Primal-Dual Hybrid Gradient Algorithm for Linear Programming

Zhonglin Xie

Beijing International Center for Mathematical Research
Peking University

April 10, 2025

Outline

Mathematical Formulation: LP Problem

General LP Problem:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & c^\top x \\ \text{s.t.} \quad & \ell_c \leq Ax \leq u_c \\ & \ell_v \leq x \leq u_v \end{aligned}$$

Deriving the Lagrangian:

- Introduce dual variables for constraints to form unconstrained problem
- Rewrite constraints: $Ax - u_c \leq 0$, $\ell_c - Ax \leq 0$, $x - u_v \leq 0$, $\ell_v - x \leq 0$
- Associate non-negative multipliers y^-, y^+, r^-, r^+ with each inequality

Lagrangian Function:

$$\mathcal{L}(x, y^-, y^+, r^-, r^+) = c^\top x + (y^-)^\top (Ax - u_c) + (y^+)^\top (\ell_c - Ax) \quad (1)$$
$$+ (r^-)^\top (x - u_v) + (r^+)^\top (\ell_v - x) \quad (2)$$

where all multipliers are non-negative

Deriving the Dual Problem

Derivation: Group x terms and form the dual function

$$\mathcal{L}(x, y^-, y^+, r^-, r^+) = x^\top (c + A^\top (y^- - y^+) + (r^- - r^+)) \tag{3}$$
$$- (y^-)^\top u_c + (y^+)^\top \ell_c - (r^-)^\top u_v + (r^+)^\top \ell_v \tag{4}$$

The minimum over x is $-\infty$ unless $c + A^\top (y^- - y^+) + (r^- - r^+) = 0$

Dual Problem: Using substitutions $y = y^+ - y^-$ and $r = r^+ - r^-$

$$\begin{aligned} \max_{y \in \mathbb{R}^m, r \in \mathbb{R}^n} \quad & - (y^-)^\top u_c + (y^+)^\top \ell_c - (r^-)^\top u_v + (r^+)^\top \ell_v \\ \text{s.t.} \quad & c - A^\top y = r \\ & y^-, y^+, r^-, r^+ \geq 0 \end{aligned}$$

Dual Problem Formulation

Simplified notation: Define $p(y; \ell, u) := u^\top y^+ - \ell^\top y^-$ where $y^+ = \max(y, 0)$ and $y^- = \max(-y, 0)$

Rewriting the dual:

$$\begin{aligned} \max_{y \in \mathbb{R}^m, r \in \mathbb{R}^n} \quad & -p(-y; \ell_c, u_c) - p(-r; \ell_v, u_v) \\ \text{s.t.} \quad & c - A^\top y = r \\ & y \in \mathcal{Y},\ r \in \mathcal{R} \end{aligned}$$

Dual feasible sets $\mathcal{Y}$ and $\mathcal{R}$: Based on constraint types

$$\mathcal{Y}_i := \begin{cases} \{0\} & (\ell_c)_i = -\infty, (u_c)_i = \infty \text{ (unconstrained)} \\ \mathbb{R}^- & (\ell_c)_i = -\infty, (u_c)_i \in \mathbb{R} \text{ (upper bound)} \\ \mathbb{R}^+ & (\ell_c)_i \in \mathbb{R}, (u_c)_i = \infty \text{ (lower bound)} \\ \mathbb{R} & \text{otherwise (both upper and lower bounds)} \end{cases} \quad \mathcal{R}_i := \begin{cases} \{0\} & (\ell_v)_i = -\infty, (u_v)_i = \infty \\ \mathbb{R}^- & (\ell_v)_i = -\infty, (u_v)_i \in \mathbb{R} \\ \mathbb{R}^+ & (\ell_v)_i \in \mathbb{R}, (u_v)_i = \infty \\ \mathbb{R} & \text{otherwise} \end{cases}$$

Saddle Point Formulation of LP

- Keeping the bounds on x, we obtain the Lagrangian function:

$$\mathcal{L}(x, y^-, y^+) = c^\top x + (y^-)^\top (Ax - u_c) + (y^+)^\top (\ell_c - Ax)$$

- Using the notation $y = y^+ - y^-$ and the function $p(y; \ell, u) = u^\top y^+ - \ell^\top y^-$:

$$\begin{aligned}(y^-)^\top (Ax - u_c) + (y^+)^\top (\ell_c - Ax) &= -((y^-)^\top u_c - (y^+)^\top \ell_c) + (y^- - y^+)^\top (Ax) \\ &= -p(-y; \ell_c, u_c) - y^\top (Ax)\end{aligned}$$

- Then the saddle point problem is:

$$\min_x \max_y \{c^\top x + y^\top (Ax) - p(-y; \ell_c, u_c)\} \quad \text{s.t. } \ell_v \le x \le u_v$$

Final saddle point formulation:

$$\min_{x \in \mathcal{X}} \max_{y \in \mathcal{Y}} \mathcal{L}(x, y) := c^\top x + y^\top Ax - p\,(y; -u_c, -\ell_c)$$

where $\mathcal{X} := \{x \in \mathbb{R}^n : \ell_v \le x \le u_v\}$

Generic Convex-Concave Saddle Point Problems

General Form:

$$\min_{x \in \mathcal{X}} \max_{y \in \mathcal{Y}} \mathcal{L}(x, y) = \langle Kx, y \rangle + g(x) - f^*(y)$$

Key components:

- K: Linear operator (matrix) mapping primal to dual space
- $g(x)$: Convex function (often includes constraints on x)
- $f^*(y)$: Convex conjugate of function f (handles dual constraints)

Convex Conjugate Definition: For any convex function f

$$f^*(y) = \sup_x \{\langle x, y \rangle - f(x)\}$$

This transforms constraints into penalties in the optimization

PDHG: Abstract Form

Primal-Dual Hybrid Gradient Algorithm:

$$x^{k+1} = \text{prox}_{\tau g}(x^k - \tau K^* y^k) \tag{5}$$

$$y^{k+1} = \text{prox}_{\sigma f^*}(y^k + \sigma K(2x^{k+1} - x^k)) \tag{6}$$

Proximal Operator: A generalization of projection

$$\text{prox}_{\tau g}(z) = \arg\min_{x \in \mathcal{X}} \left\{ g(x) + \frac{1}{2\tau}\|x - z\|^2 \right\}$$

Moreau Decomposition: Allows computing proximal operator of f^* using f

$$\text{prox}_{\sigma f^*}(y) = y - \sigma \cdot \text{prox}_{f/\sigma}(y/\sigma)$$

This is crucial for implementing PDHG efficiently without explicitly forming the conjugate function!

