Presolving for GPU-Accelerated First-Order LP Solvers

Daniel Cederberg Stephen Boyd

Stanford University

April 25, 2026

Abstract

Recent research has focused on developing GPU-accelerated first-order solvers for linear programming (LP). This line of work, however, has largely overlooked the role of presolving, and thus prior results do not fully reflect the speedups achievable through GPU acceleration in a realistic end-to-end solver pipeline. At the same time, LP presolving has traditionally been developed for CPU-based solvers, where presolve time rarely dominates the total runtime and the emphasis has been on maximizing the reduction in problem size, even at the expense of costly presolve rules. Given the high performance of modern GPU-accelerated solvers and the inherently sequential nature of presolving, it is unclear whether this traditional approach to presolving remains appropriate.

In this paper we revisit LP presolving from the perspective of GPU-accelerated first-order LP solvers. We identify a set of relatively simple presolve rules and show that a carefully engineered collection of these captures most of the reduction achieved by Gurobi's commercial state-of-the-art presolver, at a fraction of the cost. Moreover, we demonstrate that such lightweight presolving can yield substantial end-to-end speedups for the GPU-accelerated solver cuPDLPx, despite presolving sometimes constituting a significant fraction of the total runtime. These results suggest that lightweight presolving may remain beneficial as GPU performance continues to scale, while the sequential nature of presolving presumably does not.

We accompany this paper with an open-source C implementation of an LP presolver, called PSLP (Presolver for Linear Programs). PSLP is battle-tested and has been adopted by the community, with integrations in cuPDLPx, cuOpt (NVIDIA's optimization library), and HPR-LP.

Contents

1 Introduction **3**

1.1 Related work 5

2 Background **6**

2.1 Reductions 6

2.2 Primal exploration 8

2.3 Dual exploration 8

3 Design and implementation **9**

3.1 Explorers 9

3.2 Data structures 10

3.3 Our implementation 12

4 Experiments **12**

4.1 Presolve performance 13

4.2 End-to-end solver performance: experimental setup 16

4.3 End-to-end solver performance: results 16

5 Discussion **19**

1 Introduction

Linear programming (LP) is arguably the most widely used problem class in mathematical optimization, with applications spanning a broad range of fields. Decades of research and substantial engineering effort have produced highly optimized implementations of the simplex method and its many variants [Dan51, Mar02], as well as interior-point methods [RTV05]. As a result, LP solving has become a mature and remarkably efficient technology — capable of handling instances of a scale and complexity that George Dantzig himself could hardly have imagined, often in a matter of seconds.

Commercial LP solvers consist of three major components: (1) a *presolver* that reduces the size of the problem and later reconstructs a solution to the original problem [AA95]; (2) a *core step* that runs one or multiple algorithms concurrently to solve the reduced problem; and (3) a *crossover mechanism* (used only for non-simplex algorithms) that takes the approximate solution produced by the core step and converts it into a highly accurate basic feasible solution [BS94]. While the first and third components are highly sequential in nature (and can take a significant portion of the total solve time), the core step traditionally implements an interior-point method that benefits from CPU multithreading for the computational bottleneck of factorizing and solving large sparse linear systems. However, despite being compute-bound, this bottleneck appears to have an inherent limit to the amount of parallelism that can be effectively exploited (see, for example, [Rot20, Pan22]).

To fully exploit the computational power of modern hardware accelerators in the context of linear programming, recent research has focused on speeding up the core step by developing first-order methods that rely heavily on matrix-vector multiplications [ADH+21, ADH+25, CSY+25, LYH+23, LY25]. While this paradigm shift opens up exciting opportunities for solving extremely large LPs, it also raises important questions about the role of the other traditional components of LP solvers. In this paper we focus on presolving, a critical but often overlooked component, and its impact on GPU-accelerated first-order solvers.

Presolving is traditionally performed on CPUs because it involves a large number of sequential, logic-driven, and structurally irregular problem transformations. The application of one problem transformation often enables or invalidates others, creating a chain of data-dependent transformations that are inherently sequential. Many of these transformations require heavy branching, dynamic sparse-matrix modifications, and unpredictable memory access patterns. It is therefore currently believed that CPUs, with their low-latency caches and sophisticated control-flow capabilities, are far better suited for the task than GPUs. Although certain parts of a presolver admit some degree of parallelism, the engineering effort required to map these onto GPU architectures is rarely justified, and we speculate that the potential speedup would at best be modest, in particular if CPU multithreading is used for the parallelizable parts. However, GPU acceleration has a different consequence for presolving: the relative cost of presolving compared to the GPU-accelerated core step has increased significantly, putting very high demands on the implementation and speed of the presolver. For every extra second spent in presolving, there are thousands of idle GPU cores waiting to get to work, so it is unclear whether LP presolving is still worthwhile in this new context.

The idea of reducing the size of an LP before passing it to a solver has a long history, dating back at least to the 1970s. (We review related work in §1.1.) Over the past decades, a wide range of presolve techniques have been proposed, from simple ones such as eliminating constraints with just one variable [BMW75] to more elaborate procedures such as exploiting symmetry between variables [ABG+19]. This body of work has largely been developed for CPU-based solvers, where presolving and the core algorithm run on the same hardware, and where presolve time rarely dominates the overall runtime. In this traditional setting, the emphasis has been on maximizing the reduction in problem size, even at the cost of more costly presolve techniques.

In this paper we revisit LP presolving from a different perspective, motivated by GPU-accelerated first-order methods. In this setting, presolving is inherently sequential, runs on different hardware than the core algorithm, and can easily become a bottleneck. This raises a fundamental question: can meaningful performance gains be obtained using only a carefully engineered collection of simple and fast presolve techniques, and how much problem reduction is sacrificed by forgoing the more complex and costly rules found in traditional presolvers? By focusing on rules that are fast and lightweight, we aim to identify those that have the potential to remain effective as GPU performance continues to scale, while the sequential nature of presolving presumably does not.

Our experiments show, perhaps surprisingly, that a carefully engineered collection of relatively simple presolve techniques can achieve reductions in problem size comparable to those obtained by the commercial state-of-the-art presolver implemented in Gurobi [ABG+19]. Specifically, we find that our open-source C implementation of an LP presolver, called PSLP (Presolver for Linear Programs), on average captures about 90% of the reduction achieved by Gurobi's presolver, while being over 6 and 11 times faster, respectively, on two standard LP datasets. We also investigate the impact of PSLP on the total solve time of cuPDLPx [LPY25], a GPU-accelerated first-order LP solver based on the primal-dual hybrid gradient (PDHG) method [ZC08, CP11]. Our main observation is that PSLP has a strong positive impact on the total solve time of cuPDLPx across a wide range of LP instances. For example, on the large problems with more than 10^7 nonzero entries from Mittelmann's LP collection [Mit25] and the MIPLIB 2017 root-node LP relaxations [GHG+21], enabling PSLP reduces the shifted geometric mean of the total solve time by more than a factor of 2.5 and 9, respectively. Since PSLP was released, it has been integrated into several solvers by their developers, including cuPDLPx, cuOpt (NVIDIA's GPU-accelerated optimization library [NVI25]), and HPR-LP [CSY+25].

