grid has one, and a grid partitioning both axes has two. Latency and bandwidth parameters depend on the interconnect topology.

Grid Shape and Scaling Regimes. For fixed $N = N_{\text{device}} = |R||C|$, both the vector term and the nonconstant part of the network term are proportional to $n/|C| + m/|R|$. A continuous relaxation over positive grid dimensions gives the ideal shape and traffic bound:

$$\min_{|R|,|C|>0,\ |R||C|=N}\left(\frac{n}{|C|}+\frac{m}{|R|}\right) = 2\sqrt{\frac{mn}{N}}, \qquad \frac{|R|}{|C|} = \frac{m}{n}.$$

This ideal ratio guides the choice of a feasible integer grid with a similar aspect ratio, as described in Section 4.1. Under the balanced-workload approximation, the matrix term decreases as N^{-1}, while the ideal aspect-ratio-matched shape gives an $N^{-1/2}$ trend for the vector term and the leading network-payload term. Collective latency contributes a separate floor per active collective. These regimes explain why additional devices help high-workload problems more than short solves.

F.3 Parameter Estimation

To operationalize the cost model for performance prediction and the AUTO scheduling heuristic, we instantiate the hardware and algorithmic parameters based on the physical specifications of our target architecture (e.g., NVIDIA H100) and the double-precision (FP64) requirements of strict linear programming solvers.

Hardware Parameters. We approximate bandwidths using nominal hardware peaks and assign a fixed collective-latency parameter. The parameters are:

- **Memory Bandwidth (B_{mem}):** We adopt the theoretical peak High Bandwidth Memory (HBM3) throughput of the NVIDIA H100 SXM5 GPU, evaluated at $B_{\text{mem}} = 3350$ GB/s.
- **Interconnect Bandwidth (B_{net}):** Communication is routed over NVLink. We set $B_{\text{net}} = 450$ GB/s, representing the theoretical peak unidirectional bandwidth of the NVLink4 interconnect.

- **Synchronization Latency (β_{sync}):** The base latency per active collective is set to $\beta_{\text{sync}} = 0.01$ ms (10 μs).
- **Topology Multiplier (n_{hop}):** For the measured eight-GPU NVLink domain, we set the effective topology multiplier to $n_{\text{hop}} = 1$.

Algorithmic Parameters. The matrix and network coefficients count bytes; the vector coefficient approximates aggregate traffic:

- **Matrix Traffic Coefficient ($\mathcal{C}_{\text{spmv}}$):** To eliminate atomic write conflicts during the transposed SpMV step, our implementation maintains both CSR and CSC formats in device memory. Each non-zero element comprises an FP64 value (8 bytes) and an INT32 index (4 bytes). A single iteration executes two SpMV passes ($A\bar{x}$ and $A^\top y$), fetching exactly 12 bytes from the CSR array and 12 bytes from the CSC array. Therefore, the pure matrix data payload evaluates exactly to $\mathcal{C}_{\text{spmv}} = 24$ bytes.
- **Aggregated Vector Traffic ($\mathcal{C}_{\text{vec}}$):** This coefficient aggregates all memory operations proportional to the matrix dimensions. It accounts for: (1) fetching the 32-bit row and column pointers during SpMV; (2) streaming the randomized primal and dual vectors (x and y) from HBM during SpMV; and (3) loading and storing intermediate FP64 variables during the multi-step primal, dual, and Halpern updates. We use the fixed aggregate coefficient $\mathcal{C}_{\text{vec}} = 90$ bytes for all predictions.
- **Network Payload Coefficient ($\mathcal{C}_{\text{net}}$):** The collective `AllReduce` operations synchronize the aggregated partial gradients and products. Each FP64 element occupies 8 bytes and a ring collective has two phases, so $\mathcal{C}_{\text{net}} = 2 \times 8 = 16$ bytes.

F.4 Examples

We use $B_{\text{mem}} = 3350$ GB/s, $B_{\text{net}} = 450$ GB/s, $\mathcal{C}_{\text{spmv}} = 24$ B, $\mathcal{C}_{\text{vec}} = 90$ B, $\mathcal{C}_{\text{net}} = 16$ B, and $10\,\mu$s per active collective. These fixed parameters and the raw matrix dimensions are available before launching a solve. The grids are 1×1, 2×2, and 2×4 for 1, 4, and 8 GPUs, respectively.

