GPU-accelerated Optimization: Part I

Yang, Hongzhang

Shenzhen Research Institute of Big Data

9/11/2025

Overview

- Mathematical programming has long relied on CPU-based methods.
- Even top CPUs struggle with LPs of billions of variables—a scale that starkly contrasts with the immense size routinely achieved by modern deep learning models.
- This gap prompts the key question: Can GPUs improve the scalability and efficiency of optimization?

Overview

- We classify GPU-accelerated optimization research by three axes: (i) General-purpose vs. Problem-specific, (ii) LP vs. MIP, and (iii) Exact vs. Heuristic.
- Today we focus only on the General-purpose quadrant, yielding four classes that pair {LP, MIP} with {Exact, Heuristic}:

	LP	**MIP**
Exact	cuPDLP, HRP, etc.	Currently blank
Heuristic	Not necessary	cuOpt

LP-Exact: Introduction

Whereas CPUs are optimized for low-latency execution of complex, sequential tasks, GPUs are optimized for high-throughput execution of simple, parallel tasks:

	GPU	**CPU**
Core Design	thousands of lightweight cores	fewer, more powerful cores
Control and Execution	shared control units, batch processing	dedicated control units, task switching
Memory Design	huge throughput	low latency
Processing Model	thousands of simple threads	fewer, more complex threads

Can GPUs be leveraged to accelerate and scale up linear programming?

A natural starting point is to use GPUs to accelerate key computational components of linear programming solvers, namely, the basic linear algebra subroutines (BLAS):

	First-Order	**Interior-Point**	**ADMM**
BLAS	SpMV	Cholesky Factorization	LDL^T Factorization

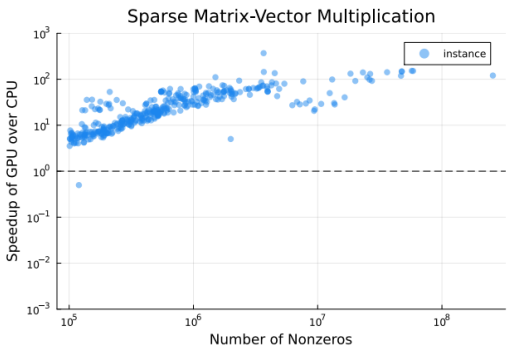
Recent developments in GPU sparse linear algebra libraries, such as cuSPARSE and cuDSS by NVIDIA, have enabled GPU acceleration for all three operations.

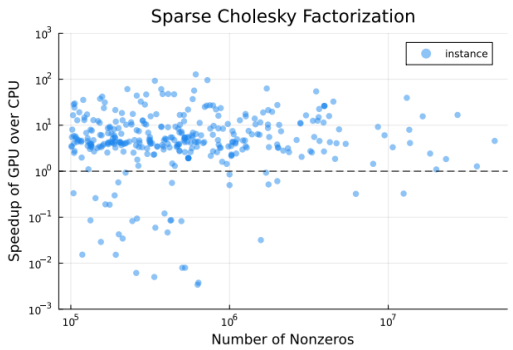
BLAS

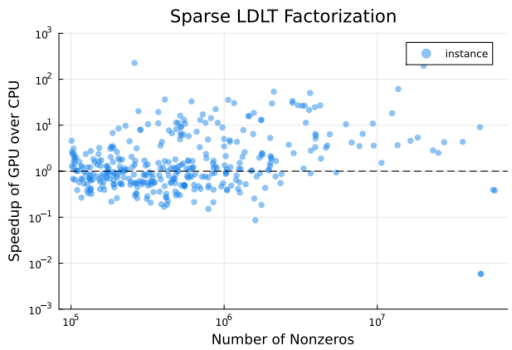
- SpMV computes Ax.
- Cholesky Factorization computes $LL^T = AA^T$.
- LDL$^\top$ Factorization computes $LDL^T = \begin{pmatrix} I & -A^\top \\ -A & 0 \end{pmatrix}$.
- A is the constraint matrix and L is a lower triangular matrix.

LP-Exact: Introduction

The speedup in running time of GPU versus CPU for SpMV, Cholesky Factorization and LDL^T Factorization with data of 383 MIPLIB instances:







- GPU speedup for SpMV scales linearly with problem size.
- The speedup for Cholesky Factorization depends heavily on the structure of the problem instance.
- LDL^T Factorization on average shows little to no clear advantage on GPUs.