The remainder of this paper begins with an overview of related work on presolving. Next, we review the main ideas behind LP presolving in §2. Our presentation emphasizes the abstractions of *reductions*, *primal exploration*, and *dual exploration*, which we believe are useful for understanding and implementing presolvers, although the classic literature never presents presolving in this way. In §3, we discuss some of the design choices behind our open-source implementation of a fast and lightweight LP presolver. In §4, we assess how much of the total reduction achieved by the commercial presolver implemented in Gurobi can be captured by the carefully engineered collection of relatively simple presolve techniques

implemented in PSLP. We also evaluate the impact of presolving on the total solve time of cuPDLPx. Finally, we discuss our findings in §5.

1.1 Related work

The idea of reducing the size of an optimization problem before passing it to a solver has a long history. Early references describing many of the transformations implemented in modern presolvers are [BMW75, Wil83, BBG83, TW86]. These references focus primarily on linear programming where the simplex method is used as the algorithm for the core step. The development of interior-point methods in the 90's sparked renewed interest in presolving, as evidenced by a string of papers investigating the importance of presolving for interior-point methods for LPs [AA95, And95, Gon97]. An efficient LP presolver forms the foundation of presolvers for more general problem classes, including quadratic programs [MS03, GT04], mixed-integer linear programming (MIP) [GKM+15, ABG+19], and nonlinear programming [ZS25].

Despite the well-established role of presolving in commercial optimization software, there are surprisingly few open-source presolvers of any kind. One notable exception is PaPILO [GGH23], a solver-independent presolver for MIP developed by Leona Gottwald with significant contributions from Ivet Galabova, Alexander Hoen, Ambros Gleixner, and several others (see the PaPILO GitHub page for the full list of contributors). The scarcity of open-source presolvers is likely due to the substantial engineering effort required to implement them: presolvers involve numerous subtleties and corner cases that make them far more complex to implement than their high-level descriptions suggest. Moreover, a slow and poorly implemented presolver can do more harm than good and will not be used. In addition, there is relatively little academic incentive to develop presolvers, since presolvers are often viewed as engineering components rather than novel research contributions. This lack of incentive is reflected in the number of modern open-source solvers developed in academia that ship without a presolver, including Clarabel [GC24], SCS [OCPB16], OSQP [SBG+20], ECOS [DCB13], cuPDLP [LYH+23, LY25], cuPDLPx [LPY25] (that now ships with PSLP), and several others. Notable exceptions are HiGHS [HH18, Gal23], CBC [FLH05], and Google's linear optimization package [PF25], which all include solver-dependent presolve functionality for LPs.

Finally, we mention that while we view presolving as a task that runs on a CPU, there has been work on accelerating the so-called *primal propagation* or *bound tightening* step of a presolver using a GPU [SGP22]. While the results in that paper are promising given the setup and performance metrics they use, it is unclear how their results would translate to a more realistic setting for LP presolving for the following two reasons. First, they run up to 100 rounds of primal propagation, which is far more than what is typically done in an LP presolver in practice. (For example, Gurobi's presolver [ABG+19, §3.2] and PSLP both propagate each constraint only *once* per round, and typically run only for a few rounds. Moreover, PSLP keeps track of which constraints need to be re-examined after each round, so in every round only a subset of the constraints are actually propagated.) Second, primal propagation is only one of many components of a presolver. For example, in our experiments

on Mittelmann's LP collection [Mit25], primal propagation on average accounts for 12.0% of the total time spent in PSLP.

2 Background

We consider an LP of the form

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & \underline{b} \leq Ax \leq \overline{b} \\ & \underline{x} \leq x \leq \overline{x}, \end{array} \tag{1}$$

with variable $x \in \mathbf{R}^n$. The problem data are the cost vector $c \in \mathbf{R}^n$, the left and right hand sides $\underline{b} \in \mathbf{R}^m$ and $\overline{b} \in \mathbf{R}^m$, the constraint matrix $A \in \mathbf{R}^{m \times n}$, and the variable bounds $\underline{x} \in \mathbf{R}^n$ and $\overline{x} \in \mathbf{R}^n$. Infinite values are allowed in both the variable bounds and the left and right hand sides, allowing for unbounded variables and one-sided inequality constraints. Equality constraints can be expressed by setting the corresponding components of $\underline{b}$ and $\overline{b}$ equal.

A solution of (1) consists of a primal solution $x^\star$ and a dual solution $(y^\star, z^\star)$ satisfying the optimality conditions

$$\begin{aligned} & \underline{b} \leq Ax^\star \leq \overline{b}, \qquad \underline{x} \leq x^\star \leq \overline{x} \\ & c - A^T y^\star - z^\star = 0 \\ & z_k^\star > 0 \implies x_k^\star = \underline{x}_k \\ & z_k^\star < 0 \implies x_k^\star = \overline{x}_k \\ & y_i^\star > 0 \implies (Ax^\star)_i = \underline{b}_i \\ & y_i^\star < 0 \implies (Ax^\star)_i = \overline{b}_i. \end{aligned} \tag{2}$$

A few subtleties are worth pointing out. First, for an unbounded variable x_k (with $\underline{x}_k = -\infty$ and $\overline{x}_k = +\infty$), the corresponding dual variable $z_k^\star$ must be zero. Second, if $\underline{b}_i = -\infty$, then $y_i^\star \leq 0$, and if $\overline{b}_i = +\infty$, then $y_i^\star \geq 0$. Third, if there does not exist any primal-dual pair $(x^\star, y^\star, z^\star)$ satisfying these conditions, then the problem is either infeasible or unbounded.

In the following subsections, we summarize some of the most important ideas of presolving an LP of the form (1). First we introduce the notion of *reductions*, each of which consists of a *presolve transformation* and a *postsolve transformation*. Next, we describe *primal* and *dual exploration*, which analyze the primal and dual constraints to identify opportunities for invoking reductions.

2.1 Reductions

The idea of a presolver is to apply a sequence of *reductions* to the original problem to obtain a smaller, equivalent problem that can be solved more efficiently. The notion of reductions is standard in both the theory and practice of computer science, where it underlies, for example, program rewriting systems in compilers [ALSU06] and transformation-based modeling frameworks for optimization such as CVXPY [AVDB18].

In the context of LP presolving, each reduction consists of a *presolve transformation* and a *postsolve transformation*. The presolve transformation takes as input one problem and outputs an equivalent problem that is, in some sense, smaller or simpler than the input problem. The postsolve transformation takes as input a solution to the reduced problem and maps it to a solution of the original problem. Each reduction is very simple and cheap to apply, and for large LPs several hundreds of thousands of reductions may be applied in sequence.