Applying PDHG to LP Problems

For LP saddle point problem:

$$\min_{x \in \mathcal{X}} \max_{y \in \mathcal{Y}} \mathcal{L}(x, y) := c^\top x + y^\top A x - p\left(y; -u_c, -\ell_c\right)$$

We identify:

- $K = A$ (linear constraint matrix)
- $g(x) = c^\top x + \delta_{\mathcal{X}}(x)$ (objective + variable bounds)
- $f^*(y) = p(y; -u_c, -\ell_c)$ (constraint bounds)

Computing proximal operators:

$$\text{prox}_{\tau g}(z) = \text{proj}_{\mathcal{X}}(z - \tau c) \tag{7}$$

$$\text{prox}_{\sigma f^*}(y) = y - \sigma \cdot \text{proj}_{[-u_c, -\ell_c]}(y/\sigma) \tag{8}$$

where projections enforce the constraints efficiently

PDHG Algorithm for LP

PDHG iterations for LP:

$$x^{k+1} = \text{proj}_{[\ell_v, u_v]}(x^k - \tau(c - A^\top y^k)) \tag{9}$$

$$\tilde{y}^{k+1} = y^k - \sigma A(2x^{k+1} - x^k) \tag{10}$$

$$y^{k+1} = \tilde{y}^{k+1} - \sigma \text{proj}_{[-u_c, -\ell_c]}(\tilde{y}^{k+1}/\sigma) \tag{11}$$

Key benefits:

- Only requires matrix-vector products (Ax and $A^\top y$)
- Projections computed element-wise (highly parallelizable)
- No matrix factorization or linear systems to solve
- Memory-efficient for very large-scale problems

Convergence Theory for PDHG on LP

Step Size Parameterization:

- $\tau = \eta/\omega$ and $\sigma = \omega\eta$ with $\eta \in (0, \infty), \omega \in (0, \infty)$
- Convergence guaranteed when $\eta < 1/\|A\|_2$
- ω: primal weight, controls scaling between primal and dual iterates

Special Norm for Convergence Analysis:

$$\|z\|_\omega := \sqrt{\omega\|x\|_2^2 + \frac{\|y\|_2^2}{\omega}}$$

for $z = (x, y)$ - Used in convergence theory, restart criteria, and primal-dual balance

Linear convergence: Under certain conditions, PDHG converges linearly for LP:

$$\|z^k - z^*\|_\omega \leq C(1 - \gamma)^k$$

where $\gamma \in (0, 1)$ depends on problem structure

Problems with Classical Solvers for Large-Scale LP

- **Simplex Method:**
  - Iterations potentially exponential in problem size
  - Poor parallelization on modern hardware
- **Interior Point Methods (IPMs):**
  - Memory requirements: $O(nnz(A))$ for matrix factorization
  - Often exceeds 1TB for problems with billions of nonzeros
- **First-Order Methods:**
  - Low memory requirements
  - Highly parallelizable matrix-vector operations
  - But historically struggle with achieving high accuracy
  - Small constraint violations can lead to significant errors

First-Order Methods for LP

- **State-of-the-art First-Order Method Solvers:**
  - SCS: ADMM-based solver with homogeneous self-dual embedding
  - OSQP: ADMM-based for convex quadratic programming
  - ECLIPSE: Gradient descent on smoothed dual formulation
  - ABIP/ABIP+: Interior-point solvers using ADMM
- **PDHG (Primal-Dual Hybrid Gradient) Advantages:**
  - Requires only matrix-vector products: Ax and $A^\top y$
  - No matrix factorization or systems of equations
  - Form of operator splitting (related to ADMM)
  - Linear convergence for LP established in theory

Outline

PDLP: Main Algorithm

Algorithm PDLP (after preconditioning and presolve)

Input: An initial solution $z^{0,0}$
Initialize outer loop counter $n \leftarrow 0$, total iterations $k \leftarrow 0$
Initialize step size $\hat{\eta}^{0,0} \leftarrow 1/\|A\|_\infty$, primal weight $\omega^0 \leftarrow \texttt{InitializePrimalWeight}$
repeat
  $t \leftarrow 0$
  repeat
    $z^{n,t+1}, \eta^{n,t+1}, \hat{\eta}^{n,t+1} \leftarrow \texttt{AdaptiveStepOfPDHG}(z^{n,t}, \omega^n, \hat{\eta}^{n,t}, k)$
    $\bar{z}^{n,t+1} \leftarrow \frac{1}{\sum_{i=1}^{t+1} \eta^{n,i}} \sum_{i=1}^{t+1} \eta^{n,i} z^{n,i}$
    $z_c^{n,t+1} \leftarrow \texttt{GetRestartCandidate}(z^{n,t+1}, \bar{z}^{n,t+1}, z^{n,0})$
    $t \leftarrow t+1$, $k \leftarrow k+1$
  until restart or termination criteria holds
  restart the outer loop. $z^{n+1,0} \leftarrow z_c^{n,t}$, $n \leftarrow n+1$
  $\omega^n \leftarrow \texttt{PrimalWeightUpdate}(z^{n,0}, z^{n-1,0}, \omega^{n-1})$
until termination criteria holds
Output: $z^{n,0}$

Practical Note: Restart criteria checked every 64 iterations to reduce overhead.

Practical Improvements for PDHG

- **Adaptive step sizes:** Dynamic adjustment based on convergence conditions
- **Restart strategies:** Reset algorithm when progress slows
- **Primal weight updates:** Balance progress in primal and dual spaces
- **Diagonal preconditioning:** Rescale problem for better numerical properties
- **Infeasibility detection:** Efficiently identify infeasible problems
- **Multithreading:** Exploit parallel computing architectures

Adaptive Step Size: Key Insights

Traditional PDHG: Fixed step size $\eta = \frac{1}{\|A\|_2}$

- Overly pessimistic
- Requires estimation of $\|A\|_2$

PDLP Approach: Adaptive step size based on convergence condition

- Calculate maximum allowable step size:

$$\bar{\eta} = \frac{\|z^{k+1} - z^k\|_\omega^2}{2(y^{k+1} - y^k)^\top A(x^{k+1} - x^k)}$$

- Stepsize selection process:
  - Initialize $\eta^{0,0} = 1/\|A\|_\infty$
  - For each step, try current η and compute $\bar{\eta}$
  - Accept step if $\eta \leq \bar{\eta}$; otherwise reduce η and retry
  - Update η' for next iteration with carefully designed formula

Adaptive Step Size Algorithm

Algorithm One step of PDHG using our step size heuristic

function `AdaptiveStepOfPDHG`$(z^{n,t}, \omega^n, \hat{\eta}^{n,t}, k)$
$(x, y) \leftarrow z^{n,t}$, $\eta \leftarrow \hat{\eta}^{n,t}$
for $i = 1, \ldots, \infty$ **do**
  $x' \leftarrow \text{proj}_{\mathcal{X}}(x - \frac{\eta}{\omega^n}(c - A^\top y))$
  $y' \leftarrow y - A(2x' - x) - \eta\omega^n \text{proj}_{[\ell_c, u_c]}\left((\eta\omega^n)^{-1} y - A(2x' - x)\right)$
  $\bar{\eta} \leftarrow \frac{\|(x'-x, y'-y)\|_{\omega,n}^2}{2(y'-y)^\top A(x'-x)}$
  $\eta' \leftarrow \min\left((1 - (k+1)^{-0.3})\bar{\eta}, (1 + (k+1)^{-0.6})\eta\right)$
  if $\eta \leq \bar{\eta}$ **then**
    return $(x', y'), \eta, \eta'$
  end if
  $\eta \leftarrow \eta'$
end for

Adaptive Restarts: Normalized Duality Gap - Properties and Advantages

Definition (From Applegate et al. 2023):

$$\rho_r^n(z) := \frac{1}{r} \max_{(\hat{x},\hat{y}) \in \{\hat{z} \in Z : \|\hat{z} - z\|_{\omega^n} \leq r\}} \{\mathcal{L}(x, \hat{y}) - \mathcal{L}(\hat{x}, y)\}$$

Lagrangian: $\mathcal{L}(x, y) = c^\top x + y^\top Ax - p(y; -u_c, -\ell_c)$

Key Properties:

- Unlike standard duality gap, always finite (bounded by search radius)
- Zero if and only if solution is optimal
- Computable in linear time
- Provides a meaningful measure of progress toward optimality

Adaptive Restart Criteria - Detailed

Notation for Convenience:

$$\mu_n(z, z_{\text{ref}}) := \rho^n_{\|z - z_{\text{ref}}\|_{\omega^n}}(z)$$

where z_{ref} is a user-chosen reference point (typically start of current restart).