Instance `zib03`. Here $m = 19{,}731{,}970$, $n = 29{,}128{,}799$, and $\text{nnz} = 104{,}422{,}573$. For example, the four-GPU network term is

$$\frac{16(29{,}128{,}799 + 19{,}731{,}970)}{4 \times 450 \times 10^9} = 0.434 \text{ ms}.$$

Table 5 gives the full decomposition. Vector traffic accounts for 63.7% of the predicted single-GPU time, so the model predicts less scaling than on the denser `sdm` instance.

Instance `sdm_50k_500k_15_10`. The raw dimensions are $m = 5{,}500{,}135$, $n = 10{,}000{,}000$, and $\text{nnz} = 690{,}000{,}000$. The eight-GPU terms are

$$\begin{aligned}
T_{\text{mat}} &= \frac{24 \times 690{,}000{,}000}{8 \times 3350 \times 10^9} = 0.618 \text{ ms}, \\
T_{\text{vec}} &= \frac{90(10{,}000{,}000/4 + 5{,}500{,}135/2)}{3350 \times 10^9} = 0.141 \text{ ms}, \\
T_{\text{net}} &= \frac{16[10{,}000{,}000 + 3(5{,}500{,}135)]}{8 \times 450 \times 10^9} = 0.118 \text{ ms}, \\
T_{\text{lat}} &= 0.020 \text{ ms}.
\end{aligned}$$

The predicted speedups are $3.346\times$ and $5.977\times$ on 4/8 GPUs. The strict-accuracy C experiment in Appendix L.4, with identical iteration counts across configurations, measures $3.250\times$ and $5.788\times$. Relative to the measured speedups, the prediction differs by approximately 3.0% and 3.3%.

Discussion. The model captures the stronger scaling of the matrix-traffic-dominated `sdm` case. It underestimates the measured scaling of `zib03`; the fixed vector coefficient, achieved bandwidths, and cache effects limit absolute predictive accuracy. The C experiment holds iteration counts fixed across GPU configurations, directly measuring relative per-iteration scaling.

Table 5: Cost-model components in milliseconds. Totals and speedups are calculated before rounding the displayed components.

Instance	GPUs	Matrix	Vector	Network	Latency	Total
zib03	1	0.748	1.313	0.000	0.000	2.061
	4	0.187	0.656	0.434	0.020	1.298
	8	0.094	0.461	0.393	0.020	0.967
sdm	1	4.943	0.416	0.000	0.000	5.360
	4	1.236	0.208	0.138	0.020	1.602
	8	0.618	0.141	0.118	0.020	0.897

Table 6: Predicted speedups and the original LP iteration-time measurements in Table 2. These measurements are distinct from the supplementary strict-accuracy C runs.

	Measured speedup		Predicted speedup	
Instance	S_4	S_8	S_4	S_8
zib03	2.42	3.31	1.59	2.13
sdm	4.10	6.18	3.35	5.98

G Detailed Experimental Setup

Hardware Configuration. All distributed experiments were conducted on a high-performance computing node equipped with eight NVIDIA H100 GPUs, each with 80 GB of HBM3 memory. These GPUs are interconnected via NVLink to enable high-bandwidth peer-to-peer communication. The host system features a dual-socket configuration with Intel Xeon Platinum 8462Y+ processors, providing 64 physical cores (32 per socket) and 128 hardware threads in total.

Implementation Details. The LP experiments in Section 5 use Julia. Appendix I compares Julia and C/CUDA, while the block-size, strict-accuracy, and repeated-run experiments in Appendices L.3 and L.4 use C/CUDA. The diagonal-QP implementation builds on PDHCG-II (Li et al., 2026b).

The Julia implementation (Bezanson et al., 2017) uses `CUDA.jl`, `MPI.jl`, and `NCCL.jl`. The C/CUDA implementation builds on `cuPDLPx` (Lu et al., 2025); Appendix I reports the performance comparison.