A first example of a reduction is *fixing a variable*. Suppose we for some reason fix a variable x_k to some value $\hat{x}_k$ (we discuss how to identify such variable fixings below), and assume that this is the only reduction we apply. The *presolve* transformation for this reduction consists of removing the variable x_k from the original problem by substituting its fixed value into the remaining parts of the problem. The variable x_k appears in the original problem but not in the reduced problem, so the *postsolve* transformation must recover an optimal primal variable $(x_k^\star)_{\text{original}}$ and an optimal dual variable $(z_k^\star)_{\text{original}}$ of the original problem from a solution of the reduced problem. The *primal* postsolve transformation simply sets $(x_k^\star)_{\text{original}} = \hat{x}_k$. The *dual* postsolve transformation sets $(y^\star)_{\text{original}} = (y^\star)_{\text{reduced}}$ and $(z_k^\star)_{\text{original}} = c_k - a_k^T(y^\star)_{\text{reduced}}$, where a_k is the kth column of A and $(y^\star)_{\text{reduced}}$ is the optimal dual variable of the reduced problem. (This choice of $(z_k^\star)_{\text{original}}$ follows from the dual feasibility condition, *i.e.*, the second line of (2).)

In addition to the reduction of fixing a variable, there are four main reductions:

- removing a constraint,
- adding a multiple of an equality constraint to another constraint,
- substituting a variable appearing in only one equality constraint, and
- changing variable bounds.

While classic papers on presolving, such as [BMW75, AA95, ABG+19], list several more complicated problem transformations as reductions, we use a different abstraction where we break down complex transformations into sequences of simple reductions. In our abstraction, a reduction is the *simplest* atomic rewriting unit of a problem. Most problem transformations described in the literature on LP presolve can be broken down into sequences of the 5 reductions mentioned above. This abstraction does not only make it easier to describe a presolver, but it also simplifies the implementation of one of the most complicated parts of the presolver: recovering optimal dual variables of the original problem from an optimal primal-dual solution of the reduced problem.

To find valid reductions (*i.e.*, problem transformations that reduce the size of the problem while provably maintaining an equivalence to the original problem), we use *primal exploration* and *dual exploration*, as described next.

2.2 Primal exploration

Primal exploration analyzes the *primal* constraints to identify opportunities for reductions. The logical rules applied during this process guarantees that the feasible set of the reduced problem is equivalent to that of the original formulation. Roughly speaking, primal exploration identifies reductions that preserve the feasible set while producing a more compact representation with less redundancy.

A simple example of primal exploration is the *redundant constraint explorer*, which uses variable bounds to identify redundant constraints. For example, consider the bounds $x_1 \geq 1$ and $x_2 \geq 2$ together with the constraint $x_1 + 2x_2 \geq 5$. Since the variable bounds imply that the constraint is satisfied, the constraint is classified as redundant. Once such a constraint is identified, the redundant constraint explorer invokes the corresponding reduction to remove it from the problem.

Another example is the *doubleton row explorer*, which finds equality constraints with only two variables and substitutes one variable in terms of the other. For example, consider the constraint $x_1 + x_2 = 1$ together with the bounds $x_1, x_2 \geq 0$. First, we transfer the bound $x_1 \geq 0$ to an upper bound on x_2, giving $x_2 \leq 1$ and making x_1 unbounded. (This is done by invoking the reduction that changes variable bounds.) Second, we substitute $x_1 = 1 - x_2$ in the remaining parts of the problem. (This is done by first repeatedly invoking the reduction that adds a multiple of an equality constraint to another constraint, followed by the reduction that substitutes a variable appearing in only one equality constraint.) Third, we remove the constraint $x_1 + x_2 = 1$. (This is done by invoking the reduction that deletes a constraint.) Once the doubleton row explorer identifies an equality constraint with only two variables, it invokes these reductions in sequence. The reduced problem has one less variable and one less constraint, and the reduced feasible set is equivalent to the feasible set of the original problem in the sense that if $x_2 = a$ for some $a \in \mathbf{R}$ is feasible for the reduced problem, then $(x_1, x_2) = (1 - a, a)$ is feasible in the original problem.

We list more examples of primal exploration in §3.1.

2.3 Dual exploration

Dual exploration analyzes the *dual* constraints to identify opportunities for reductions based on complementary slackness and dual variable bounds. Unlike primal exploration, dual exploration may identify reductions that do not preserve the feasible set of the original problem. However, the logical rules applied during dual exploration guarantee that any optimal solution of the reduced problem can still be mapped to an optimal solution of the original problem.

Dual exploration is further categorized into *strong* and *weak* exploration. The semantics behind weak dual exploration imply that any reduction it identifies preserves the entire optimal solution set: *every* optimal solution of the original problem can, in principle, be recovered from some optimal solution of the reduced problem. In contrast, the semantics of strong dual exploration are looser: it may yield more aggressive reductions, but these only guarantee that *at least one* optimal solution of the original problem can be recovered from

an optimal solution of the reduced problem.

A simple example of dual exploration is the *variable lock explorer*. An *uplock* of a variable is a constraint that prevents the variable from increasing to infinity. If a variable x_k has objective coefficient $c_k < 0$ and no uplock, then the dual constraints imply that $z_k < 0$ for any dual feasible point. By complementary slackness, this means that x_k will be equal to its upper bound $\overline{x}_k$ at any optimal solution, so the variable lock explorer can invoke the reduction that fixes x_k to $\overline{x}_k$. (If $\overline{x}_k = +\infty$, then the problem is unbounded.) This is an example of weak dual exploration, since *any* optimal solution of the original problem must have $x_k = \overline{x}_k$. If instead $c_k = 0$ and x_k has no uplock, the variable lock explorer can still invoke the reduction of fixing x_k to $\overline{x}_k$, but not every primal optimal solution $x^\star$ of the original problem is guaranteed to have $x^\star_k = \overline{x}_k$, so this is an example of strong dual exploration.

The variable lock explorer is a special case of a more general dual exploration phase that propagates the dual constraints to strengthen dual variable bounds and identify reductions based on complementary slackness (see, *e.g.*, [AA95, §3.4]). Other examples of dual exploration include searching for *column singletons in inequality constraints* [BMW75, §1.2] and *dominated columns* [GKM+15, §4]. Interestingly, many dual exploration rules can be interpreted as primal explorers applied to the dual problem. For example, both the variable lock explorer and the dominated columns explorer are just special primal redundant constraint explorers applied to the dual problem.

3 Design and implementation

While the main concepts behind LP presolving are straightforward and simple to describe, implementing them in practice involves numerous design choices. In this section, we discuss some of the most important decisions and describe our implementation.

3.1 Explorers

The most important design choice is which explorers to implement. Each explorer applies a set of logical rules to identify opportunities for reductions. Adding more explorers typically yields smaller reduced problems, but at the cost of increased presolve time. Table 1 describes and groups commonly used explorers into three categories based on their computational cost.

- *Fast* explorers require minimal computation to identify reductions, as they operate primarily by scanning *internal statistics* maintained by the presolver (we describe what internal statistics are in greater detail in §3.2).
- *Medium* explorers require additional computation and cannot be executed using internal statistics alone. For example, the explorer that detects parallel rows belongs to this category because it requires a two-level hashing algorithm together with pairwise comparison of rows to identify constraints that are parallel [ABG+19, §5.2].