PDLP Parameters:

$$\beta_{\text{necessary}} = 0.9, \quad \beta_{\text{sufficient}} = 0.1, \quad \beta_{\text{artificial}} = 0.5$$

Criterion 1: Sufficient Decay

$$\mu_n(z_c^{n,t+1}, z^{n,0}) \leq \beta_{\text{sufficient}} \mu_n(z^{n,0}, z^{n-1,0})$$

- Guarantees linear convergence on LP problems
- Threshold of 10% of initial gap is aggressive

Adaptive Restart Criteria - Detailed

Criterion 2: Necessary Decay + No Progress

$$\mu_n(z_c^{n,t+1}, z^{n,0}) \leq \beta_{\text{necessary}} \mu_n(z^{n,0}, z^{n-1,0})$$
$$\text{and} \quad \mu_n(z_c^{n,t+1}, z^{n,0}) > \mu_n(z_c^{n,t}, z^{n,0})$$

- Inspired by restart schemes for accelerated gradient descent
- Triggers restart when progress begins to stall

Criterion 3: Long Inner Loop

$$t \geq \beta_{\text{artificial}} k$$

- Ensures primal weight is updated frequently enough
- Prevents a bad initial primal weight from causing long-term stalling

Restart Mechanism - Implementation Details

Restart Candidate Selection:

$$\text{GetRestartCandidate}(z^{n,t+1}, \bar{z}^{n,t+1}, z^{n,0}) = \begin{cases} z^{n,t+1} & \text{if } \mu_n(z^{n,t+1}, z^{n,0}) < \mu_n(\bar{z}^{n,t+1}, z^{n,0}) \\ \bar{z}^{n,t+1} & \text{otherwise} \end{cases}$$

Implementation Note:

- Restart criteria evaluated every 64 iterations to reduce overhead
- Makes minimal impact on total iteration count
- Running averages $\bar{z}^{n,t+1}$ weighted by step sizes

Primal Weight Updates

Motivation: Balance progress in primal and dual spaces

- The ω-norm balances primal and dual iterates:

$$\|z\|_\omega = \sqrt{\omega \|x\|_2^2 + \frac{\|y\|_2^2}{\omega}}$$

- For optimal convergence, we want equal progress in both spaces:

$$\|(x^{n,t} - x^*, \mathbf{0})\|_{\omega^n} = \|(\mathbf{0}, y^{n,t} - y^*)\|_{\omega^n}$$

- This yields the ideal primal weight:

$$\omega^n = \frac{\|y^{n,t} - y^*\|_2}{\|x^{n,t} - x^*\|_2}$$

Primal Weight Update Algorithm

Implementation:

- Since x^* and y^* are unknown, estimate using consecutive iterates:

$$\Delta_x^n = \|x^{n,0} - x^{n-1,0}\|_2, \quad \Delta_y^n = \|y^{n,0} - y^{n-1,0}\|_2$$

- Apply log-scale exponential smoothing to avoid oscillations:

$$\omega^n = \exp\left(\theta \log\left(\frac{\Delta_y^n}{\Delta_x^n}\right) + (1-\theta)\log(\omega^{n-1})\right)$$

Primal Weight Update Algorithm

Algorithm Primal weight update

function `PrimalWeightUpdate`$(z^{n,0}, z^{n-1,0}, \omega^{n-1})$
$\Delta_x^n = \|x^{n,0} - x^{n-1,0}\|_2$, $\Delta_y^n = \|y^{n,0} - y^{n-1,0}\|_2$
if $\Delta_x^n > \varepsilon_0$ and $\Delta_y^n > \varepsilon_0$ **then**
    return $\exp\left(\theta \log\left(\frac{\Delta_y^n}{\Delta_x^n}\right) + (1-\theta)\log(\omega^{n-1})\right)$
else
    return ω^{n-1}
end if

Key innovation: Updates only occur after restarts

- Allows larger weight changes without causing instability
- Focuses on balancing distance traveled rather than residuals
- Significantly improves performance compared to per-iteration updates

Diagonal Preconditioning

Approach: Rescale constraint matrix to $\tilde{A} = D_1 A D_2$

- Transforms variables: $\tilde{x} = D_2^{-1} x$
- Transforms constraints: $\tilde{\ell}_c = D_1 \ell_c, \tilde{u}_c = D_1 u_c$
- Transforms bounds: $\tilde{\ell}_v = D_2^{-1} \ell_v, \tilde{u}_v = D_2^{-1} u_v$

PDLP Implementation: Hybrid approach

- Apply 10 iterations of Ruiz scaling:

$$(D_1)_{jj} = \sqrt{\|A_{j,\cdot}\|_\infty}, \quad (D_2)_{ii} = \sqrt{\|A_{\cdot,i}\|_\infty}$$

- Followed by Pock-Chambolle scaling:

$$(D_1)_{jj} = \sqrt{\|A_{j,\cdot}\|_1}, \quad (D_2)_{ii} = \sqrt{\|A_{\cdot,i}\|_1}$$

Infeasibility Detection

Challenge: Efficiently identify infeasible LP instances

PDLP Approach: Check multiple certificate candidates

- Difference of consecutive iterates
- Normalized iterates
- Normalized running average (since last restart)

Advantages:

- Minimal computational overhead
- Different sequences converge at different rates depending on problem geometry
- Checking all three is often faster than relying on just one

Practical Performance

Implementation Details:

- Evaluate restart/termination criteria only every 64 iterations
- Check termination at start and after numerical errors
- Implemented in C++, available in Google OR-Tools

Success Metrics:

- Ability to solve large-scale problems with **billions** of nonzeros
- Strong performance on sparse problems with complex constraints
- Effective utilization of parallel computing resources

Theoretical Guarantees for Enhancements

- PDLP enhancements are motivated by theoretical insights, but not all preserve theoretical guarantees
- **Convergence guarantees:**
  - Adaptive step size: No formal proof of convergence yet
  - Primal weight updates: Practical benefits observed empirically
  - Adaptive restarts: Proven to maintain convergence in simpler settings
- **Infeasibility detection:**
  - Theory developed for non-restarted PDHG iterations
  - Multiple certificate candidates improve practical performance
- **Balance:** PDLP strikes a careful balance between theoretical foundations and practical performance

Outline

The Feasibility Challenge for First-Order Methods

Problem with First-Order Methods:

- Traditional FOMs converge slowly to high-precision solutions
- Small constraint violations can lead to significant errors
- Example: SCS solver with tolerance 10^{-4} had 13% error in objective
- Tighter tolerance (10^{-8}) failed to converge in 1 hour

Feasibility Polishing: Key Insights

Key Insight: In practice, approximately optimal but **exactly feasible** solutions are often acceptable:

- Integer programming solvers commonly terminate at small optimality gaps (e.g., 1%)
- Many applications require strict feasibility but can tolerate small optimality gaps
- First-order methods need a mechanism to achieve tight feasibility without sacrificing speed

Goal: Find solutions with:

- Extremely small feasibility violations (e.g., 10^{-8})
- Moderate duality gaps (e.g., 1%)
- Much faster than solving to full optimality

Feasibility Polishing: Key Insights

Two Fundamental Observations:

- **Insight 1:** PDHG converges much faster for feasibility problems than optimality problems

$$\text{Feasibility problem: } \min_{x \in \mathbb{R}^n} 0 \text{ subject to constraints}$$

- **Insight 2:** Starting from an approximately optimal solution, PDHG will find a *nearby* feasible solution
  - PDHG has non-increasing distance to optimal solutions
  - Warm-starting from approximately optimal point preserves objective quality

