HiPDLP: A High-Performance First-Order Solver for Large-Scale LP

Yanyu Zhou

University of Edinburgh

1 June 2026

HiGHS Workshop 2026

The challenge: when simplex and IPM hit the wall

- Modern LPs can involve 10^7 to 10^9 variables.
- **Simplex:** inherently sequential; limited parallel speedup.
- **IPM:** memory-intensive matrix factorizations.

Instance	Variables	Constraints	Gurobi
TSP-GAIA-100M	1.1 Billion	162 Million	OOM
QAP-THO-150	249 Million	6.7 Million	OOM / Timeout

Our direction: replace factorizations with parallel matrix-vector products.

Supported by NESO

Standard form linear programming

We solve primal-dual LP problems of the form:

Primal problem:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & c^T x \\ \text{subject to:} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

- Where $A \in \mathbb{R}^{m \times n}$, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$

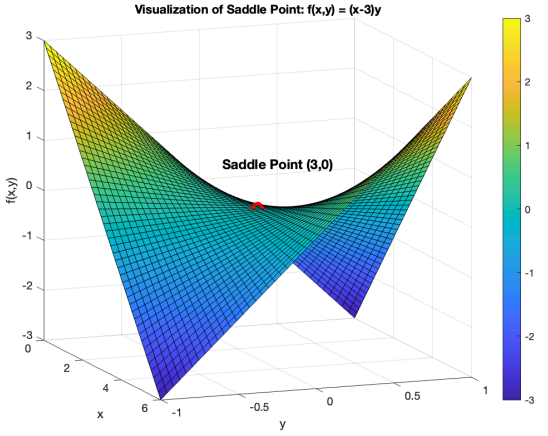
Dual problem:

$$\begin{aligned} \max_{y \in \mathbb{R}^m} \quad & b^T y \\ \text{subject to:} \quad & A^T y \leq c \\ & y \text{ is free} \end{aligned}$$

The saddle-point problem

Linear programming is equivalent to finding the saddle point of

$$\min_{x \geq 0} \max_{y} \mathcal{L}(x, y) = c^T x - y^T (Ax - b)$$



The GDA trap:

- Gradient descent-ascent (GDA) updates:

$$x^{k+1} = \text{proj}(x^k - \eta(c - A^T y^k))$$
$$y^{k+1} = y^k + \sigma(b - Ax^k)$$

- **Problem:** GDA cycles or diverges!

The engine: Primal-dual hybrid gradient (PDHG)

Solve the saddle-point problem: $\min_{x \geq 0} \max_{y} c^T x - y^T (Ax - b)$

The Vanilla PDHG

$$x^{k+1} = \text{proj}_{\mathbb{R}^n_+}(x^k + \eta(A^T y^k - c))$$
$$y^{k+1} = y^k + \sigma(b - A(2x^{k+1} - x^k))$$

Pros:

- Only requires Ax and $A^T y$
- Low memory footprint
- Highly parallelizable (GPU/HPC friendly)

Cons:

- "Vanilla" PDHG is notoriously slow
- Sensitive to problem scaling

From theory to practice: heuristics

PDHG alone isn't enough. **PDLP** (Applegate et al., 2021) introduced critical heuristics that changed the game:

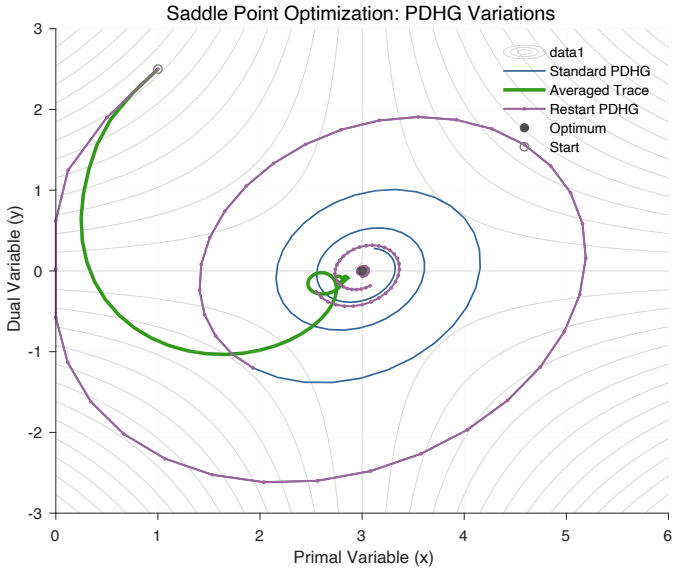
- **Adaptive step sizes:** No more manual tuning of η and σ
- **Diagonal preconditioning:** Balances the matrix A for faster convergence
- **Restarting:** Periodically resetting the algorithm to the average iterate

The Rate Upgrade

By combining PDHG steps with an **Adaptive Restart** scheme, we "upgrade" the convergence:

$$O(1/k) \text{ (Sublinear)} \longrightarrow O(e^{-k}) \textbf{ (Linear Convergence)}$$

Restart


Saddle Point Optimization: PDHG Variations
data1
Standard PDHG
Averaged Trace
Restart PDHG
Optimum
Start
Dual Variable (y)
Primal Variable (x)
3
2
1
0
-1
-2
-3
0
1
2
3
4
5
6

Stabilizing chaos: restarting & averaging

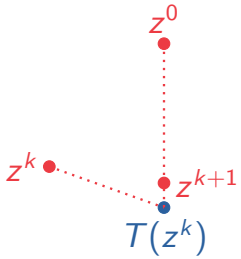
- **Adaptive restart:** We monitor the duality gap and "jump" when progress slows.
- **The result:** A 37× speed-up over vanilla iterations.