- *Slow* explorers require substantially more computation. For example, one explorer in this category detects linear dependence among equality constraints and requires rank-revealing matrix factorizations [And95, MG03]. A related explorer identifies opportunities for reducing the number of nonzero entries in the constraint matrix by adding a multiple of an equality constraint to another constraint to make the latter sparser [Gon97], similar to Gaussian elimination.

The categorization in table 1 is based on our experience with presolving and is therefore somewhat subjective.

Almost all of the explorers listed in table 1 were proposed over several decades by researchers with either CPU-based simplex or interior-point solvers in mind, so it is natural to ask whether all of them are worthwhile for GPU-accelerated first-order solvers. For example, first-order solvers open up new opportunities for solving extremely large LPs where traditional simplex and interior-point solvers may be too slow or run out of memory when factorizing large sparse matrices. Since the linear dependence explorer requires matrix factorization itself, it seems counterintuitive to include it in the presolve for GPU-accelerated first-order solvers, as one of the main motivations for using such solvers is to *avoid* matrix factorization. Moreover, our experiments in §4 show that most of the problem reduction (on average about 90%) is detected by fast and medium explorers. It is therefore unclear whether the additional reductions detected by slow explorers generally justify the extra presolve time for problems where first-order methods may be the only viable option.

3.2 Data structures

While most exploration rules are simple to describe, their efficient implementation is often quite involved. To make the exploration phase efficient, a presolver keeps track of several internal statistics, which typically include the number of nonzeros in each row and column of A, the variable locks (*i.e.*, the number of constraints that prevent a variable from increasing or decreasing to $\pm\infty$), and the smallest and largest value a constraint can take given the current variable bounds (these are often referred to as the *minimum* and *maximum activity* of a constraint [ABG+19, §2]). The internal statistics must be maintained, so they are updated each time a reduction is applied. Furthermore, presolving is arranged in *rounds*. To avoid useless work from re-examining all constraints in every round to identify opportunities for reductions, one option is to keep track of the constraints modified in the previous round and apply operations such as primal propagation only to those constraints.

Some of the explorers require fast access to rows while others require fast access to columns. One option is to store the constraint matrix A in compressed sparse row (CSR) format to enable fast and cache-friendly access to rows, and supplement the structure of A by a linked lists superstructure describing its columns [GT04]. Another option, adopted by PaPILO [GGH23] and our presolver PSLP, is to store A in both CSR and compressed sparse column (CSC) formats. Storing A in CSC format enables fast and cache-friendly access to columns, but care must be taken to ensure that the two representations of A remain consistent once reductions are applied. Furthermore, some reductions require modifying the

Explorer name		Description
		Fast explorers
Singleton rows	(P)	Searches for rows with a single nonzero entry, which can be used to fix the value or update the bounds of a variable [BMW75, §1.2].
Doubleton rows	(P)	Searches for equality constraints with two nonzero entries, allowing one variable to be eliminated via substitution [BBG83].
Redundant constraints	(P)	Searches for constraints that are implied by variable bounds and can thus be deleted [BMW75, §1.2].
Column singleton (in equality constraint)	(P)	Searches for equality constraints with a variable appearing in only that constraint, possibly allowing the variable to be substituted and the constraint removed [AA95, §3.2].
Variable locks	(D)	Searches for variables whose locks allow them to be fixed [ABG+19, §4.4].
Column singleton (in inequality constraint)	(D)	Searches for inequality constraints with a singleton column, possibly allowing the variable to be fixed or the inequality constraint converted to an equality constraint [BMW75, §1.2].
		Medium explorers
Parallel rows	(P)	Searches for constraints that are parallel, allowing many of them to be deleted [TW86], [ABG+19, §5.2].
Parallel columns	(P)	Searches for columns that are parallel, allowing many of them to be aggregated [AA95, §3.6], [ABG+19, §6.3].
Primal propagation	(P)	Searches for variable bounds implied by the constraints [BMW75, §1.2], [ABG+19, §3.2].
Dual propagation	(D)	Searches for dual variable bounds implied by the dual constraints [AA95, §3.4], [ABG+19, §7.5].
		Slow explorers
Linear dependence	(P)	Searches for linearly dependent equality constraints [And95, MG03].
Variable substitution	(P)	Searches for variables that can be eliminated via substitution without introducing excessive fill-in [MS03, §2.2], [ABG+19, §4.5].
Variable symmetry	(P)	Searches for symmetries among variables, allowing variables to be aggregated [GKMS14], [ABG+19, §7.1].
Primal sparsification	(P)	Searches for opportunities to combine constraints to make them sparser [Gon97], [ABG+19, §5.3].
Dual sparsification	(D)	Searches for opportunities to combine dual constraints to make them sparser [GC+20].
Dominated columns	(D)	Searches for pairs of variables where one is *dominated*, *i.e.*, worse than the other with respect to both the constraints and the objective, and can thus possibly be fixed [GKM+15, §4], [ABG+19, §6.4].

Table 1: Explorers grouped by computational cost. The letters (P) and (D) indicate whether the explorer is primarily primal or dual.

sparsity pattern of A and may lead to a larger number of nonzero entries in certain rows or columns, so it is useful to append each row in the CSR representation and each column in the CSC representation with a small amount of extra space to allow for new nonzero entries. If the extra space is exhausted for a specific row or column, nearby rows or columns may have more extra space, which can be used after shifting the data for nearby rows or columns.

3.3 Our implementation

Our implementation, which is based on the reduction-based abstraction described in §2, is called PSLP (Presolver for Linear Programs) and is available at

https://github.com/dance858/PSLP.

PSLP is implemented in C and runs (mostly) single-threaded on the CPU, with certain parts parallelized on two or four POSIX threads. To avoid becoming a bottleneck when used together with a GPU-accelerated LP solver, PSLP is designed to be fast and lightweight. Consequently, it implements only the fast and medium explorers listed in table 1, and omits the slow explorers. For certain problem instances, the additional reductions detected by slow explorers could be large enough to justify their use. However, PSLP does not currently implement slow explorers due to (1) the substantial additional engineering effort required to implement them efficiently, and (2) their marginal benefit in terms of problem-size reduction appears limited. Specifically, PSLP already attains, on average, about a 90% reduction in problem size relative to Gurobi's presolver that implements the slow explorers [ABG+19], while being significantly faster (see table 2 in §4).

PSLP is solver independent, meaning that it can be used as a standalone presolver module for any LP solver. Its interface is minimal and just requires the user to provide pointers to the problem data, so solvers that lack a built-in presolver can easily integrate it as an external component with just a few lines of code. PSLP takes in an LP of the form (1), and emits a reduced LP on the same form. Given an (approximate) solution to the reduced problem, it recovers an (approximate) solution to the original problem, *i.e.*, $(x^\star, y^\star, z^\star)$ (approximately) satisfying (2).

4 Experiments

In this section we first compare the presolve performance of PSLP against the commercial state-of-the-art presolver implemented in Gurobi [ABG+19]. There are two main reasons for this comparison. First, we wish to gain insight into the relative importance of fast and medium explorers versus slow explorers, as PSLP implements only the former while Gurobi's presolver implements all three categories. Second, since a central goal of this paper is to assess whether presolving improves the performance of GPU-accelerated first-order LP solvers, it is essential that the presolver itself is reasonably effective. An inefficient presolver would prevent a fair and honest assessment of the impact of presolving and could distort the

results. Comparing PSLP to a commercial state-of-the-art presolver therefore also serves to validate that it is adequate for a fair assessment of the impact of presolving.