Feasibility Polishing: Three-Phase Algorithm

Algorithm Feasibility Polishing Algorithm

1: Run PDLP on original LP. After k iterations, if relative gap $< \epsilon_{\text{rel-gap}}$, pause.
2: Run PDLP on **primal feasibility problem**, starting from $(x^k, 0)$:
- Run for $k/8$ iterations or until primal violations $< 10^{-8}$
- If not converged, return to step 1
- Otherwise, let $\tilde{x}$ be the output solution

3: Run PDLP on **dual feasibility problem**, starting from $(0, y^k)$:
- Run for $k/8$ iterations or until dual violations $< 10^{-8}$
- Let $\tilde{y}$ be the output dual solution

4: Check if $(\tilde{x}, \tilde{y})$ meets termination criteria, if not return to step 1

Implementation Details:

- Primal feasibility problem focuses solely on satisfying constraints
- Dual feasibility problem focuses on satisfying optimality conditions
- Warm-starting from nearly optimal solution preserves objective quality

Practical Benefits of Feasibility Polishing

Dramatic Performance Improvement:

- **Convergence speed:** Orders of magnitude faster than solving to full optimality
- **Memory efficiency:** No additional memory requirements
- **Solution quality:** Maintains excellent objective values while achieving exact feasibility

Cost-Benefit Analysis:

- Extra iterations on simpler problems (feasibility) vs. extra iterations on harder problem (optimality)
- Solving feasibility is significantly faster than forcing high-precision convergence
- Trade-off: Allows controllable balance between feasibility and optimality

Outline

PDLP Implementation: From Research to Practice

C++ Implementation in Google OR-Tools:

- First released with OR-Tools version 9.3 in March 2022
- Designed for production deployments
- Accessible via OR-Tools framework and CVXPY

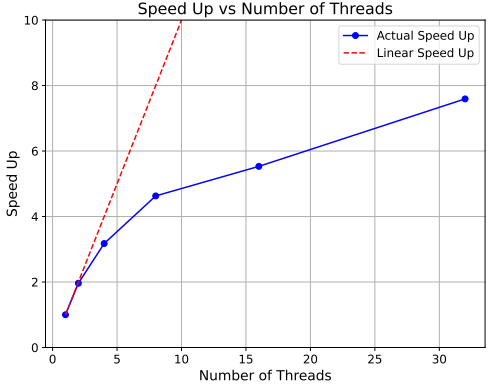
Key Improvements Over Research Prototype:

- **Multithreading:** Parallelized operations for matrix-vector products, projections, etc.
  - Sharded implementation divides data among threads
  - Default: $4\times$ more shards than threads for balanced workload
- **General LP form:** Supports both lower and upper bounds on constraints
- **Presolve:** Integrated presolving capabilities from GLOP (OR-Tools simplex solver)

PDLP: Multithreading Performance

Thread Scaling Results:

- Almost 8× speedup with 32 threads on large instances
- Effective parallelization of:
  - Sparse matrix-vector operations
  - Vector-vector operations
  - Projections and norms
- Benefits increase with problem size



Current Limitations:

- Thread synchronization overhead
- Inefficient memory bandwidth usage
- Research opportunities: Apply SpMV optimization techniques

Large-Scale Benchmark Suite

Motivation & Design:

- Designed to challenge or exceed state-of-the-art solver capabilities
- Diverse problem domains and structures
- All instances fit in memory of high-end machines (1TB)
- Smallest: 125M nonzeros; Largest: 6.3B nonzeros

Instance Scaling:

- Objective vectors standardized to {-1, 0, 1}, right-hand sides standardized to {-1, 0, 1} with special handling for two-sided constraints
- Generator: `https://github.com/ohinder/large-scale-LP-test-problems`
- Public availability: `https://www.oliverhinder.com/large-scale-lp-problems`

Benchmark Instance Details

Problem Domains:

- **Statistical matching:** 2.76B nonzeros (design-match)
- **TSP relaxations:** 475M—6.3B nonzeros (tsp-gaia-10m, tsp-gaia-100m)
- **PDE-constrained optimization:** 125M nonzeros (heat-source-easy/hard)
- **Production-inventory:** 500M nonzeros
- **Quadratic assignment relaxations:** 198M—1B nonzeros (qap-wil-100, qap-tho-150)
- **Shipping network optimization:** 629M—689M nonzeros
- **Supply chain optimization:** 403M nonzeros

Challenge: Most instances beyond reach of classical methods

Experimental Setup

Hardware:

- AMD EPYC 7302 (16 cores, 32 threads, 256GB RAM)
- Intel Xeon Platinum 8352Y (32 cores, 64 threads, 1TB RAM)
- PDLP: 16 threads on 256GB machine, 32 threads on 1TB machine
- Gurobi: Default thread settings (auto-tuned to machine)
- First ran a solver on an instance on the machine with 256 GB of memory and then only switched to the 1 TB machine if insufficient memory was available
- Cost: around $20,000 to run on Google cloud (excluding the cost of building and rescaling the instances)

Experimental Setup

Solver Configurations:

- **PDLP:** With and without feasibility polishing
- **Gurobi 11.0.2:** Barrier, primal simplex, and dual simplex (crossover disabled)

Termination Criteria:

- **PDLP:** ℓ_∞ primal/dual residuals $< 10^{-8}$, relative gap $< 1\%$:

$$\frac{\left|c^\top x + p\left(y; \ell_c, u_c\right) + p\left(r; \ell_v, u_v\right)\right|}{\max\left\{\left|c^\top x\right|, \left|p\left(y; \ell_c, u_c\right) + p\left(r; \ell_v, u_v\right)\right|\right\}} \leq 10^{-2}$$

- **Gurobi:** Default settings (ℓ_∞ tolerance 10^{-6}, gap 10^{-8})

"We do not believe that matching Gurobi's termination criteria to ours would make a significant difference to the results. In particular, Gurobi's barrier implementation mostly experiences OOM errors, and so is unaffected by the termination criteria; and simplex methods only find a primal and dual feasible solution at the optimum."

Performance Comparison: PDLP vs. Commercial Solvers

Key Findings:

- **Memory efficiency:** Gurobi barrier hit OOM on 8/11 instances
- **PDLP with polishing:** Solved 8/11 instances
- **PDLP without polishing:** Solved only 2/11 instances
- **Gurobi simplex:** Competitive in memory usage but slower (dual simplex solved only 4/11 instances within 144 hours)

Solution Quality:

- Relative duality gaps from PDLP typically much better than 1% target
- Most instances: $< 0.1\%$ gap
- Shipping instances: $< 0.4\%$ gap

Performance Results: Solve Time (Hours)

	PDLP		Gurobi		
Instance name	**Without polishing**	**With polishing**	**Barrier**	**Dual**	**Primal**
design-match	68.0	**9.3**	OOM	75.3	32.4
tsp-gaia-10m	TIME	**3.0**	28.1	90.1	40.2
tsp-gaia-100m	TIME	**21.1**	OOM	OOM	OOM
heat-source-easy	TIME	**60.0**	OOM	ERR	TIME
heat-source-hard	TIME	TIME	OOM	ERR	TIME
production-inventory	TIME	TIME	**5.8**	TIME	43.7
qap-tho-150	ERR	ERR	OOM	TIME	TIME
qap-wil-100	44.4	**0.3**	OOM	TIME	TIME
world-shipping	TIME	**53.8**	OOM	TIME	TIME
mediterranean-shipping	TIME	**32.6**	OOM	TIME	TIME
supply-chain	TIME	18.9	**2.5**	4.2	TIME

Legend:

- **TIME:** Exceeded 144 hours limit
- **OOM:** Out of memory (exceeded 1TB RAM)
- **ERR:** Solver encountered an error

