Solving large-scale LPs on modern GPUs

François Pacaud

CAS, Mines Paris - PSL

RTE R&D

March 23, 2026

Today's topic

Introducing one of the most famous optimization problem

Linear program (LP) in standard form

For given data $A \in \mathbb{R}^{m \times n}$, $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$, solve

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

Classical solution algorithms:

- Simplex algorithm
- Interior-point method
- ...and PDLP

Solving LPs on GPUs

```
for (int iy = 0; iy < 5; ++iy) {
    for (int ix = 0; ix < 20; ++ix) {
        for (int jy = 0; jy < 8; ++jy) {
            for (int jx = 0; jx < 16; ++jx) {
                foo(iy, ix, jy, jx);
            }
        }
    }
}
```

Figure: Programming on a CPU

```
__device__ void foo(int iy, int ix, int jy, int jx) {}

__global__ void mykernel() {
    int ix = blockIdx.x;
    int iy = blockIdx.y;
    int jx = threadIdx.x;
    int jy = threadIdx.y;
    foo(iy, ix, jy, jx);
}

int main() {
    dim3 dimGrid(20, 5);
    dim3 dimBlock(16, 8);
    mykernel<<<dimGrid, dimBlock>>>();
    cudaDeviceSynchronize();
}
```

Figure: Programming on a GPU

- CPUs have dozen of cores, while GPUs have thousand of them
- GPU are designed for single instruction multiple data (SIMD) programing patterns
- Many classical algorithms (including simplex) do not work well on GPUs!

Towards solving LPs in batch

Batch LPs

For a list of parameters $\{(c_i, b_i, A_i)\}_{i=1,\cdots,N}$ with similar sizes, solve

$$\begin{aligned} \min_{x_i \in \mathbb{R}^n} \quad & c_i^\top x_i \\ \text{subject to} \quad & A_i x_i = b_i \qquad \forall i = 1, \cdots, N \\ & x_i \geq 0 \end{aligned}$$

- The number N defines the batch sizes
- The solution of all LPs can proceed simultaneously
- The constraint matrices A_k should have the same sparsity pattern for efficient implementation

Outline

Primal-dual hybrid gradient (PDHG)

Using Lagrangian duality, the LP is equivalent to

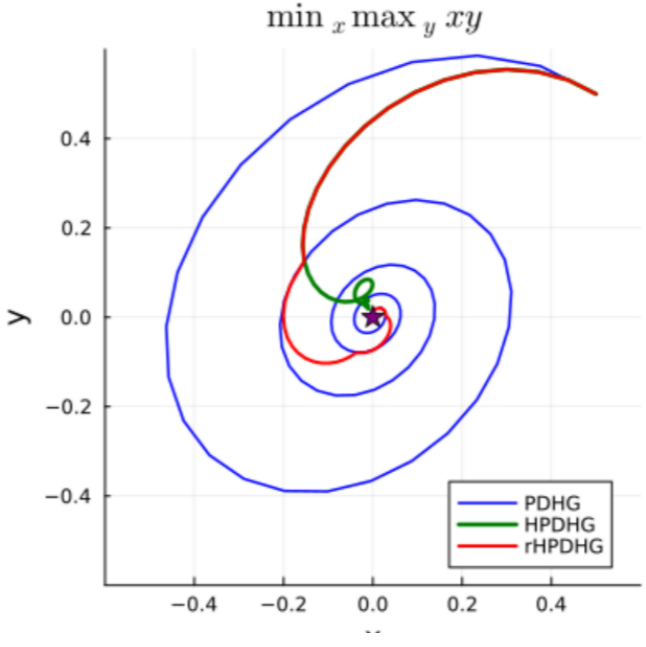
$$\min_{x \geq 0} \max_{y \in \mathbb{R}^m} \mathcal{L}(x, y) := c^\top x + y^\top (b - Ax)$$

PDHG ($\approx$ Arrow-Hurwicz)

Decouple the primal and dual problems using operator splitting:

$$x^{k+1} \in \arg\min_{x \geq 0} \; c^\top x + (y^k)^\top Ax + \frac{1}{2\eta}\|x - x^k\|^2$$

$$y^{k+1} \in \arg\min_{y \in \mathbb{R}^m} \; b^\top y - y^\top A(2x^{k+1} - x^k) + \frac{1}{2\eta}\|y - y^k\|^2$$



PDHG exhibits a well known spiralling behavior
(image from Haihao Lu, "GPU-Accelerated Linear Programming and Beyond")

PDLP: PDHG for LPs

"Engineered" variant of PDHG

As all first-order methods, PDHG's performance is impacted by many parameters

Enhancements

- Diagonal preconditioning *(scale the data beforehand)*

$$x = D_x \tilde{x} \ , \quad y = D_y \tilde{y}$$

- Specialized step-size η *(update η depending on the current residual)*
- Adaptive restarts *(avoid spiralling behavior in the algorithm)*
- Primal weight updates *(for scale invariance)*
- Presolve *(always import for LPs)*