After comparing PSLP to Gurobi's presolver, we evaluate how presolving using PSLP or Gurobi affects the performance of cuPDLPx [LPY25], a GPU-accelerated first-order LP solver based on PDHG.

Benchmark datasets. We conduct numerical experiments on two LP benchmark datasets. First, we consider Mittelmann's LP collection [Mit25], which consists of 48 publicly available instances with more than 10^5 nonzero entries. Second, we consider 383 instances derived from root-node LP relaxations of MIPLIB 2017 [GHG+21], using the filtering criteria described in [LY25]. We categorize problems into three groups based on their size: *small* (10^5-10^6 nonzero entries), *medium* (10^6-10^7 nonzero entries), and *large* (more than 10^7 nonzero entries).

Hardware. Problems are presolved on the CPU and solved on the GPU. For the presolve step, we use a Macbook M4 pro (14 cores) with 48 GB of unified memory, and then execute cuPDLPx on a NVIDIA H100-SXM-80GB GPU with CUDA 12.4.

4.1 Presolve performance

Setup. We compare PSLP with Gurobi's presolver (version 13.0, run with default settings) using the following metrics:

- Arithmetic and geometric mean of the presolve time, given in seconds (denoted by "AM-T" and "GM-T", respectively). For each problem instance, we average the presolve time over 10 runs.

- The average reduction in the number of nonzero entries, given as the ratio of the number of nonzero entries in the reduced problem to that in the original problem (denoted by "ratio (nnz)").

Results. Table 2 summarizes the results. On Mittelmann's LP collection, the average per-instance ratio of Gurobi's presolve time to PSLP's presolve time is 11.8 (not shown in the table). Despite this substantial speedup of an order of magnitude, PSLP on average achieves 90% of the reduction in the number of nonzero entries achieved by Gurobi's presolver. On the MIPLIB relaxations, the corresponding average time ratio is 6.58 (also not shown in the table), while PSLP on average attains 94% of the reduction in the number of nonzero entries achieved by Gurobi's presolver.

Figures 1 and 2 compare the reductions in the number of nonzero entries achieved by PSLP and Gurobi's presolver across the two datasets, together with the corresponding presolve times. (For the MIPLIB relaxations, results are reported only for the 50 largest instances due to space constraints.) Overall, PSLP achieves reductions comparable to those of Gurobi's presolver on all but five instances (`L1_sixm1000obs`, `scpm1`, and `L1_sixm250obs` from Mittelmann's LP collection, and `neos-3208254-reui` and `scpm2` from the MIPLIB relaxations), while often requiring significantly less presolve time.

Figure 1: Comparison of PSLP and Gurobi's presolver on Mittelmann's LP collection. Each bin shows the ratio of the nonzero entries in the reduced problem to that in the original problem. Above each bin, we show the number of nonzero entries in the original problem, and the presolve times of PSLP and Gurobi. The problems are sorted by the number of nonzero entries in the original problem.

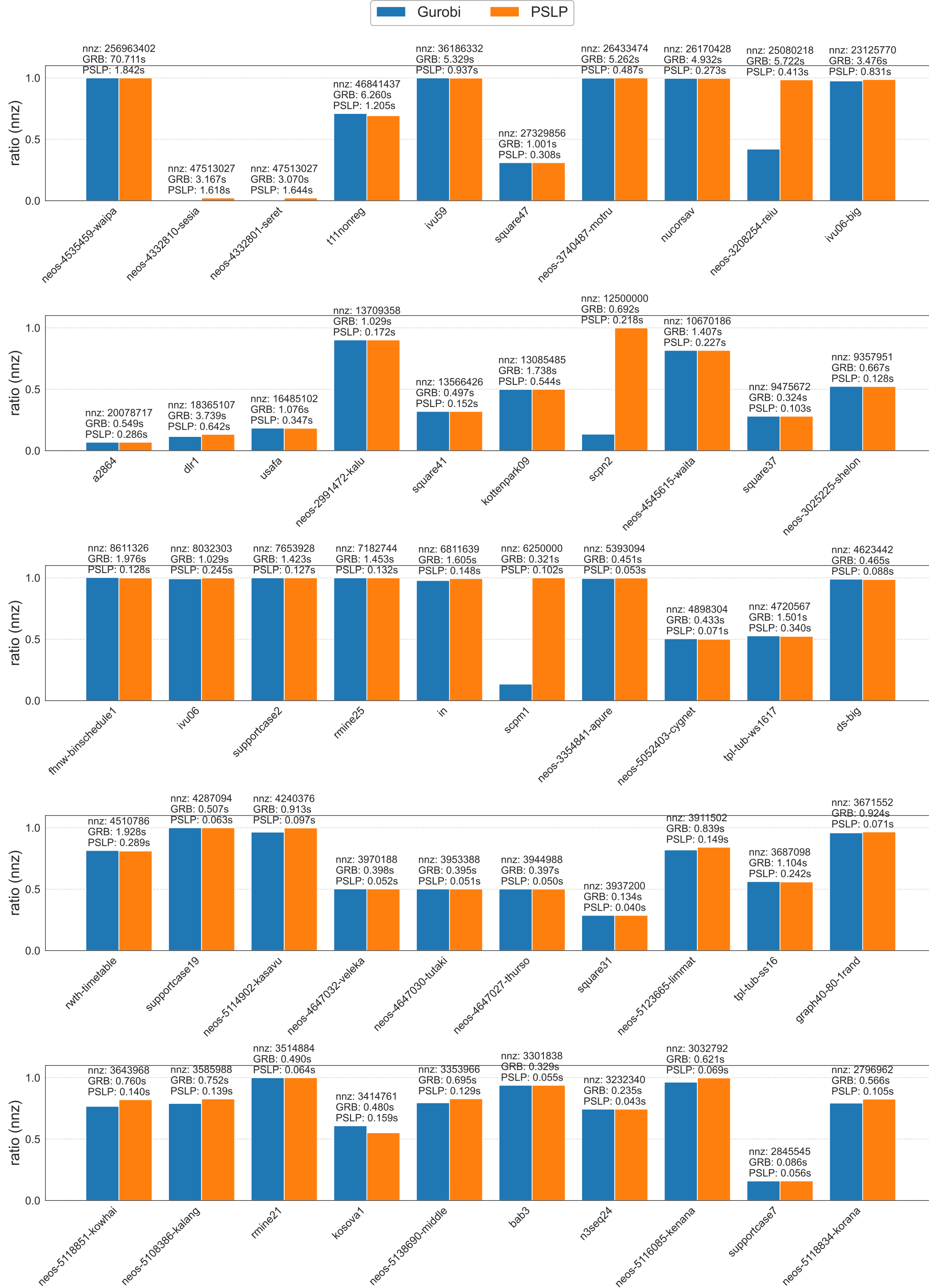


Figure 2: Comparison of PSLP and Gurobi's presolver on the 50 largest MIPLIB relaxations.