Solution Quality: PDLP with Feasibility Polishing

Instance name	Primal objective	Absolute gap	Relative gap (%)
design-match	2.87×10^{6}	5.53×10^{-2}	0.0000001
tsp-gaia-10m	1.3×10^{8}	3.01×10^{3}	0.0012
tsp-gaia-100m	1.94×10^{9}	5.82×10^{4}	0.0015
heat-source-easy	5.16×10^{7}	9.06×10^{-4}	0.00000000088
qap-wil-100	2.21×10^{5}	1.14×10^{2}	0.026
world-shipping	-1.18×10^{5}	3.7×10^{2}	0.16
mediterranean-shipping	-1.07×10^{3}	6.95×10^{0}	0.33
supply-chain	1.08×10^{8}	6.67×10^{3}	0.0031

Key Observations:

- All problems solved with duality gaps substantially better than 1% target
- Six instances achieved gaps $< 0.1\%$
- Heat-source-easy achieved near-perfect solution quality (gap $\approx 10^{-12}\%$)
- Shipping problems (more challenging) still achieved $< 0.4\%$ gaps

Key Takeaways from Numerical Experiments

- **Feasibility polishing is transformative:**
  - 4× increase in problems solved
  - 2-148× speedup over vanilla PDLP
  - Delivers high-quality, feasible solutions with small gaps
- **Memory efficiency makes previously unsolvable problems tractable:**
  - Successfully handles problems with billions of nonzeros
  - Solves problems where interior point methods run out of memory
- **Competitive time-to-solution on very large instances:**
  - Outperforms simplex methods on most large instances
  - Barrier method faster when problems fit in memory

Outline

GPU vs. CPU Architecture: Why GPUs for LP?

CPU Design:

- Few cores (16-64) with deep pipelines
- Optimized for sequential processing
- Sophisticated branch prediction
- Limited memory bandwidth

GPU Design:

- Thousands of cores (e.g., 7296 on NVIDIA H100)
- Single Instruction Multiple Data (SIMD)
- Optimized for parallel computation
- Very high memory bandwidth (2 TB/sec vs 137 GB/sec)

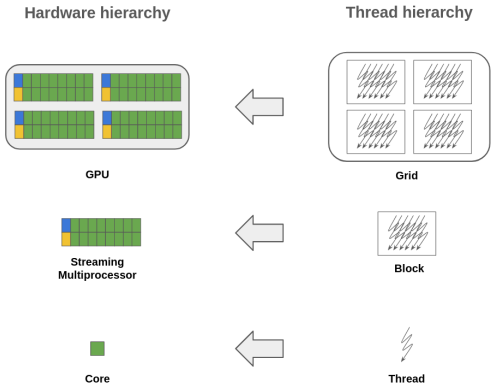
Key Challenge: Previous attempts at GPU acceleration for LP failed due to:

- Matrix factorization in simplex/IPM doesn't parallelize well
- First-order methods (like PDHG) rely on matrix-vector operations - ideal for GPUs!

GPU Thread Hierarchy and Execution Model

Thread Hierarchy:

- **Thread:** Basic execution unit
- **Warp:** 32 threads executing in lockstep
- **Block:** Group of threads with shared memory
- **Grid:** Collection of blocks executing same kernel



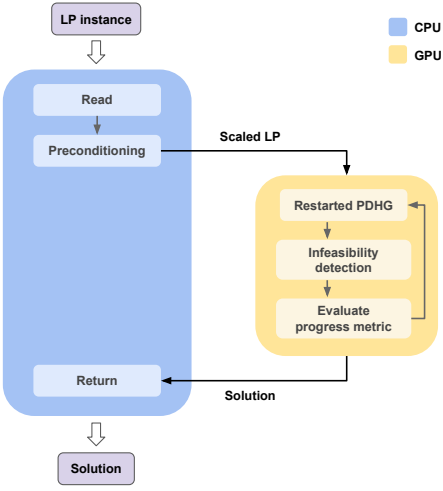
Implications for PDLP:

- Matrix-vector operations can be massively parallelized
- Each thread can process individual vector elements
- Challenge: Reducing CPU-GPU communication overhead

cuPDLP.jl: Design Principles

Minimizing CPU-GPU Communication:

- Initial transfer: Problem instance from CPU to GPU
- Final transfer: Solution from GPU to CPU
- All iterations computed entirely on GPU



Efficient GPU Implementation Details

Implementation Framework:

- Implemented in Julia using CUDA.jl
- Custom CUDA kernels for PDHG updates
- cuSPARSE library for sparse matrix operations

Matrix and Vector Operations:

- Sparse matrix stored in Compressed Sparse Row (CSR) format
- Matrix-vector multiplication via cuSPARSE library
- Custom CUDA kernels for vector operations and projections
- One thread per vector element for maximum throughput

Efficient GPU Implementation Details

KKT-Based Restart Scheme:

- Original PDLP: Trust-region algorithm for normalized duality gap
  - Sequential nature - poor fit for GPU architecture
- cuPDLP.jl innovation: KKT error-based restart
  - Highly parallelizable computation
  - Maintains convergence properties
  - Better suited for GPU execution model

KKT-Based Restart Details

KKT Error Definition:

$$\mathrm{KKT}_\omega(z) = \sqrt{\omega^2 \left\| \begin{pmatrix} Ax - b \\ [h - Gx]^+ \end{pmatrix} \right\|_2^2 + \frac{1}{\omega^2}\|c - K^\top y - \lambda\|_2^2 + (q^\top y + l^\top \lambda^+ - u^\top \lambda^- - c^\top x)^2}$$

Restart Candidate Selection:

$$z_c^{n,t+1} = \begin{cases} z^{n,t+1} & \text{if } \mathrm{KKT}_{\omega^n}(z^{n,t+1}) < \mathrm{KKT}_{\omega^n}(\bar{z}^{n,t+1}) \\ \bar{z}^{n,t+1} & \text{otherwise} \end{cases}$$

Restart Conditions: Algorithm restarts if any of these holds:

- **Sufficient decay:** $\mathrm{KKT}_{\omega^n}(z_c^{n,t+1}) \leq 0.2 \cdot \mathrm{KKT}_{\omega^n}(z^{n,0})$
- **Necessary decay + no progress:** $\mathrm{KKT}_{\omega^n}(z_c^{n,t+1}) \leq 0.8 \cdot \mathrm{KKT}_{\omega^n}(z^{n,0})$ and no improvement
- **Long inner loop:** Iteration count exceeds threshold

Outline

Experimental Setup

Hardware Configuration:

- **GPU:** NVIDIA H100-PCIe-80GB (26 TFLOPS FP64, 2 TB/sec bandwidth)
- **CPU:** Intel Xeon Gold 6248R 3.00GHz (16 threads, 256 GFLOPS FP64)

Benchmark Sets:

- **MIP Relaxations:** 383 instances from MIPLIB 2017
  - Small: 269 instances (100K-1M nonzeros)
  - Medium: 94 instances (1M-10M nonzeros)
  - Large: 20 instances (>10M nonzeros)
- **Mittelmann's LP:** 49 classic LP benchmark instances

Compared Methods

Compared Methods:

- **cuPDLP.jl:** GPU implementation in Julia
- **PDLP:** CPU implementations (Julia and C++ with 1/4/16 threads)
- **Gurobi 11.0:** Primal simplex, dual simplex, and barrier methods
- PDLP and cuPDLP.jl terminate when the original LP instance satisfies:

$$\left|q^\top y + l^\top \lambda^+ - u^\top \lambda^- - c^\top x\right| \le \epsilon\left(1 + \left|q^\top y + l^\top \lambda^+ - u^\top \lambda^-\right| + \left|c^\top x\right|\right)$$

$$\left\|\begin{pmatrix} Ax - b \\ [h - Gx]^+ \end{pmatrix}\right\|_2 \le \epsilon(1 + \|q\|_2) \qquad \left\|c - K^\top y - \lambda\right\|_2 \le \epsilon(1 + \|c\|_2)$$

  where $\epsilon = 10^{-4}$ for moderately accurate and $\epsilon = 10^{-8}$ for high-quality solutions.
- Gurobi's parameters `FeasibilityTol`, `OptimalityTol`, and `BarConvTol` are set to match the tolerances (10^{-4} or 10^{-8}).

cuPDLP.jl vs. Gurobi: Moderate Accuracy (10^{-4})

	Small (269)		Medium (94)		Large (20)		Total (383)	
	Count	**Time**	**Count**	**Time**	**Count**	**Time**	**Count**	**Time**
cuPDLP.jl	266	8.61	92	14.80	19	111.19	377	12.02
Primal simplex	268	12.56	69	188.81	11	3145.49	348	39.81
Dual simplex	268	8.75	84	66.67	15	591.63	367	21.75
Barrier	268	5.30	88	45.01	18	415.78	374	14.92

Key Observations:

- cuPDLP.jl solves 377/383 instances (98.4%)
- Clear advantage on medium and large instances:
  - 3x faster than simplex on medium instances
  - 3.7x faster than barrier on large instances
- Results with presolve show even better performance

cuPDLP.jl vs. CPU PDLP: Speedup Analysis

	Small (269)		Medium (94)		Large (20)		Total (383)	
	Count	**Time**	**Count**	**Time**	**Count**	**Time**	**Count**	**Time**
cuPDLP.jl	266	8.61	92	14.80	19	111.19	377	12.02
FirstOrderLp.jl	253	35.94	82	155.67	12	2002.21	347	66.67
PDLP (1 thread)	256	22.69	85	98.38	15	1622.91	356	43.81
PDLP (4 threads)	260	24.03	91	42.94	15	736.20	366	34.57
PDLP (16 threads)	238	104.72	84	142.79	15	946.24	337	127.49

GPU Speedup vs. CPU:

- vs. FirstOrderLp.jl (Julia): 4x on small, 10x on medium, 18x on large instances
- vs. PDLP with 4 threads (best CPU): 2.9x overall speedup
- Solved 30 more instances than FirstOrderLp.jl at tolerance 10^{-4}
- Speedup increases with problem size - ideal for large-scale problems

Performance Analysis: Time to Solution

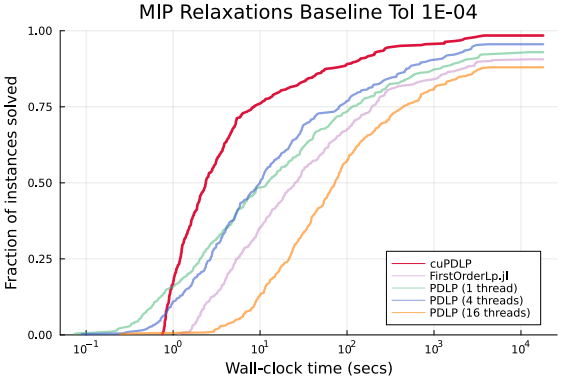


Figure: Performance of cuPDLP.jl vs. PDLP

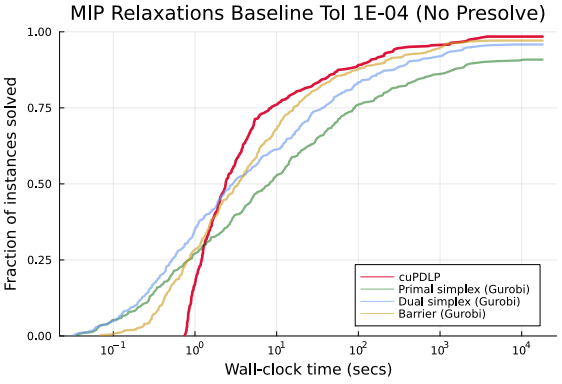


Figure: Performance of cuPDLP.jl vs. PDLP without presolve

Performance Analysis: Key Insights

Key Insights:

- cuPDLP.jl has 1 second overhead for GPU initialization
- After 10 seconds, cuPDLP.jl matches or exceeds barrier method performance
- cuPDLP.jl significantly outperforms all CPU PDLP implementations
- GPU advantage most significant on medium/large problems

Performance on Mittelmann's LP Benchmark

	Tol 1E-04		Tol 1E-08	
	Count	**Time**	**Count**	**Time**
cuPDLP.jl	44	71.26	40	231.91
Primal simplex	35	1937.78	34	1715.62
Dual simplex	36	1201.48	37	1116.68
Barrier	45	108.29	44	127.58

	Tol 1E-04		Tol 1E-08	
	Count	**Time**	**Count**	**Time**
cuPDLP.jl	44	71.26	40	231.91
FirstOrderLp.jl	34	917.49	25	2504.79
PDLP (4 threads)	40	302.54	34	930.41

Observations:

- Similar performance pattern as with MIP Relaxations
- cuPDLP.jl 4.2x faster than best CPU PDLP (4 threads)
- cuPDLP.jl significantly outperforms simplex methods
- Performance comparable to barrier method

Outline

PDHG for Linear Programming

Consider the LP problem $\mathcal{M} = (G; l, u, c; h)$ in standard form

$$\begin{aligned} \min\ & c^\top x \\ \text{s.t. } & Gx \geq h \\ & l \leq x \leq u \end{aligned}$$

where $G \in \mathbb{R}^{m\times n}, h \in \mathbb{R}^m, c \in \mathbb{R}^n, l \in (\mathbb{R} \cup \{-\infty\})^n, u \in (\mathbb{R} \cup \{+\infty\})^n$

Saddle point problem form: $\min\limits_{l\leq x\leq u} \max\limits_{y\geq 0} L(x, y; \mathcal{M}) = c^\top x - y^\top Gx + h^\top y$

Primal-Dual Hybrid Gradient (PDHG)

– Initialize $x^0 \in \mathbb{R}^n,\ y^0 \in \mathbb{R}^m$

For $k = 0, 1, 2, ..., K-1$

$$\begin{cases} x^{k+1} = \mathbf{Proj}_{l\leq x\leq u}(x^k - \tau(c - G^\top y^k)); \\ y^{k+1} = \mathbf{Proj}_{y\geq 0}(y^k + \sigma(h - 2Gx^{k+1} + Gx^k)). \end{cases}$$

Unrolling PDHG into PDHG-Net

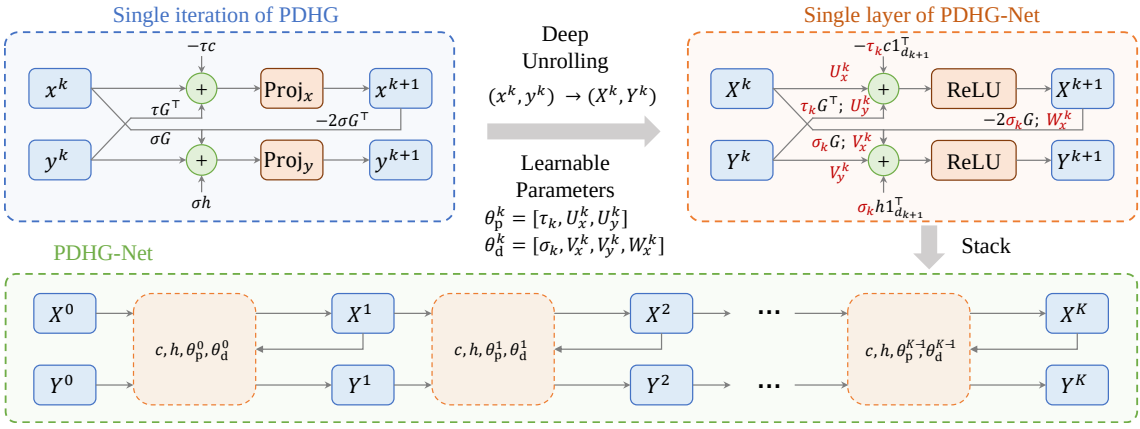


Figure: Overview of how each layer in PDHG-Net corresponds to each iteration of the traditional PDHG algorithm, along with the overall architecture of PDHG-Net.