- At the end, PDLP requires only matrix-vector operations Ax and $A^\top y$!
- Take many iterations to converge, but each iteration is itself super fast
- Easy to implement on the GPU

Modern implementations

A non-exhaustive list

PDLP is very easy to implement, leading to numerous different implementations of the same algorithm

Solvers implementing PDLP

- HiGHS
- Almost all commercial solvers (Xpress, Gurobi, Knitro, ...)
- cuOPT (NVIDIA)
- cuPDLP.jl/ cuPDLPx (reference academic implementation)
- CoolPDLP.jl
- MPAX (batch implementation in Jax)
- and many others...

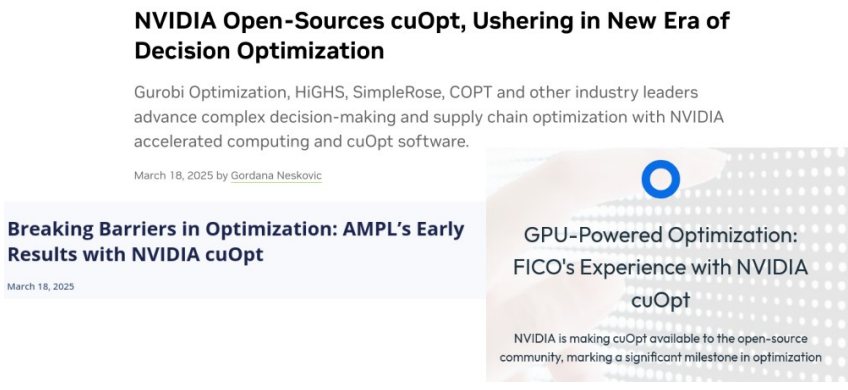


Figure: March 18, 2025

However, questions have been raised (for example by Rothberg in a recent Gurobi webinar) about the accuracy of the solutions obtained by PDLP. As a first order method, the linear convergence of PDLP means that finding high-accuracy solutions is relatively more expensive than when using an interior point method. PDLP also terminates when optimality criteria relative to the magnitudes of the costs and RHS values of the LP are satisfied. Hence, when these contain large values, it is possible for absolute optimality criteria to be far from satisfied. Ju-

Figure: HiGHS newsletter

Outline

Stationary conditions

KKT conditions

A variable x is solution if there exists multipliers (y, z) such that

$$c + A^\top y - z = 0$$
$$Ax - b = 0$$
$$0 \leq x \perp z \geq 0$$

where $0 \leq x \perp z \geq 0$ encodes the complementarity constraints

$$x_i \times z_i = 0 \quad \forall i = 1, \cdots, n$$

Primal-dual interior-point method

Central path

For a barrier parameter $\mu > 0$, we say that w is on the central path if $(x, z) > 0$ and

$$c + A^\top y - z = 0$$
$$Ax - b = 0$$
$$XZe = \mu e$$

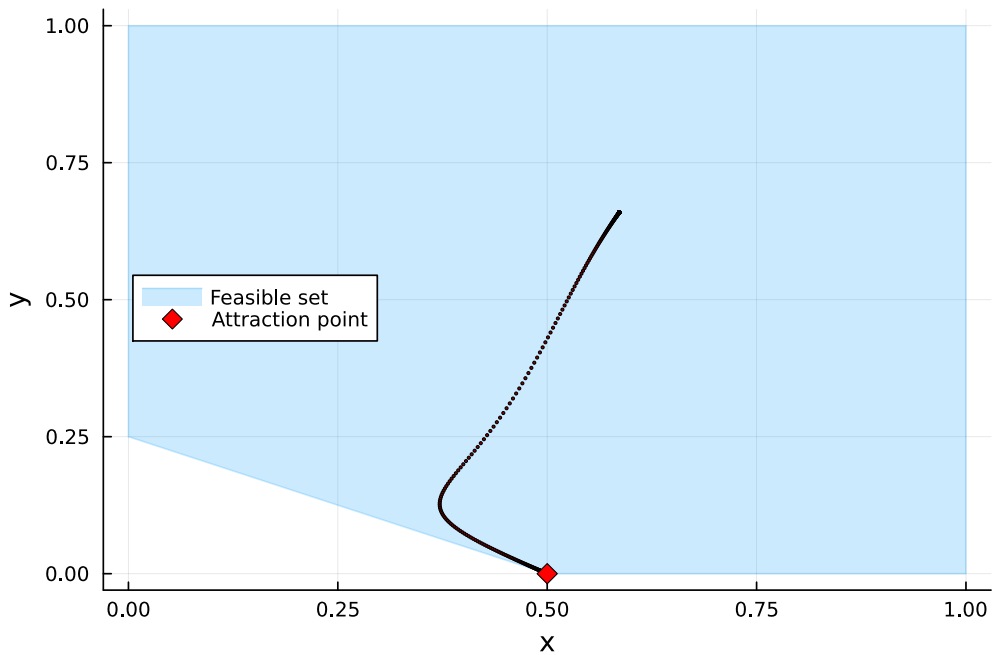


Figure: Primal-dual interior-point methods track the central path using the Newton method