Dataset	Presolver	AM	GM	ratio (nnz)
Mittelmann	Gurobi	3.59	0.70	0.71
	PSLP	0.33	0.084	0.79
MIPLIB relaxations	Gurobi	0.47	0.077	0.76
	PSLP	0.057	0.013	0.81

Table 2: Comparison of PSLP and Gurobi's presolver. The arithmetic mean (AM) and geometric mean (GM) of the presolve time are given in seconds.

4.2 End-to-end solver performance: experimental setup

Solver. We run cuPDLPx (version 0.2.5) both with and without PSLP and Gurobi's presolver enabled. The termination criterion adopted by cuPDLPx depends on a relative tolerance parameter $\epsilon_{\text{rel}} > 0$. We use $\epsilon_{\text{rel}} = 10^{-8}$ in all experiments, which corresponds to the default value for higher accuracy [LPY25].

Time limit. For the MIPLIB relaxations, we set a time limit of 3600 seconds for small and medium-sized problems, and 18000 seconds for large problems. For Mittelmann's LP collection, we set a time limit of 15000 seconds. (These time limits are the same as those used in [LPY25].) If an instance remains unsolved when the time limit is reached, we count it as a failure and set its solve time to the time limit for the purpose of computing average solve times.

Performance metrics. We consider the following common metrics:

- **Count**: The number of problems solved within the time limit.
- **SGM**: We use the shifted geometric mean of the total solve time, given in seconds. The shifted geometric mean is defined as $(\prod_{i=1}^{K}(t_i + \Delta))^{1/K} - \Delta$, where t_i is the total solve time for the ith instance (in seconds), K is the total number of instances, and $\Delta > 0$ is the shift. We shift by $\Delta = 1$ and $\Delta = 10$ seconds and denote these metrics as SGM1 and SGM10, respectively. Without presolving, the total solve time is simply the time taken by cuPDLPx to solve the original problem. With presolving, the total solve time is the sum of the presolve time and the time taken by cuPDLPx to solve the reduced problem.
- **Win rate**: The percentage of problem instances for which the total solve time with presolve enabled is less than the total solve time with presolve disabled.

4.3 End-to-end solver performance: results

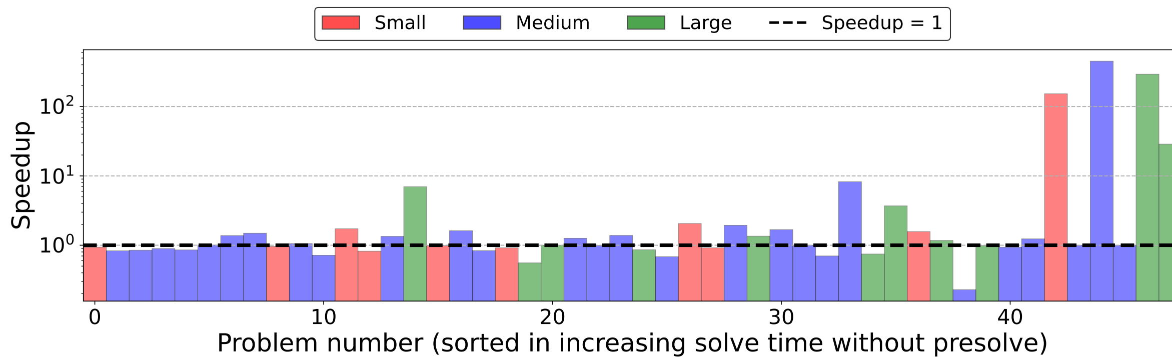
Presolve impact on total solve time. Table 3 summarizes the results. We see that enabling PSLP improves the shifted geometric mean in all three categories for both datasets

	Mittelmann				MIPLIB			
	Count	**SGM1**	**SGM10**	**WR (%)**	**Count**	**SGM1**	**SGM10**	**WR (%)**
Small problems	10	——	——	——	269	——	——	——
cuPDLPx	9	16.23	33.20	——	265	2.27	6.08	——
cuPDLPx + PSLP	10	8.31	14.23	40	269	1.48	3.54	60
cuPDLPx + Gurobi	10	11.83	19.75	50	269	1.40	3.01	44
Medium problems	27	——	——	——	94	——	——	——
cuPDLPx	24	28.85	60.76	——	92	5.57	12.28	——
cuPDLPx + PSLP	25	21.15	45.17	52	93	4.62	10.54	53
cuPDLPx + Gurobi	25	19.70	39.26	41	93	4.87	10.01	34
Large problems	11	——	——	——	20	——	——	——
cuPDLPx	9	136.51	169.33	——	14	286.03	408.26	——
cuPDLPx + PSLP	11	50.44	63.05	64	19	22.12	44.12	75
cuPDLPx + Gurobi	11	56.10	68.05	36	19	22.97	42.18	70
All problems	48	——	——	——	383	——	——	——
cuPDLPx	42	36.78	69.01	——	371	3.90	10.77	——
cuPDLPx + PSLP	46	21.43	39.57	52	381	2.41	6.13	59
cuPDLPx + Gurobi	46	22.64	39.35	42	381	2.37	5.54	43

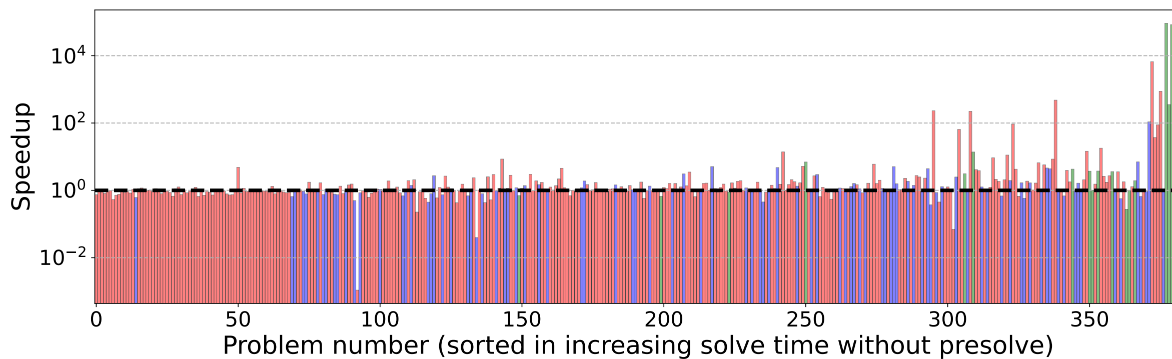
Table 3: Results on Mittelmann's LP collection and MIPLIB relaxations. The abbreviation WR stands for win rate, which is the percentage of instances for which presolve enabled is faster than presolve disabled, and SGM1 and SGM10 are given in seconds.

compared to the case with presolve disabled. The improvement is more significant for the large problems, with almost a three-fold improvement for Mittelmann's LP collection and more than a nine-fold improvement for the MIPLIB relaxations. We also see that the solve times with either PSLP or Gurobi's presolver enabled are similar. Interestingly, given the significant improvement in solve time with presolve enabled, the win rate (*i.e.*, the percentage of instances for which presolve enabled is faster than presolve disabled) is lower than one might expect. One possible explanation is that for problems where cuPDLPx already converges fast with a modest number of iterations, presolving becomes harmful. On the other hand, there exist problems where cuPDLPx is very slow without presolve but becomes fast with presolve, which gives a significant improvement in solve time. We illustrate this in Figure 3, which for each problem instance shows the speedup of enabling PSLP for cuPDLPx. In the right part of the figure, which corresponds to problems with the longest solve time without presolve, we see that the speedup from presolve can be orders of magnitude.