LP-Exact: Introduction

Geometric mean of additional memory usage of GPU-based SpMV, Cholesky and LDL^T Factorization on 383 MIPLIB instances:

	SpMV	Cholesky Factorization	$LDL^\top$ Factorization	Instances
Allocated Memory	0.971 MB	481.067 MB	243.486 MB	4.915 MB

- GPUs onboard memory is typically an order of magnitude smaller than that of high-end CPUs.
- SpMV imposes negligible overhead.
- Both factorization methods demand substantially more memory.

LP-Exact: Introduction

To fully leverage the computational potential of GPUs, such methods should satisfy two key criteria:

- The majority of computational time should be concentrated in components that are highly parallelizable.
- Due to limited device memory of GPUs, computational primitives with cheaper memory cost are favorable.

First-order methods (FOMs), which can often purely base on SpMV, emerge as a promising alternative.

Early approaches to FOMs for solving LP:

Feasible Direction Methods

- Move along feasible descent directions for "big jumps" instead of crawling along edges.
- Requires solving linear systems for direction updates.

Projected Gradient Algorithms

- Use gradient projection to ensure iterates stay within feasible regions.
- Projections onto polyhedral constraints are computationally expensive.

LP-Exact: FOMs

Recent approaches to FOMs for solving LP:

cuPDLP

- GPU-accelerated primal-dual hybrid gradient (PDHG) with restarts and GPU-specific heuristics.

HPR-LP

- Uses Halpern Peaceman-Rachford (HRP) method with semi-proximal terms and adaptive techniques.
- Theoretical convergence guarantees are complex and tuning parameters affects performance.

ADMM + Conjugate gradient

- Combines ADMM with CG for solving linear systems via matrix-vector multiplications.
- Each ADMM iteration needs many CG steps.

PDLP takes the following general LP form as input:

$$\begin{aligned} \min_{x} \quad & c^T x \\ \text{s.t.} \quad & Ax = b,\ Gx \geq h, \\ & l \leq x \leq u, \end{aligned} \tag{1}$$

where $G \in \mathbb{R}^{m_1 \times n}$, $A \in \mathbb{R}^{m_2 \times n}$, $c \in \mathbb{R}^n$, $h \in \mathbb{R}^{m_1}$, $b \in \mathbb{R}^{m_2}$, $l \in (\mathbb{R} \cup \{-\infty\})^n$, $u \in (\mathbb{R} \cup \{+\infty\})^n$.

PDLP solves this LP by addressing its primal-dual form:

$$\min_{x \in X} \max_{y \in Y}\ L(x, y) := c^T x - y^T K x + q^T y, \tag{2}$$

where $K^T = (G^T, A^T)$ and $q^T := (h^T, b^T)$,
$X := \{x \in \mathbb{R}^n : l \leq x \leq u\}$, and $Y := \{y \in \mathbb{R}^{m_1 + m_2} : y_{1:m_1} \geq 0\}$.

PDLP is an enhanced Primal-Dual Hybrid Gradient (PDHG) method for LP, defined by the following update rule:

$$\begin{aligned} x^{k+1} &\leftarrow \operatorname{proj}_X\left(x^k - \tau(c - K^T y^k)\right), \\ y^{k+1} &\leftarrow \operatorname{proj}_Y\left(y^k + \sigma(q - K(2x^{k+1} - x^k))\right), \end{aligned} \tag{3}$$

where τ is the primal step-size and σ is the dual step-size. The primal and the dual step-sizes are reparameterized in PDLP as

$$\tau = \eta/\omega, \quad \sigma = \eta\omega, \quad \text{with } \eta, \omega > 0.$$

LP-Exact: PDLP

Algorithm 1: PDLP (after preconditioning and presolve)

Input: An initial solution $z^{0,0}$;

Initialize outer loop counter $n \leftarrow 0$, total iterations $k \leftarrow 0$, step size $\hat{\eta}^{0,0} \leftarrow 1/\|K\|_\infty$, primal weight $\omega^0 \leftarrow \texttt{InitializePrimalWeight}(c, q)$;

repeat

  $t \leftarrow 0$;

  repeat

    $z^{n,t+1}, \eta^{n,t+1}, \hat{\eta}^{n,t+1} \leftarrow \texttt{AdaptiveStepOfPDHG}(z^{n,t}, \omega^n, \hat{\eta}^{n,t}, k)$;