H Solver-Time Comparison with External Solvers

To position D-PDLP relative to existing LP technology, we compare it against representative commercial and open-source solvers on the Mittelmann public benchmark set under a uniform stopping tolerance of 10^{-4}. Table 7 reports the number of solved instances and the shifted geometric mean (SGM10) of solve times in seconds.

Table 7: Solver-time comparison on the Mittelmann benchmark ($\epsilon = 10^{-4}$). SGM10 times are scaled relative to COPT. Higher "Solved" counts and lower SGM10 are better. Best results are **bold**, second-best are underlined.

Method	Solved	SGM10 (scaled)
COPT (simplex/barrier)	**50**	**1.00**
D-PDLP (AUTO)	<u>46</u>	<u>1.83</u>
D-PDLP (4 GPUs)	<u>46</u>	1.86
cuPDLPx (1 GPU)	<u>46</u>	2.13
ABIP+	29	27.22
ABIP	22	57.40

Discussion. The results confirm that commercial simplex/barrier methods (COPT) remain highly competitive on medium-scale benchmarks, achieving the fastest absolute times and the highest success rate. Among first-order methods, D-PDLP improves over the single-GPU cuPDLPx baseline and is substantially faster than the ADMM-based competitors ABIP and ABIP+. cuPDLPx has been established as the strongest available PDLP implementation (Lu et al., 2025). The speedup over cuPDLPx directly measures the gain from our distributed design relative to the current state-of-the-art in the PDLP family.

I Further Optimization via C Implementation

Our C implementation is developed based on the `cuPDLPx` repository[1], a state-of-the-art PDLP solver suggested in Lu et al. (2025). Table 8 presents a direct performance comparison between the Julia and C implementations across distributed settings with 4 and 8 GPUs. The results demonstrate that the C implementation generally outperforms the Julia version in terms of aggregate cumulative iteration time. This advantage is quantified by the SGM10, where the C solver reduces the aggregate metric from 101.2 to 86.1 seconds on 4 GPUs, and from 77.7 to 62.2 seconds on 8 GPUs.

The performance gain is most pronounced in the Unit Commitment instances. For example, on the `ds1` instance with 4 GPUs, the C implementation reduces the solution time from 143 seconds to 42 seconds, representing a speedup of approximately $3.4\times$. This improvement is driven not only by faster per-iteration execution times but also, in several cases, by a reduction in the total number of iterations required for convergence. Consistent improvements are also observed across the QAP and Pagerank benchmarks, where the C solver yields lower runtimes in every tested instance.

Table 8: Performance comparison: Julia vs. C Implementation on distributed settings (4 and 8 GPUs). Values represent solver-reported cumulative iteration time in seconds. Best results are marked in **bold**.

		4 GPUs		8 GPUs	
Source	Instance	Julia	C	Julia	C
Koch	zib03	336	**283**	245	**205**
Pagerank	rand-10m-nodes	5	**3**	4	**2**
	com-livejournal	2	**1**	1	**1**
	soc-livejournal1	1	**1**	1	**1**
Multicommodity Flow	mcf_2500_100_500	935	**836**	504	**435**
	mcf_5000_50_500	**2994**	4072	**1677**	2178
	mcf_5000_100_250	3159	**2378**	1732	**1245**
Design-matching	sdm_50k_500k_15_10	**92**	122	**61**	72
QAP	wil50	31	**29**	25	**22**
	lipa50a	20	**18**	17	**13**
	lipa50b	21	**19**	17	**14**
	tai50a	21	**21**	18	**15**
	tai50b	52	**43**	43	**35**
Unit Com.	ds1	143	**42**	122	**31**
	ds2	691	**397**	591	**288**
SGM10		101.2	**86.1**	77.7	**62.2**

J Ablation Study

In this section, we conduct ablation studies to isolate and evaluate the individual performance contributions of our proposed algorithmic components.

[1] https://github.com/MIT-Lu-Lab/cuPDLPx

J.1 Permutation and Partitioning Strategies

To justify the configuration choices adopted in the main text, Table 9 presents a controlled comparison across three permutation schemes (*No Permutation*, *Full Random*, *Block Random*) combined with either uniform (*No NNZ*) or nonzero-aware (*Yes NNZ*) matrix partitioning.