Time spent in presolve. We analyze the ratio between the time spent in presolve and the time spent on solving the *reduced* problem. Table 4 summarizes the average ratio across different problem categories. We see that the presolve time for PSLP is generally a small fraction of the solve time that tends to be more significant for larger problems. Gurobi's presolve time is even more significant and sometimes dominates the time spent on solving the reduced problem.



(a) Mittelmann's LP collection.



(b) MIPLIB relaxations.

Figure 3: The speedup (*i.e.*, the ratio of the total solve time without presolve to the total solve time with presolve) of enabling PSLP for cuPDLPx for each problem instance.

	Mittelmann				MIPLIB			
	Small	**Medium**	**Large**	**All**	**Small**	**Medium**	**Large**	**All**
PSLP (%)	1.3	12.6	19.3	11.8	7.4	20.6	39.1	12.1
Gurobi (%)	6.2	100.9	82.3	76.9	29.1	105.9	198.3	56.0

Table 4: Average presolve time as a percentage of the time it takes for cuPDLPx to solve the reduced problem.

5 Discussion

We have revisited LP presolving, a classical topic in the optimization literature, from the perspective of GPU-accelerated first-order solvers. While a myriad of presolve techniques have been proposed over the decades, a systematic investigation of their cost-effectiveness in the context of GPU-accelerated first-order methods has been lacking. To address this, we organized these techniques into a three-class taxonomy based on computational cost and implemented the two cheaper classes in PSLP, a new open-source presolver designed to be lightweight. By comparing PSLP to the commercial state-of-the-art presolver in Gurobi, which implements all three classes of presolve techniques, our experiments on standard benchmarks reveal several findings:

- A lightweight presolver can achieve reductions comparable to those of a sophisticated state-of-the-art commercial presolver, at a fraction of the computational cost (see table 2 and figures 1 and 2).
- The time spent in presolve can be significant relative to the time spent on solving the reduced problem (see table 4).
- Lightweight presolve can substantially improve the end-to-end performance of GPU-accelerated first-order solvers (see table 3). While many problems are largely unaffected by presolve, the speedup can be orders of magnitude for certain larger problems (see figure 3).

Presolve has traditionally been regarded as a preprocessing step whose cost is negligible relative to the solve phase. Our first two findings challenge this assumption in the context of GPU-accelerated first-order methods, and together highlight the growing importance of presolve cost. As GPU hardware improves, the solve phase of first-order methods will continue to accelerate, but the sequential nature of presolving means it will not benefit from these advances to the same extent. Over time, presolve therefore risks becoming an increasingly dominant fraction of the total solve time. Our results show that by restricting to cheaper presolve techniques, much of the reduction benefit can be retained without presolve becoming an excessive bottleneck. Nonetheless, even with a lightweight presolver such as PSLP, the presolve time is already non-negligible (see table 4). As first-order solvers become faster, the overall speedup in total solve time may therefore be limited unless the presolve phase itself

is accelerated. (For example, if cuPDLPx became twice as fast due to hardware improvements, a rough estimate based on table 4 suggests that the speedup in total solve time for large MIPLIB problems would be about 1.20× if Gurobi's presolver were used, rather than the expected 2×.) While skipping presolve altogether is possible, our results demonstrate the benefit of presolving with PSLP, suggesting that it should generally not be skipped in solvers similar to cuPDLPx. More broadly, future research should consider accelerating not only the solve phase but also the presolve phase, which has been largely neglected in the literature. Given the substantial engineering effort required to develop an efficient presolver, such research questions have until now been impractical to explore. By open-sourcing PSLP, we hope to enable and encourage research in this direction.

Our third finding highlights the continued importance of presolving for linear programming, even in the current era of GPU-accelerated first-order solvers. It also indicates that presolving has a stronger impact on first-order solvers (at least cuPDLPx) than is suggested by the seminal work of [ADH+21], where PaPILO [GGH23] was used for presolving and only very marginal reductions in iteration counts were reported when presolving was applied prior to a first-order method. Furthermore, the presolve time (which is significant relative to the solve time for the reduced problem) was not included in the performance metrics reported in [ADH+21].

An efficient LP presolver forms the foundation of presolvers for more general problem classes. In ongoing work, we are extending PSLP to support quadratic programs and convex conic programs involving second-order cones and exponential cones.

Acknowledgments

We are very grateful to Zedong Peng, main developer of cuPDLPx, for collecting the data for the cuPDLPx experiments in §4, and for providing exceptionally detailed and valuable feedback on PSLP. Without his input, PSLP would not be in the shape it is. We also thank Max Schaller, Kasper Johansson, Chris Maes, and Rajesh Gandham for helpful feedback and discussions.

References

[AA95] E. Andersen and K. Andersen. Presolving in Linear Programming. *Mathematical Programming*, 71(2):221–245, 1995.

[ABG+19] T. Achterberg, R. Bixby, Z. Gu, E. Rothberg, and D. Weninger. Presolve Reductions in Mixed Integer Programming. *INFORMS Journal on Computing*, 32(2):473–506, 2019.

[ADH+21] D. Applegate, M. Díaz, O. Hinder, H. Lu, M. Lubin, B. O'Donoghue, and W. Schudy. Practical Large-Scale Linear Programming Using Primal-Dual Hybrid Gradient. *Advances in Neural Information Processing Systems*, 34:20243–20257, 2021.

[ADH+25] D. Applegate, M. Díaz, O. Hinder, H. Lu, M. Lubin, B. O'Donoghue, and W. Schudy. PDLP: A Practical First-Order Method for Large-Scale Linear Programming. *arXiv preprint arXiv:2501.07018*, 2025.

[ADLL24] D. Applegate, M. Díaz, H. Lu, and M. Lubin. Infeasibility Detection with Primal-Dual Hybrid Gradient for Large-Scale Linear Programming. *SIAM Journal on Optimization*, 34(1):459–484, 2024.

[ALSU06] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2 edition, 2006.

[And95] E. Andersen. Finding All Linearly Dependent Rows in Large-Scale Linear Programming. *Optimization Methods and Software*, 6(3):219–227, 1995.

[AVDB18] A. Agrawal, R. Verschueren, S. Diamond, and S. Boyd. A Rewriting System for Convex Optimization Problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

[BBG83] G. Bradley, G. Brown, and G. Graves. Structural Redundancy in Large-Scale Optimization Models. In *Redundancy in Mathematical Programming*, pages 145–169, Berlin, Heidelberg, 1983. Springer Berlin Heidelberg.