Tracking the central path with the Mehrotra predictor-corrector

For a given primal-dual iterate $w = (x, y, z)$, define the **current barrier parameter** (average complementarity) as:

$$\mu = \frac{z^\top x}{n} .$$

Affine step: Compute Δ^{aff} by solving

$$\begin{bmatrix} 0 & A^\top & -I \\ A & 0 & 0 \\ Z & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x^{\text{aff}} \\ \Delta y^{\text{aff}} \\ \Delta z^{\text{aff}} \end{bmatrix} = - \begin{bmatrix} c + A^\top y - z \\ Ax - b \\ XZe \end{bmatrix} .$$

Corrector step: Compute Δ^{corr} by solving

$$\begin{bmatrix} 0 & A^\top & -I \\ A & 0 & 0 \\ Z & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x^{\text{corr}} \\ \Delta y^{\text{corr}} \\ \Delta z^{\text{corr}} \end{bmatrix} = - \begin{bmatrix} 0 \\ 0 \\ \sigma\mu e - \Delta Z^{\text{aff}} \Delta X^{\text{aff}} e \end{bmatrix} ,$$

with σ given by a heuristic

Update: For $\Delta^k = \Delta^{\text{aff}} + \Delta^{\text{corr}}$, set

$$w^{k+1} = w^k + \alpha^k \Delta^k$$

with α a step computed using a fraction-to-boundary rule

Mehrotra. On the implementation of a primal-dual interior point method. 1992.

Sparse linear solver

The affine step and the corrector step are both solving the **unsymmetric linear system**:

$$\begin{bmatrix} 0 & A^\top & -I \\ A & 0 & 0 \\ Z & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} .$$

Augmented KKT system

The unsymmetric system reduces to:

$$\begin{bmatrix} \Sigma & A^\top \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} r_1 + X^{-1} r_3 \\ r_2 \end{bmatrix} ,$$

with the diagonal matrix $\Sigma := X^{-1} Z$.

Normal KKT system

We can also eliminate Δy to recover the positive-definite system:

$$\left(A \Sigma^{-1} A^\top \right) \Delta y = A \Sigma^{-1} \left(r_1 + X^{-1} r_3 \right) - r_2 .$$

How to solve efficiently a linear system $Kx = b$?

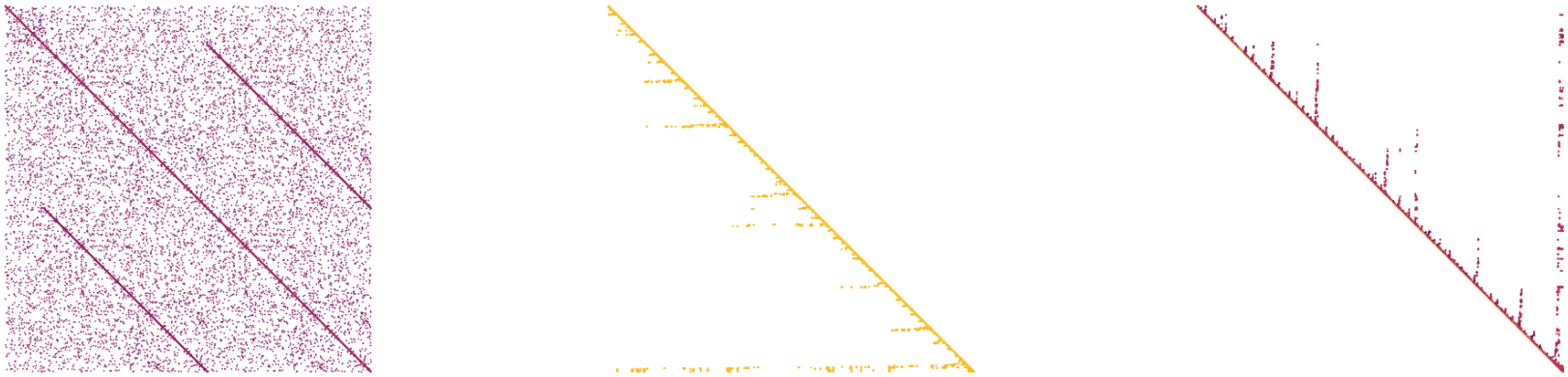


Figure: LU decomposition of the power flow Jacobian for 1354pegase (computed using KLU)