As the table shows, the choice of permutation strategy is crucial for performance stability. The *No Permutation* baseline is highly vulnerable to catastrophic load imbalance, as evidenced by extreme runtimes on structured instances such as `zib03` (over 7000 seconds on 8 GPUs) and the largest overall SGM10 of 220.08 seconds. Applying a *Full Random* permutation substantially alleviates this issue by redistributing nonzeros, while disrupting local block structure. Block-wise permutation preserves this structure and achieves lower aggregate solving times, as quantified below.

In contrast, the proposed *Block Random* permutation achieves a better trade-off. It effectively disperses dense regions to balance the workload while preserving local block structures that enable high-throughput computation. Overall, block-wise permutation achieves faster solving than the *Full Random* baseline (SGM10 of 79.51s versus 86.80s).

Second, nonzero-aware matrix partitioning plays a vital complementary role. In the absence of permutation, the *Yes NNZ* strategy yields a marked improvement by correcting severe imbalances in the natural ordering, reducing SGM10 from 220.08s to 194.13s. More importantly, this partitioning continues to provide measurable gains even when combined with the *Block Random* permutation, further improving the SGM10 from 79.51s to 75.85s.

The combined *Block Random* + *Yes NNZ* configuration achieves the fastest aggregate solving time among the six strategies.

Table 9: Impact of permutation and partitioning strategies on solving time (seconds). We compare three permutation schemes (No Permutation, Full Random, Block Random) combined with either uniform (No NNZ) or nonzero-aware (Yes NNZ) matrix partitioning. Best results are marked in **bold**, and second-best are underlined.

Instance	GPU	No Permutation		Full Random		Block Random	
		No NNZ	Yes NNZ	No NNZ	Yes NNZ	No NNZ	Yes NNZ
sdm_50k_500k_15_10	4	328.20	107.12	227.02	237.99	142.49	**92.33**
	8	527.91	377.41	69.06	91.09	78.19	**60.90**
zib04	4	80.80	80.95	93.07	92.54	79.70	**78.27**
	8	61.94	62.18	63.38	65.05	60.24	**58.61**
zib03	4	2926.91	2017.57	501.96	502.93	**336.53**	338.26
	8	7368.89	2012.84	300.60	291.65	**243.76**	245.29
ds1_lp	4	777.84	682.63	157.49	155.88	147.39	**142.67**
	8	293.32	690.24	129.47	127.95	124.65	**122.27**
ds2_lp	4	3742.87	3299.29	748.69	746.62	710.96	**690.54**
	8	1407.81	3325.82	621.12	617.57	594.08	**591.10**
lipa50a	4	33.48	30.95	20.84	20.60	19.95	**19.95**
	8	35.48	33.18	17.08	**16.77**	16.80	16.94
lipa50b	4	35.03	32.12	21.82	21.32	**21.06**	21.24
	8	36.26	35.29	17.50	17.52	**17.26**	17.31
tai50a	4	34.96	32.08	21.98	21.65	21.11	**20.80**
	8	36.95	34.97	**17.25**	17.36	17.86	17.77
tai50b	4	101.52	81.91	53.94	54.91	52.47	**52.42**
	8	92.80	88.73	44.29	43.60	43.72	**42.59**
SGM10		220.08	194.13	86.80	87.96	79.51	**75.85**

J.2 One-Time Preprocessing Overhead

Block-wise random permutation and nonzero-aware partitioning are performed once before iterative solving, and their costs are amortized over all subsequent iterations.

Permutation. Constructing the row and column index maps takes $O(m+n)$ work, and applying them visits each nonzero once, requiring $O(\mathrm{nnz})$ work. The remapped column indices are then sorted within each row. Thus, the total work is

$$O\left(m+n+\mathrm{nnz}+\sum_{i=1}^{m} d_i \log(1+d_i)\right),$$

where d_i is the number of nonzeros in row i. The row-wise operations are parallelized on the CPU.