    $\bar{z}^{n,t+1} \leftarrow \frac{1}{\sum_{i=1}^{t+1} \eta^{n,i}} \sum_{i=1}^{t+1} \eta^{n,i} z^{n,i}$;

    $z_{\mathrm{c}}^{n,t+1} \leftarrow \texttt{GetRestartCandidate}(z^{n,t+1}, \bar{z}^{n,t+1}, z^{n,0})$;

    $t \leftarrow t+1, k \leftarrow k+1$;

  until *restart or termination criteria holds*;

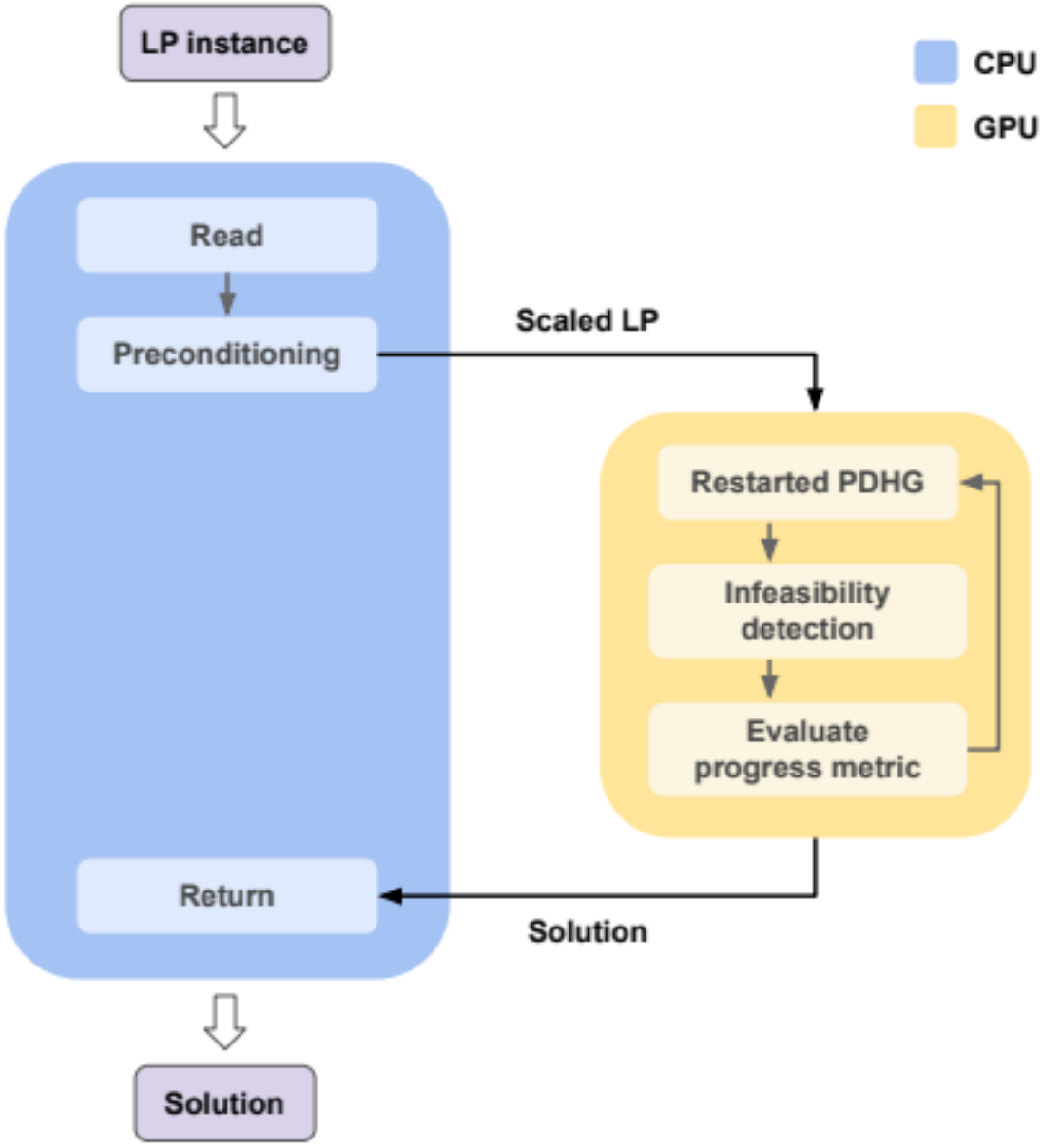
  restart the outer loop. $z^{n+1,0} \leftarrow z_{\mathrm{c}}^{n,t}, n \leftarrow n+1$;