[BMW75] A. Brearley, G. Mitra, and H. Williams. Analysis of Mathematical Programming Problems Prior to Applying the Simplex Algorithm. *Mathematical Programming*, 8(1):54–83, 1975.

[BS94] R. Bixby and M. Saltzman. Recovering an Optimal LP Basis from an Interior Point Solution. *Operations Research Letters*, 15(4):169–178, 1994.

[CP11] A. Chambolle and T. Pock. A First-Order Primal-Dual Algorithm for Convex Problems with Applications to Imaging. *Journal of Mathematical Imaging and Vision*, 40(1):120–145, 2011.

[CSY+25] K. Chen, D. Sun, Y. Yuan, G. Zhang, and X. Zhao. HPR-LP: An Implementation of an HPR Method for Solving Linear Programming. *Mathematical Programming Computation*, pages 1–28, 2025.

[Dan51] G. Dantzig. Maximization of a Linear Function of Variables Subject to Linear Inequalities. In T. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347. Wiley and Chapman Hall, 1951. Available at https://cowles.yale.edu/sites/default/files/2022-09/m13-all.pdf.

[DCB13] A. Domahidi, E. Chu, and S. Boyd. ECOS: An SOCP Solver for Embedded Systems. In *2013 European Control Conference (ECC)*, pages 3071–3076, 2013.

[FLH05] J. Forrest and R. Lougee-Heimer. *CBC User Guide*, pages 257–277. INFORMS, 2005.

[Gal23] I. Galabova. *Presolve, Crash and Software Engineering for HiGHS*. PhD dissertation, The University of Edinburgh, 2023.

[GC+20] P. Gemander, W. Chen, , D. Weninger, L. Gottwald, A. Gleixner, and A. Martin. Two-Row and Two-Column Mixed-Integer Presolve Using Hashing-Based Pairing Methods. *EURO Journal on Computational Optimization*, 8(3):205–240, 2020.

[GC24] P. Goulart and Y. Chen. Clarabel: An Interior-Point Solver for Conic Programs with Quadratic Objectives. *arXiv preprint arXiv:2405.12762*, 2024.

[GGH23] A. Gleixner, L. Gottwald, and A. Hoen. PaPILO: A Parallel Presolving Library for Integer and Linear Optimization with Multiprecision Support. *INFORMS Journal on Computing*, 35(6):1329–1341, 2023.

[GHG+21] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. Christophel, K. Jarck, T. Koch, J. Linderoth, M. Lübbecke, H. Mittelmann, D. Ozyurt, T. Ralphs, D. Salvagnin, and Y. Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation*, 13(3):443–490, 2021.

[GKM+15] G. Gamrath, T. Koch, A. Martin, M. Miltenberger, and D. Weninger. Progress in Presolving for Mixed Integer Programming. *Mathematical Programming Computation*, 7(4):367–398, 12 2015.

[GKMS14] M. Grohe, K. Kersting, M. Mladenov, and E. Selman. Dimension Reduction via Colour Refinement. In *European Symposium on Algorithms*, pages 505–516. Springer, 2014.

[Gon97] J. Gondzio. Presolve Analysis of Linear Programs Prior to Applying an Interior Point Method. *INFORMS Journal on Computing*, 9(1):73–91, 1997.

[GT04] N. Gould and P. Toint. Preprocessing for Quadratic Programming. *Mathematical Programming*, 100(1):95–132, 2004.

[HH18] Q. Huangfu and J. Hall. Parallelizing the Dual Revised Simplex Method. *Mathematical Programming Computation*, 10(1):119–142, Mar 2018.

[LPY25] H. Lu, Z. Peng, and J. Yang. cuPDLPx: A Further Enhanced GPU-Based First-Order Solver for Linear Programming. *arXiv preprint arXiv:2507.14051*, 2025.

[LY25] H. Lu and J. Yang. cuPDLP.jl: A GPU Implementation of Restarted Primal-Dual Hybrid Gradient for Linear Programming in Julia. *Operations Research*, 2025.

[LYH+23] H. Lu, J. Yang, H. Hu, Q. Huangfu, J. Liu, T. Liu, Y. Ye, C. Zhang, and D. Ge. cuPDLP-C: A Strengthened Implementation of cuPDLP for Linear Programming by C Language. *arXiv preprint arXiv:2312.14832*, 2023.

[Mar02] I. Maros. *Computational Techniques of the Simplex Method*, volume 61. Springer Science & Business Media, 2002.

[MG03] L. Miranian and M. Gu. Strong Rank-Revealing LU Factorizations. *Linear Algebra and Its Applications*, 367:1–16, 2003.

[Mit25] H. Mittelmann. Decision Tree for Optimization Software. http://plato.asu.edu/guide.html, 2025. Accessed: 2025-11-06.

[MS03] C. Mészáros and U. Suhl. Advanced Preprocessing Techniques for Linear and Quadratic Programming. *OR Spectrum*, 25(4):575–595, 2003.

[NVI25] NVIDIA Corporation. NVIDIA cuopt: GPU-accelerated optimization engine. https://docs.nvidia.com/cuopt/index.html, 2025. Version 25.10.

[OCPB16] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. Conic Optimization via Operator Splitting and Homogeneous Self-Dual Embedding. *Journal of Optimization Theory and Applications*, 169(3):1042–1068, June 2016.

[Pan22] E. Panos. CPLEX Barrier Options for TIMES Models, March 2022. Slides from ETSAP Webinar.

[PF25] L. Perron and V. Furnon. OR-tools, 2025.

[Rot20] E. Rothberg. How to Exploit Parallelism in Linear Programming and Mixed-Integer Programming. Gurobi Optimization webinar / slides, 2020. Gurobi Optimization, LLC.

[RTV05] C. Roos, T. Terlaky, and J. Vial. *Interior Point Methods for Linear Optimization.* Springer, 2005.

[SBG+20] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. OSQP: An operator splitting solver for quadratic programs. *Mathematical Programming Computation*, 12(4):637–672, 2020.

[SGP22] B. Sofranac, A. Gleixner, and S. Pokutta. Accelerating Domain Propagation: An Efficient GPU-Parallel Algorithm over Sparse Matrices. *Parallel Computing*, 109:102874, 2022.

[TW86] L. Tomlin and J. Welch. Finding Duplicate Rows in a Linear Programming Model. *Operations Research Letters*, 5(1):7–11, 1986.

[Wil83] H. Williams. A Reduction Procedure for Linear and Integer Programming Models. In *Redundancy in Mathematical Programming*, pages 87–107, Berlin, Heidelberg, 1983. Springer Berlin Heidelberg.

[ZC08] M. Zhu and T. Chan. An Efficient Primal-Dual Hybrid Gradient Algorithm for Total Variation Image Restoration. *UCLA CAM Report*, 34(2), 2008.

[ZS25] Y. Zhang and N. Sahinidis. A Combined Linear and Nonlinear Presolve for Nonlinear Optimization. *EURO Journal on Computational Optimization*, page 100119, 2025.