Partitioning. Computing row and column nonzero counts requires one pass over the sparse matrix, and selecting boundaries scans these counts, giving $O(m+n+\mathrm{nnz})$ work per partitioning call. Constructing the subproblems for all N_{device} GPUs repeats this process, yielding an overall upper bound of $O(N_{\mathrm{device}}(m+n+\mathrm{nnz}))$.

Table 10 reports preprocessing times on four representative instances. The overhead ratio is the total permutation and partitioning time divided by the solving-time reduction from one to eight GPUs, $T_{1\mathrm{GPU}} - T_{8\mathrm{GPU}}$, using Table 2. Preprocessing accounts for only 0.14%–4.47% of this reduction.

Table 10: Preprocessing overhead relative to multi-GPU solving time gains. All times are reported in seconds.

Instance	Permutation (s)	Partition (s)	Improvement (s)	Ratio (%)
zib03	1.874	8.921	567	1.90
mcf_2500_100_500	1.944	18.271	2,439	0.83
ds2	0.266	0.741	697	0.14
tai50b	0.329	1.419	39	4.47

Across these four instances, permutation takes 0.266–1.944 s. Permutation and partitioning together account for only 0.14%–4.47% of the solving-time reduction from one to eight GPUs, including 0.83% on `mcf_2500_100_500`. These measurements support the small practical cost of the one-time preprocessing on the tested instances.

Table 11: Problem dimensions: comparison of scales between the original data and the converted QP instances.

	Original Scale			QP Scale		
Problem	m	n	**Sparsity**	**Rows**	**Columns**	**Nonzeros**
SLS	1,748,122	62,729	6.21E-05	1,873,578	1,873,578	7,117,944
rcv1_test	677,399	47,236	1.55E-03	771,871	771,871	50,422,601
avazu-site.tr	23,567,843	1,000,000	1.50E-05	25,567,767	25,567,767	381,085,127
avazu-app	40,428,967	1,000,000	1.50E-05	14,642,166	14,642,166	206,274,862
avazu-site	25,832,830	1,000,000	1.50E-05	27,832,754	27,832,754	417,324,822
kddb2010_test	748,401	1,163,024	7.74E-06	6,729,169	6,729,169	34,675,012
kdda2010_test	510,302	20,216,830	1.87E-05	4,539,640	4,539,640	27,831,585
kddb2010_train	19,264,097	1,163,024	7.97E-06	79,044,287	79,044,287	705,170,365
kdda2010_train	8,407,752	20,216,830	1.80E-06	48,841,412	48,841,412	394,888,582

K Details of the Large-scale QP Dataset

To evaluate the capability of our algorithm in solving massive quadratic programming problems, we consider a benchmark suite derived from real-world Lasso regression tasks. This section provides the detailed derivation of the conversion, lists the scales of the resulting dataset, and describes the experimental settings and compared solvers.

K.1 LASSO TO QP CONVERSION

The primal Lasso problem is defined as:

$$\min_{x} \quad \frac{1}{2}\|Ax - b\|_2^2 + \lambda\|x\|_1, \tag{24}$$

where $A \in \mathbb{R}^{m\times n}$ is the feature matrix and $b \in \mathbb{R}^m$ is the target vector. To transform this into a standard QP form suitable for our primal-dual solver, we utilize the following equivalent formulation:

$$\begin{aligned} \min_{x,y,t} \quad & \frac{1}{2}y^\top y + \lambda \mathbf{1}^\top t \\ \text{s.t.} \quad & Ax - y = b, \\ & -t \le x \le t. \end{aligned} \tag{25}$$

This formulation yields a sparse QP that maintains the underlying structure of the original Lasso problem while enabling the application of first-order optimization methods.

K.2 DATASET SCALES AND CHARACTERISTICS

The data used in our evaluation is sourced from the LIBSVM collection (Chang & Lin, 2011) and the SuiteSparse Matrix Collection (Davis & Hu, 2011). Table 11 summarizes the problem dimensions for both the original datasets and the converted QP instances. All selected problems contain more than 6,000,000 nonzero elements, including exceptionally large instances such as `kddb2010_train`, which features over 700 million nonzeros in its QP form.