Key Technique: Channel Expansion

- Expanding the n-dimensional vectors x^k, y^k into $(n \times d_k)$-dimensional matrices X^k, Y^k with d_k columns (or called channels following the convention of neural network)
- The linear combination $x^k - \tau(c - G^\top y^k)$ of primal-dual is replaced by

$$X^k U_x^k - \tau_k (c \cdot \mathbf{1}_{d_{k+1}}^\top - G^\top Y^k U_y^k)$$

  where $\Theta_{\mathrm{p}}^k = (\tau_k, U_x^k, U_y^k) \in \mathbb{R} \times \mathbb{R}^{d_k \times d_{k+1}} \times \mathbb{R}^{d_k \times d_{k+1}}$ is the trainable parameter of the k-th primal NN block
- **Generalizability to LP instances of different sizes**: Following the principle of classical unrolling, a natural idea would be to unroll $x^k - \tau(c - G^\top y^k)$ to

$$x^k - \tau(c - W^k y^k)$$

  where W^k is trainable matrix. This is **unsuitable** for applying to LP problems with different sizes

Architecture of PDHG-Net

$$
\begin{aligned}
&- \text{Initialize } X^0 = [x^0, I, u, c],\ \ Y^0 = [y^0, h] \\
&\text{For}\ \ k = 0, 1, 2, ..., K-1 \\
&\quad \left| \begin{array}{l} X^{k+1} = \text{ReLU}\left(X^k U_x^k - \tau_k (c \cdot \mathbf{1}_{d_{k+1}}^\top - G^\top Y^k U_y^k)\right), \\ Y^{k+1} = \text{ReLU}\left(Y^k V_y^k \right. \\ \qquad\qquad \left. + \sigma_k (h \cdot \mathbf{1}_{d_{k+1}}^\top - 2GX^{k+1} W_x^k + GX^k V_x^k)\right), \end{array} \right. \\
&- \text{Output } X^K \in \mathbb{R}^n,\ \ Y^K \in \mathbb{R}^m
\end{aligned}
$$

The trainable parameter is $\Theta = \{\Theta_{\mathrm{p}}^k, \Theta_{\mathrm{d}}^k\}_{k=0}^{K-1}$, where

$$
\begin{aligned}
\Theta_{\mathrm{p}}^k &= (\tau_k, U_x^k, U_y^k) \in \mathbb{R} \times \mathbb{R}^{d_k \times d_{k+1}} \times \mathbb{R}^{d_k \times d_{k+1}} \\
\Theta_{\mathrm{d}}^k &= (\sigma_k, V_x^k, V_y^k, W_x^k) \in \mathbb{R} \times \mathbb{R}^{d_k \times d_{k+1}} \times \mathbb{R}^{d_k \times d_{k+1}} \times \mathbb{R}^{d_{k+1} \times d_{k+1}}
\end{aligned}
$$

Convergence Property of PDHG

Theorem

Let $(x^k, y^k)_{k\geq 0}$ be the primal-dual variables generated by the PDHG algorithm for the LP problem $\mathcal{M} = (G; l, u, c; h)$. If the step sizes τ, σ satisfy $\tau\sigma\|G\|_2^2 < 1$, then for any $(x, y) \in \mathbb{R}^n \times \mathbb{R}^m_{\geq 0}$ satisfying $l \leq x \leq u$, the primal-dual gap satisfies

$$\begin{aligned} & L(\bar{x}^k, y; \mathcal{M}) - L(x, \bar{y}^k; \mathcal{M}) \\ \leq & \frac{1}{2k}\Big(\frac{\|x - x^0\|^2}{\tau} + \frac{\|y - y^0\|^2}{\sigma} - (y - y^0)^\top G(x - x^0)\Big), \end{aligned}$$

where $\bar{x}^k = (\sum_{j=1}^k x^j)/k$, $\bar{y}^k = (\sum_{j=1}^k y^j)/k$, and L is the Lagrangian defined by LP.

Alignment Theorem: PDHG is a Specific PDHG-Net

Theorem

Given any pre-determined network depth K and the widths $\{d_k\}_{k\leq K-1}$ with $d_k \geq 10$, there exists a K-layer PDHG-Net with its parameter assignment Θ_{PDHG} satisfying the following property: given any LP problem $\mathcal{M} = (G; I, u, c; h)$ and its corresponding primal-dual sequence $(x^k, y^k)_{k\leq K}$ generated by PDHG algorithm within K iterations, we have

1. *For any hidden layer k, both $\bar{x}^k$ and x^k can be represented by a linear combination of X^k's channels, both $\bar{y}^k$ and y^k can be represented by a linear combination of Y^k's channels. Importantly, these linear combinations do not rely on the LP problem $\mathcal{M}$.*
2. *PDHG-Net's output embeddings $X^K \in \mathbb{R}^{n\times 1}$ and $Y^K \in \mathbb{R}^{m\times 1}$ are equal to the outputs $\bar{x}^K$ and $\bar{y}^K$ of the PDHG algorithm, respectively.*

Estimate the Approximation Efficiency

Theorem

Given the approximation error bound ϵ, there exists a PDHG-Net with $\mathcal{O}(1/\epsilon)$ number of neurons and the parameter assignment Θ_{PDHG} fulfilling the following property. For any LP problem $\mathcal{M} = (G; l, u, c; h)$ and $(x, y) \in \mathbb{R}^n \times \mathbb{R}^m_{\geq 0}$ satisfying $l \leq x \leq u$, it holds that

$$L(X^K, y; \mathcal{M}) - L(x, Y^K; \mathcal{M}) < \epsilon.$$

- The proof is rather concise to compare with Bipartite representation theorem
- Give an explicit estimation of the number of neurons required to represent a solution

Numerical Result

- **Training dataset:** A set of instances denoted by $\mathcal{I} = \{(\mathcal{M}, z^*)\}$. The input of an instance is an LP problem $\mathcal{M} = (G; l, u, c; h)$; the label $z^* = (x^*, y^*)$ is the solution
- **Loss Function:** We train the PDHG-Net to minimize the ℓ_2 square loss

$$\min_{\Theta} \mathcal{L}_{\mathcal{I}}(\Theta) = \frac{1}{|\mathcal{I}|} \sum_{(\mathcal{M}, z^*) \in \mathcal{I}} \left\| Z^K(\mathcal{M}; \Theta) - z^* \right\|_2^2$$

- **Metric:** We calculate the improvement ratio over PDLP using the following equation:

$$\text{Improv.} = \frac{\text{PDLP} - \text{ours}}{\text{PDLP}},$$

where this metric is applicable to both the solving time and the number of iterations

Overview of the Datasets

# nodes	# vars.	# cons.	# nnz.
10^3	1,000	1,001	7,982
10^4	10,000	10,001	79,982
5×10^4	50,000	50,001	399,982
10^5	100,000	100,001	799,982
10^6	1,000,000	1,000,001	7,999,982

Table: Sizes of utilized PageRank instances.

dataset	# vars.	# cons.	# nnz.
IP-S	31,350	15,525	5,291,250
IP-L	266,450	91,575	94,826,250
WA-S	80,800	98,830	3,488,784
WA-L	442,000	541,058	45,898,828

Table: Sizes of utilized instances.