Method	Iterations
Vanilla	17,680
+ Scaling	1,680
+ Restarting	**440**

Accelerating via Halpern iteration & reflection

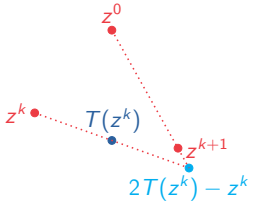
Halpern Scheme: Anchors the update to z^0 to prevent "drifting":

$$z^{k+1} = \frac{k+1}{k+2} T(z^k) + \frac{1}{k+2} z^0$$



Reflection Enhancement: If T is FNE, then $R = 2T - I$ is *non-expansive*.

$$z^{k+1} = \frac{k+1}{k+2}\Big(2T(z^k) - z^k\Big) + \frac{1}{k+2} z^0$$



Theoretical Breakthrough

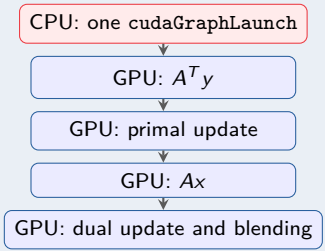
Reflected Halpern achieves $\|z^k - T(z^k)\|_P \leq \frac{1}{k+1}\mathrm{dist}_P(z^0, \mathcal{Z}^*)$. This provides a **2× speedup** and higher numerical stability.

CUDA optimization: two complementary directions

Takeaway

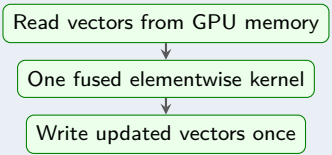
CUDA graphs reduce launch overhead; kernel fusion reduces memory traffic.

1. CUDA graphs: fewer launches



- As used in cuPDLPx: record the repeated kernel chain once.
- Replay it without paying CPU launch overhead for every kernel.

2. Kernel fusion: less memory traffic



- Combine compatible vector operations.
- Reduce global-memory round trips and intermediate writes.

HiPDLP: the newest member of the HiGHS family

HiPDLP is our new high-performance C++ & CUDA implementation, designed to be integrated directly into **HiGHS**

- **Zero-dependency C++:** high-speed execution without the overhead of heavy frameworks
- **Tight integration:** uses HiGHS's world-class **presolve**
- **CPU/GPU ready:** built to scale on modern multicore architectures

Numerical Performance Roadmap

Solver	Solved (414)	Avg Time (s)	Architecture
OR-Tools (PDLP)	293	442s	CPU (Multicore)
cuPDLP-C (GPU)	337	121s	Hybrid Sync
HiPDLP (Ours)	**Benchmarking**	**Goal: $<$100s**	**Full GPU Resident**

- **Strategy:** HiGHS industrial presolve + efficient C++ kernels + CUDA.
- We aim to eliminate the communication overhead that limits current state-of-the-art GPU solvers on massive million-scale instances.

Summary and research directions

- **First-order methods** make large LPs tractable using parallel matrix-vector products.
- **HiPDLP** combines HiGHS presolve and scaling with modern PDHG variants and GPU engineering.
- **GPU optimization** should attack both launch overhead and memory traffic.

Next research questions

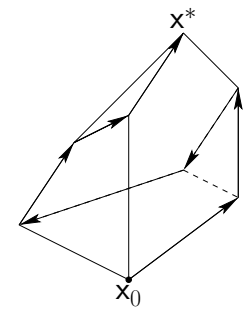
- How should we cross over efficiently to a basic solution?
- How can we detect infeasibility reliably and early?
- Any better first-order methods?

References and questions

Questions? yanyu.zhou@ed.ac.uk `github.com/ERGO-Code/HiGHS`

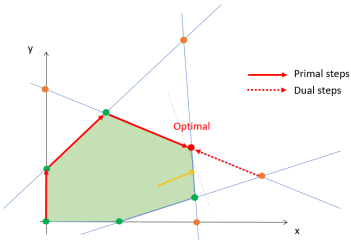
Three ways to solve an LP

Simplex: Vertex-to-vertex



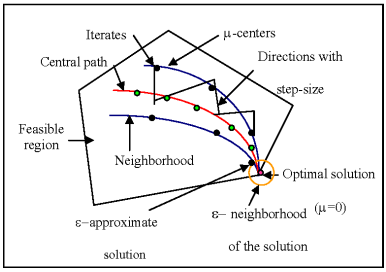
- Pro: Guaranteed to find optimal solution in finite steps
- Con: Difficult to parallelize

Dual simplex: Vertex-to-vertex



- Widely used in commercial solvers (CPLEX, Gurobi)

IPM: Newton steps (central path).



- Pro: Local quadratic convergence
- Con: Need matrix factorization