As demonstrated in the "QP Scale" columns of Table 11, these resulting problems scale up to nearly 80 million rows and columns. Such extreme dimensions provide a rigorous test for the scalability and memory efficiency of our distributed implementation.

K.3 EXPERIMENTAL SETTINGS AND COMPARED SOLVERS

Compared Solvers. The solver-time comparison in Table 3 includes the following solvers: HPR-QP (Chen et al., 2025), PDHCG (Huang et al., 2025), PDQP (Lu & Yang, 2023b), SCS (GPU) (O'Donoghue et al., 2023), OSQP (Stellato et al., 2020), and COPT (Ge et al., 2022). The single-GPU baseline PDHCG-II is reported in (Li et al., 2026b).

Configuration. For each instance, we designate the last column of matrix A as the target vector b and set the regularization parameter to $\lambda = 0.01\|A^\top b\|_\infty$, consistent with the protocol established in PDHCG (Huang et al., 2025; Li et al., 2026b). The termination tolerance is fixed at $\epsilon = 10^{-6}$. All experiments are conducted on the hardware described in Appendix G: a single-node 8×H100 NVLink server.

L SUPPLEMENTARY VALIDATION

L.1 ESTABLISHED PRIMITIVES AND SOLVER-SPECIFIC CONTRIBUTIONS

Two-dimensional sparse-matrix layouts, randomized load balancing, and collective reductions are established techniques. In particular, Ogielski & Aiello (1993) study randomized packing of sparse matrices on rectangular processor arrays, while Vuduc & Moon (2005) exploit variable block structure to improve SpMV locality and storage efficiency. These works provide precedents for the sparse-linear-algebra ingredients used here.

Our two main contributions are multi-GPU acceleration of the complete PDLP solver and a theoretical analysis that guides distributed execution. The implementation combines the established ingredients around the complete PDLP iteration: a common layout for both $A\bar{x}$ and $A^\top y$, replicated primal and dual state, local projections and Halpern updates, and distributed KKT, restart, and primal-weight computations. Appendix E gives the scalar-reduction construction for the restart coupling term; Appendix F derives the two orthogonal communication payloads and their scaling. The measured contribution of load balancing is quantified in Table 9: Block Random+NNZ reduces SGM10 from 87.96 to 75.85 s relative to Full Random+NNZ (13.8%), and from 79.51 to 75.85 s relative to Block Random with uniform cuts (4.6%).

L.2 Exact Block Permutation and Solution Mapping

For each dimension $d \in \{m, n\}$, form contiguous index blocks

$$\mathcal{B}_k = \{kB, \ldots, \min((k+1)B, d) - 1\}, \qquad k = 0, \ldots, \lceil d/B \rceil - 1.$$

Independently shuffle the row-block and column-block IDs, preserving the order of indices inside every block. The last, possibly shorter, block is retained without padding. Concatenating the shuffled blocks defines row and column permutation matrices P_r and P_c. The permuted LP uses

$$\begin{aligned} \widetilde{A} &= P_r A P_c^\top, & \widetilde{c} &= P_c c, \\ \widetilde{\ell}_v &= P_c \ell_v, & \widetilde{u}_v &= P_c u_v, \\ \widetilde{\ell}_c &= P_r \ell_c, & \widetilde{u}_c &= P_r u_c. \end{aligned} \tag{26}$$

An equality right-hand side, if present, is mapped as $\widetilde{b} = P_r b$. Initial points use $\widetilde{x}^0 = P_c x^0$ and $\widetilde{y}^0 = P_r y^0$. Nonzero-aware marginal cuts are then computed on $\widetilde{A}$, and the solver returns $x = P_c^\top \widetilde{x}$ and $y = P_r^\top \widetilde{y}$. Thus the objective, constraints, starts, and returned solutions all use consistent maps.

The block size B controls permutation granularity, while the $|R| \times |C|$ grid determines device ownership after repartitioning. Figure 2 illustrates how row and column permutations redistribute the nonzeros of a toy matrix across device tiles while preserving their row/column incidence structure.