  $\omega^n \leftarrow \texttt{PrimalWeightUpdate}(z^{n,0}, z^{n-1,0}, \omega^{n-1})$;

until *termination criteria holds*;

Output: $z^{n,0}$.

LP-Exact: PDLP



The algorithm as a whole has no convergence guarantee, although some individual enhancements do:

Key Enhancements

- **Platform**.
- **Input form**.
- **Update rule**.
- Step size choice.
- **Restart scheme**.
- Primal weight update.
- Preconditioning and Presolve.
- Feasibility polishing.
- Infeasibility detection.

FirstOrderLp.jl in 2021

- First version of PDLP introduced as a CPU-based Julia package.
- Research-oriented implementation, foundational for algorithmic development.

Google OR-Tools in 2022

- Multi-threading CPU-based C++ implementation open-sourced.
- Improved CPU parallelism, adopted different standard form of LP compared to original.

LP-Exact: PDLP

cuPDLP.jl in Nov 2023

- GPU-optimized variant released.
- First major GPU acceleration, introduced alternative restarting scheme for better GPU performance.

cuPDLP-C in Dec 2023

- COPT open-sourced a GPU-based C re-implementation.
- Lower-level, high-performance backend enabling broader integration.

MPAX in Dec 2024

- A JAX-based implementation, introduced.
- Implemented reflected Halpern variant for improved convergence, supported auto-differentiation, batch solving, and multi-GPU training in deep learning pipelines.

Production-grade Deployment

- COPT integrated cuPDLP-C into its commercial solver in Feb 2024.
- HiGHS and NVIDIA cuOpt incorporated cuPDLP-C in Mar 2024.
- FICO Xpress integrated PDLP into its solver, Gurobi announced future integration in Apr 2024.
- NVIDIA announced open-sourcing of its PDLP implementation in Mar 2025.

Three Milestone PDLP Solvers

- **FirstOrderLp.jl** (2021): Built upon the PDHG framework with multiple algorithmic **enhancements**.
- **cuPDLP.jl** (2023): Introduced a GPU-tailored **restart scheme**.
- **r^{2}HPDHG** (2024): Adopted a reflected Halpern-type iteration as its core **update rule**.

Numerical Performances

- cuPDLP.jl and r2HPDHG with three methods implemented in Gurobi, i.e., primal simplex method, dual simplex method, and interior-point method.
- cuPDLP.jl and r2HPDHG with FirstOrderLp.jl.

Experiment Setup

A total of 383 instances from MIPLIB 2017 are selected based on specific criteria:

	Small	**Medium**	**Large**
Number of nonzeros	100K - 1M	1M - 10M	>10M
Number of instances	269	94	20

Comparison with Gurobi

	Small (269) (1-hour limit)		Medium (94) (1-hour limit)		Large (20) (5-hour limit)		Total (383)	
	Count	**Time**	**Count**	**Time**	**Count**	**Time**	**Count**	**Time**
cuPDLP.jl	266	8.61	92	14.80	19	111.19	377	12.02
r²HPDHG	267	6.61	93	7.84	19	90.81	379	8.58
Primal simplex (Gurobi)	268	12.56	69	188.81	11	3145.49	348	39.81
Dual simplex (Gurobi)	268	8.75	84	66.67	15	591.63	367	21.75
Barrier (Gurobi)	268	5.30	88	45.01	18	415.78	374	14.92

Table 5: Solve time in seconds and SGM10 of different solvers on instances of `MIP Relaxations` with tolerance 10^{-4}: cuPDLP.jl/r²HPDHG versus Gurobi without presolve.