Workflow

1. *Symbolic factorization:* Analyse the matrix sparsity pattern and find a static ordering that reduces the fill-in. Instantiate the sparse factors in memory
2. *Numerical factorization:* Compute inplace the nonzero values in L and U
3. *Backsolve:* Solve the system $L^{-1}b$ and $U^{-1}d$ to get solution of the linear system

Matrix decompositions: a hall of fame

A is generic

LU decomposition:

$$K = PLUQ$$

with L lower triangular and U upper triangular matrices

A is symmetric definite positive (SDP)

Cholesky decomposition:

$$K = PLL^\top P^\top$$

with L lower triangular

A is symmetric indefinite

Bunch-Kaufman factorization

$$K = PQLBL^\top Q^\top P^\top$$

with L lower triangular and B block diagonal matrix (with blocks of size 1×1 or 2×2)

A is symmetric quasi-definite (SQD)

Signed-Cholesky factorization

$$K = PLDL^\top P^\top$$

Solving sparse linear system on the GPU

Observation

- Numerical pivoting is difficult to parallelize
  (and should be avoided at all cost on the GPU)
- Use Cholesky or signed-Cholesky whenever you can!

NVIDIA cuDSS

Released in November 2023

- Symbolic factorization on the CPU, the rest on the GPU
- Implement sparse LU, Cholesky and signed-Cholesky
- Support batch solution of linear systems!

GPU-accelerated solvers

Solvers

Currently, only a few solvers support GPU acceleration with cuDSS:

- Gurobi (experimental support)
- Moreau
- cuClarabel
- MadIPM!

Next experiments are done using MadIPM

Numerical results on DC-OPF problems

Algorithm	HiGHS Dual simplex	MadIPM (CPU) IPM	MadIPM (GPU) IPM
9241pegase	1.4	1.3	0.7
13659pegase	2.0	1.2	0.8
19402goc	32.2	1.4	1.1
30000goc	8.0	3.1	2.3
78484epigrids	193.0	8.2	4.9

Table: Total running time (in seconds). Instances from the PGLIB-OPF benchmark.

Observation

These instances are too small to observe a real benefit from GPU-acceleration

Practical results on large-scale LPs from ANTARES

Algorithm	HiGHS Dual simplex	MadIPM (CPU) IPM	MadIPM (GPU) IPM
study-1-4-10	3.2	85.2	2.0
study-2-5-10	53.2	359.0	7.6

Table: Total running time (in seconds). Instances from ANTARES.

Observation

On the largest instance, 7x speed-up compared to HiGHS

Solving DC-OPF LPs in batch

Recent results obtained by Michael Klamike @ Georgia Tech

batch size N	MadIPM (CPU)	MadIPM (GPU)	speedup
1	0.10	0.16	0.6x
2	0.20	0.16	1.2x
4	0.41	0.17	2.5x
8	0.81	0.17	4.7x
16	1.63	0.20	8.1x
32	3.31	0.24	13.8x
64	7.20	0.32	22.2x
128	15.05	0.49	30.5x

Table: Solve N LPs in parallel on the CPU (using multithreading) and on the GPU. Instance is 1354pegase, from the PGLIB-OPF benchmark

Observation

GPUs are very good at solving LPs in batch!

Outline

Concurrent Crossover for PDHG

Edward Rothberg
Gurobi Optimization, LLC
rothberg@gurobi.com

October 28, 2025

Figure: This is a hot topic!

Research question

Oftentimes, we want the exact active set at the LP solution

- PDLP: known at $\approx 10^{-4}$
- IPM: known at $\approx 10^{-8}$

Towards a GPU-friendly crossover method?

Differential optimization & Sensitivity analysis

Suppose we have a solution x for the LP

$$\min_{x \in \mathbb{R}^n} \; c^\top x \text{ subject to } Ax = b \;,\; x \geq 0$$

Research question

Can we deduce the solution $x + \Delta x$ of the perturbed LP?

$$\min_{x \in \mathbb{R}^n} \; (c + \Delta c)^\top x \text{ subject to } (A + \Delta A)x = b + \Delta b \;,\; x \geq 0$$

for $\Delta c \in \mathbb{R}^n$, $\Delta b \in \mathbb{R}^m$, $\Delta A \in \mathbb{R}^{m \times n}$ small enough

Use-cases:

- Warm-start!
- Optimization layers in machine learning
- Contextual learning in stochastic optimization

Summary of the current situation

Take-away

- GPU-accelerated optimization is gaining momentum
- Solving LPs in batch is a promising method

Caveats

- Mostly research-driven
- Not adopted by the major players (at the exception of NVIDIA)
- Impossible trade-off: expensive GPU or expensive license?