L.3 Block-Size Sensitivity

We evaluate the C implementation on four instances using eight H100 GPUs, tolerance 10^{-6}, and a common random seed. Only the host-side permutation block size B changes; the CUDA kernel configuration is fixed. Table 12 reports solver-reported iteration time, meaning the cumulative time spent in the iterative solving phase, in seconds. All 20 runs terminate with status `OPTIMAL`.

Table 12: Sensitivity to permutation block size on eight H100 GPUs. "neos320" denotes `neos-3208254-reiu`. Times and SGM10 are reported to two decimal places.

B	zib03	neos320	rmine25	ivu06-big	SGM10
4	260.29	26.17	37.91	109.69	76.53
32	217.99	24.93	38.87	108.62	72.43
128	218.78	24.65	39.50	108.62	72.60
256	218.68	24.63	39.90	110.58	73.08
1024	223.69	22.79	42.67	110.15	73.44

The results show a broad plateau over $B = 32$–1024, with SGM10 within 1.4% of the best value. Our default $B = 256$ is within 0.9% of the minimum. At $B = 4$, SGM10 is 4.7% higher than at $B = 256$, with a 19.0% increase on `zib03`. The sweep supports choosing B within the broad performance plateau.

L.4 Stopping Tolerances, Strict Accuracy, and Timing Variability

The broad LP benchmark sweep and the `mcf` instances use $\epsilon = 10^{-4}$, matching the public cuPDLPx stopping configuration used for resource-scaling comparisons. The other application LPs and all diagonal-QP comparisons use $\epsilon = 10^{-6}$. Within each comparison, GPU configurations use the same stopping thresholds. The additional LP results below test whether scaling persists at 10^{-8}.

Numerical behavior of distributed reductions. The two orthogonal `AllReduce` operations sum local matrix–vector products to recover Ax and $A^\top y$. This decomposition preserves the centralized updates in exact arithmetic. In FP64 arithmetic, changes in summation order can produce rounding differences, particularly in sums with cancellation. Tables 13 and 14 show that the tested configurations attain the target tolerances with identical iteration counts across GPU counts at each tolerance.

Timing convention. Solving time measures the cumulative solving phase. All baselines in our LP and QP comparisons use the same timing convention: input loading and one-time preprocessing are excluded. For D-PDLP and D-PDHCG, this is the cumulative iterative phase. Permutation and partitioning costs are reported separately in Appendix J.2.

Raw `sdm` at 10^{-8}. For `sdm_50k_500k_15_10`, the raw input has 5,500,135 rows, 10,000,000 columns, and 690,000,000 nonzeros. The C implementation reaches the target in exactly 340,200 iterations on each of 1, 4, and 8 GPUs (Table 13). Across these configurations, the worst relative primal residual, dual residual, and gap are 9.98×10^{-9}, 1.43×10^{-11}, and 4.16×10^{-11}, respectively. The identical iteration counts isolate the improvement in execution time from changes in iteration count.

Table 13: Strict-accuracy C runs on raw `sdm` at $\epsilon = 10^{-8}$. Times measure the cumulative iterative phase.

GPUs	Iterations	Time (s)	Speedup
1	340,200	1368.142	1.000×
4	340,200	420.967	3.250×
8	340,200	236.394	5.788×

Repeated runs across tolerances. On presolved `neos-3208254-reiu` (8.82M rows, 2.92M columns, and 23.43M nonzeros), each fixed-seed C configuration is repeated three times for every tolerance and GPU count. Table 14 gives median cumulative iteration times. Iteration counts agree across GPU counts at each tolerance. The maximum coefficient of variation (standard deviation divided by mean) of iteration time is 0.211%; the eight-GPU speedup changes by approximately 0.25% between 10^{-4} and 10^{-8}. At 10^{-8}, the worst relative primal residual, dual residual, and gap are 2.73×10^{-9}, 1.03×10^{-10}, and 3.06×10^{-9}.

Table 14: Tolerance and repeated-run results on presolved `neos-3208254-reiu`. Median cumulative iteration times over three runs are in seconds.