	Small (269) (1-hour limit)		Medium (94) (1-hour limit)		Large (20) (5-hour limit)		Total (383)	
	Count	**Time**	**Count**	**Time**	**Count**	**Time**	**Count**	**Time**
cuPDLP.jl	261	23.47	86	40.69	16	421.40	363	32.35
r²HPDHG	260	19.13	87	28.35	16	229.47	363	24.79
Primal simplex (Gurobi)	268	12.43	74	157.59	13	2180.23	355	36.68
Dual simplex (Gurobi)	268	8.00	83	59.93	15	687.17	366	20.40
Barrier (Gurobi)	267	6.24	88	48.62	18	438.69	373	16.46

Table 6: Solve time in seconds and SGM10 of different solvers on instances of `MIP Relaxations` with tolerance 10^{-8}: cuPDLP.jl/r²HPDHG versus Gurobi without presolve.[11]

Comparison with Gurobi

	Small (269) (1-hour limit)		Medium (94) (1-hour limit)		Large (20) (5-hour limit)		Total (383)	
	Count	**Time**	**Count**	**Time**	**Count**	**Time**	**Count**	**Time**
cuPDLP.jl	269	5.35	93	10.31	19	33.93	381	7.37
r²HPDHG	267	3.95	94	6.45	19	17.13	380	5.04
Primal simplex (Gurobi)	269	5.67	71	121.23	19	297.59	359	20.84
Dual simplex (Gurobi)	268	4.17	86	37.56	19	179.49	373	11.84
Barrier (Gurobi)	269	1.21	94	15.32	20	30.70	383	4.65

Table 7: Solve time in seconds and SGM10 of different solvers on instances of `MIP Relaxations` with tolerance 10^{-4}: cuPDLP.jl/r²HPDHG versus Gurobi with presolve.

	Small (269) (1-hour limit)		Medium (94) (1-hour limit)		Large (20) (5-hour limit)		Total (383)	
	Count	**Time**	**Count**	**Time**	**Count**	**Time**	**Count**	**Time**
cuPDLP.jl	264	17.53	90	30.05	19	81.07	373	22.13
r²HPDHG	261	15.24	90	21.67	19	56.19	370	18.07
Primal simplex (Gurobi)	269	5.19	75	100.03	18	171.72	362	18.11
Dual simplex (Gurobi)	268	3.53	89	27.17	19	121.94	376	9.53
Barrier (Gurobi)	269	1.34	94	16.85	20	33.48	383	5.03

Table 8: Solve time in seconds and SGM10 of different solvers on instances of `MIP Relaxations` with tolerance 10^{-8}: cuPDLP.jl/r²HPDHG versus Gurobi with presolve.