Two-stage Algorithm: PDHG-Net as Warm-start

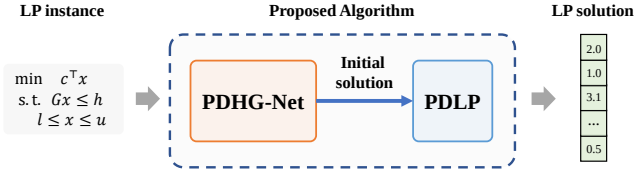


Figure: The proposed post-processing procedure warm-starts the PDLP solver using the prediction of PDHG-Net as initial solutions to ensure optimality.

Table: Solve time comparison between the proposed framework and vanilla PDLP on PageRank instances. The improvement ratio of the solving time is also reported.

# nodes	ours	PDLP	Improv.
10^3	0.01sec.	0.04sec.	↑ 45.7%
10^4	0.4 sec.	1.1sec.	↑ 67.6%
10^5	22.4sec.	71.3sec.	↑ 68.6%
10^6	4,508sec.	16,502sec.	↑ 72.7%

Efficiency and the Number of Restarts

Table: Comparison of the proposed framework against default PDLP in solving IP and WA instances.

dataset.	time (sec.)			# iters.		
	ours	PDLP	Improv.	ours	PDLP	Improv.
IP-S	**9.2**	11.4	↑ 19.5%	**422**	525	↑ 19.5%
IP-L	**7,866.3**	10,045.6	↑ 21.7%	**6,048**	8,380	↑ 27.8%
WA-S	**114.7**	137.8	↑ 16.7%	**8,262**	9,946	↑ 16.9%
WA-L	**4817.6**	6426.2	↑ 25.0%	**14,259**	17,280	↑ 17.5%

Table: The average number of restarts in the PDLP solving process with our framework (ours) and default settings (represented by PDLP).

# of Nodes		5×10^3	1×10^4	2×10^4	4×10^4
# restarts	Ours	2.2	4.15	2.0	2.0
	PDLP	5.9	11.7	20.25	11.3

Prediction Accuracy v.s. Epochs

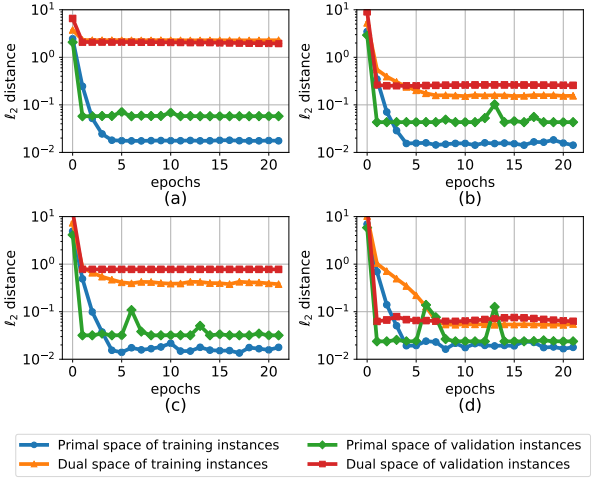
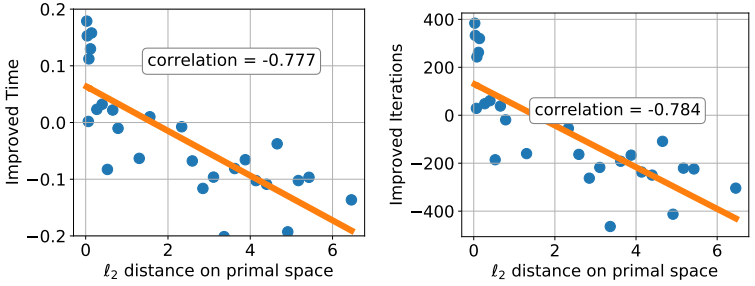


Figure: The distance between the predicted solution of PDHG-Net and optimal solution in PageRank training and validation instances with (a) 5×10^3, (b) 1×10^4, (c) 2×10^4, (d) 4×10^4 variable sizes.

Improvement v.s. Prediction Accuracy



(a) Solving time (b) Number of iterations

Figure: We present the improvement ratio in both solving time and the number of iterations for solutions extrapolated at varying distances from the optimal solution. Each blue dot symbolizes an extrapolated solution, while the yellow line represents the trend line fitted through these points. Results demonstrate a strong correlation.

Generalizability to Larger Sizes

Table: Solving time and number of iterations for PageRank, `IP` and `WA` instances larger than training set sizes. For clarity, we denote the size of the largest instance of `IP` and `WA` datasets as `Large`.

metric	Dataset	size	ours	PDLP	Improv.
time (sec.)	PageRank	5×10^4	**5.5**	11.2	↑ 50.9%
		1×10^5	**17.0**	32.5	↑ 47.8%
	`IP`	`Large`	**6796.7**	8631.4	↑ 21.3%
	`WA`	`Large`	**5599.1**	5859.4	↑ 4.4%
# iter.	PageRank	5×10^4	**1,605**	3,397	↑ 52.7%
		1×10^5	**1,958**	3,914	↑ 50.0%
	`IP`	`Large`	**7,291**	8,970	↑ 18.7%
	`WA`	`Large`	**16,166**	17,280	↑ 6.4%

Portion of GPU Time & Comparison with GNNs

Table: Comparison of total solving time and GPU time for initial solutions, including the ratio of GPU time to total solving time.

# nodes.	10^3	10^4	10^5	10^6
GPU time (sec.)	0.01	0.02	0.21	1.12
CPU time (sec.)	0.02	0.4	22.4	4,508.3
Ratio	52.6%	5.7%	0.9%	0.02%

Table: Comparison of improvement ratio and ℓ_2 distance between the proposed framework implemented with PDHG-Net and GNN.

# nodes.	Improv. ours	Improv. GNN	ℓ_2 distance ours	ℓ_2 distance GNN
10^3	$\uparrow$ 45.7%	$\uparrow$ 1.4%	0.05	0.51
10^4	$\uparrow$ 67.6%	$\uparrow$ 19.3%	0.2	1.38
10^5	$\uparrow$ 71.3%	$\downarrow$ 4.0%	0.95	30.35

Outline

Conclusion: The Role of PDLP

PDLP fills a critical gap in LP solving:

- **Complements traditional solvers:**
  - Interior point methods: Fast but memory-intensive
  - Simplex methods: Low memory but slow on very large problems
  - PDLP: Low memory and reasonable speed on very large problems
- **Feasibility polishing bridges the gap:**
  - Delivers high-quality feasible solutions
  - Overcomes traditional limitations of first-order methods
- **Practical impact:**
  - Enables solution of previously impossible problems
  - Integrated into commercial offerings (COPT, FICO Xpress, etc.)
  - Open source availability expands access to large-scale LP solutions

Summary: GPU Acceleration for LP

Are GPUs useful for solving LP?

- **Yes!** cuPDLP.jl demonstrates comparable or better performance to commercial solvers
- GPU advantage increases with problem size
- First-order methods + GPU = competitive solution for large-scale LP

Key Contributions:

- Efficient GPU implementation of restart scheme using KKT error
- Minimal CPU-GPU communication design
- Reliable for real-world problems (solve rate $>98\%$)
- Superior scalability for large instances compared to CPU implementations

Future Work

Future Work:

- Extension to QP and other problem classes
- Multi-GPU parallelization
- Integration with mixed-integer programming heuristics