Tolerance	Iterations	1 GPU	4 GPUs	8 GPUs	S_4	S_8
10^{-4}	15,800	10.084	7.541	7.029	1.337	1.435
10^{-6}	55,600	35.417	26.408	24.646	1.341	1.437
10^{-8}	60,600	38.592	28.784	26.835	1.341	1.438

L.5 Workload Regimes and the AUTO Interface

Table 1 covers heterogeneous iterative workloads. MIPLIB Large contains instances with more than 10M nonzeros and a range of iteration counts. Its SGM10 improves from 90.34 to 56.81 s (1.59×) on eight GPUs, while the solved count rises from 13 to 20. AUTO attains aggregate SGM10 values of 18.38 s on Mittelmann and 7.31 s on MIPLIB by choosing resources per instance.

Table 2 includes higher-workload cases: the three `mcf` instances improve from 2943/9114/9173 s to 504/1677/1732 s, and `sdm` improves from 377 to 61 s. These measurements quantify the reduction in solving time relative to single-GPU execution. The LP table reaches 127.5M variables and 690M nonzeros; Table 11 reaches 79.0M rows/columns and 705M nonzeros for converted QPs.

AUTO evaluates the pre-launch tuple (m, n, nnz) using Eq. (7). Permutations and alternative cuts do not change this tuple, so the model makes the same resource choice for all six strategies in Table 9; it selects eight GPUs for each listed instance. Holding the device count fixed at four or eight therefore isolates the load-balancing comparison from resource selection. The four- and eight-GPU layouts exercise partitioning along both grid axes.

L.6 Published CPU-Solver Context

The cuPDLP-C study (Lu et al., 2023, Table 2) reports COPT times at target tolerance 10^{-6} of 16.5 h for `zib03`, 592 s for `ds1`, and 606 s for `ds2`; neither COPT simplex nor barrier solves its five QAP relaxations within 3600 s. For the overlapping instances, our Table 2 reports eight-GPU iteration times of 245 s, 122 s, 591 s, and 17–43 s, respectively. These published measurements provide context for sparse-factorization costs on large LPs. The studies use different hardware, implementations, and timing conventions. The separate Mittelmann comparison is given in Appendix H.

L.7 Diagonal-QP Updates and Compared Formulations

The implemented extension handles convex QPs with diagonal $Q \succeq 0$ and linear constraints. If $Q = \mathrm{diag}(q)$, the primal update in Eq. (8) is componentwise:

$$x_i^{k+1} = \mathrm{proj}_{[(\ell_v)_i,(u_v)_i]}\left(\frac{x_i^k - \tau_k[c_i - (A^\top y^k)_i]}{1 + \tau_k q_i}\right).$$

Sharding q with x adds no collective to the LP communication pattern.

Table 3 compares general-purpose solvers on the same linearly constrained diagonal-QP formulation at 10^{-6} tolerance. A specialized native-Lasso implementation, such as cuML, operates on the compact Lasso formulation and can exploit problem-specific algebra; a runtime comparison would therefore also measure formulation and specialization effects. The table reports 2/4/8-GPU solving times, with "t" denoting the 7200 s time limit.

L.8 Systems Terminology and Hardware Scope

Memory coalescing combines nearby accesses made by adjacent GPU threads into memory transactions. Cache hit rate is the fraction of requests served by on-chip cache instead of HBM. Thread divergence occurs when threads in a warp follow different control-flow paths. Table 9 evaluates block-wise permutation through solving time. Appendix F explains the memory-traffic model using bytes and arithmetic operations per nonzero.

Our experiments use eight H100 GPUs within one NVLink domain. An NVLink domain is a set of GPUs connected by the NVLink fabric. NVIDIA GB200 NVL72 connects 72 GPUs in such a domain (NVIDIA, 2026). Such larger NVLink domains motivate future scaling studies.

L.9 Relevance to Machine Learning Optimization

First-order LP methods support a broader line of machine learning optimization, including PDLP (Applegate et al., 2021), online-learning-based LP algorithms (Gao et al., 2023), and PDHG-unrolled learning-to-optimize (Li et al., 2024). LP formulations also arise in large-scale Markov decision problems (Malek et al., 2014), while optimal transport connects LP optimization (Zhang et al., 2025) to generative modeling (Kornilov et al., 2024). D-PDLP supplies general-purpose multi-GPU optimization infrastructure for such workloads.