Comparison with Gurobi

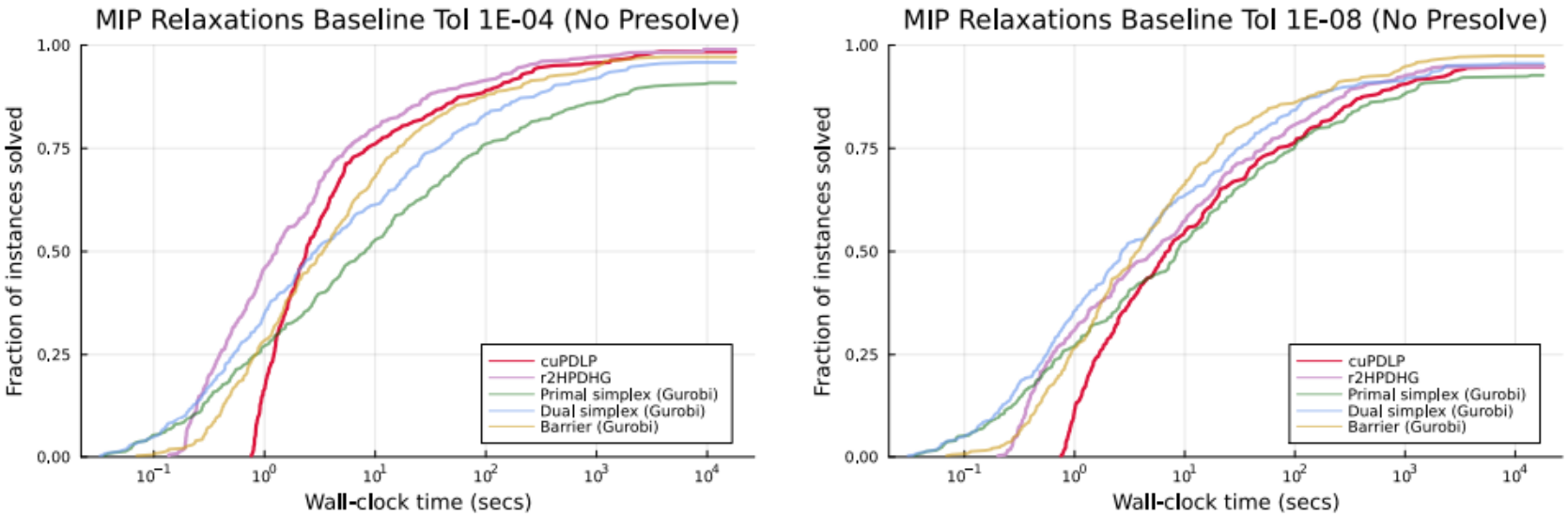


Figure 4: Number of instances solved for `MIP Relaxations` under moderate accuracy (left) and high accuracy (right): cuPDLP.jl versus Gurobi without presolve.

Comparison with Gurobi

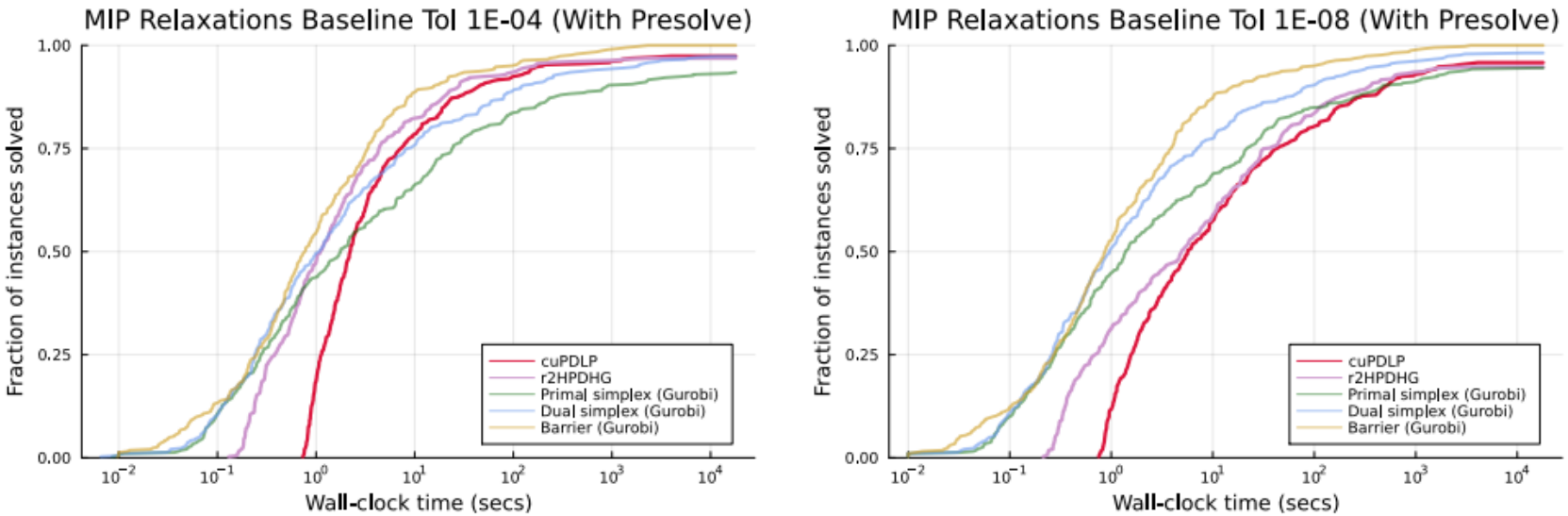


Figure 5: Number of instances solved for `MIP Relaxations` under moderate accuracy (left) and high accuracy (right): cuPDLP.jl versus Gurobi with presolve.

Comparison with FirsOrderLp.jl

	Small (269) (1-hour limit)		Medium (94) (1-hour limit)		Large (20) (5-hour limit)		Total (383)	
	Count	**Time**	**Count**	**Time**	**Count**	**Time**	**Count**	**Time**
cuPDLP.jl	266	8.61	92	14.80	19	111.19	377	12.02
r²HPDHG	267	6.61	93	7.84	19	90.81	379	8.58
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

Table 9: Solve time in seconds and SGM10 of different solvers on instances of `MIP Relaxations` with tolerance 10^{-4}: cuPDLP.jl/r²HPDHG versus PDLP.

	Small (269) (1-hour limit)		Medium (94) (1-hour limit)		Large (20) (5-hour limit)		Total (383)	
	Count	**Time**	**Count**	**Time**	**Count**	**Time**	**Count**	**Time**
cuPDLP.jl	261	23.47	86	40.69	16	421.40	363	32.35
r²HPDHG	260	19.13	87	28.35	16	229.47	363	24.79
FirstOrderLp.jl	235	91.14	68	389.34	9	3552.50	312	160.63
PDLP (1 thread)	250	49.31	73	259.04	12	3818.42	335	96.86
PDLP (4 threads)	245	54.19	81	136.16	14	1789.54	340	83.49
PDLP (16 threads)	214	248.34	69	403.17	14	2475.57	297	316.27

Table 10: Solve time in seconds and SGM10 of different solvers on instances of `MIP Relaxations` with tolerance 10^{-8}: cuPDLP.jl/r²HPDHG versus PDLP.

Comparison with FirsOrderLp.jl

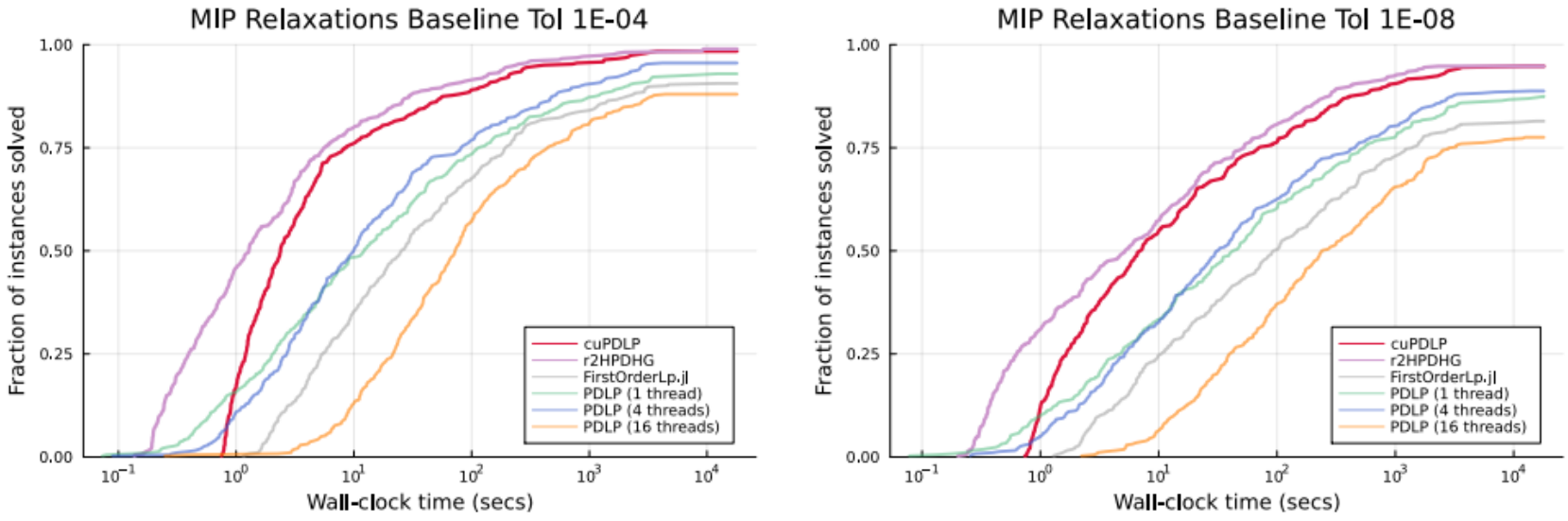


Figure 6: Number of instances solved for `MIP Relaxations` under moderate accuracy (left) and high accuracy (right): cuPDLP.jl versus PDLP.