cuHALLaR: A GPU Accelerated Low-Rank Augmented Lagrangian Method for Large-Scale Semidefinite Programming *

Jacob M. Aguirre† Diego Cifuentes‡ Vincent Guigues§ Renato D.C. Monteiro¶
Victor Hugo Nascimento‖ Arnesh Sujanani**

May 19, 2025 (second version: October 14th, 2025, third version November 11, 2026)

Abstract

This paper presents cuHALLaR, a GPU-accelerated solver with an explicit trace constraint $\mathrm{tr}(X) \le \tau^2$, where τ is supplied as part of the problem data. The solver is a Julia-based implementation of the hybrid low-rank augmented Lagrangian method, HALLaR, previously proposed by three of the authors. It exploits massive parallelism by performing all core operator evaluations, including linear maps, their adjoints, and gradients, entirely on the GPU. Extensive numerical experiments are conducted on three structured problem classes for which valid trace bounds can be easily computed: matrix completion, maximum stable set, and phase retrieval. The results demonstrate substantial performance gains over the optimized CPU implementations and GPU-based solvers included in the comparison. On the largest instances, cuHALLaR achieves speedups of up to 165 times and solves a matrix completion SDP with a two-million-by-two-million matrix variable and over 260 million constraints in 53 seconds to a relative precision of 10^{-5}. Similarly, speedups of up to 135x are observed for maximum stable set problems on graphs with over one million vertices. These results demonstrate that cuHALLaR can solve extremely large SDPs with bounded domains.

Keywords: semidefinite programming, augmented Lagrangian, low-rank methods, GPU acceleration, Frank-Wolfe method, Conditional Gradient

1 Introduction

Let $\mathbb{S}^n$ denote the set of $n \times n$ symmetric matrices, and define $\mathrm{tr}(A) = \sum_{i=1}^n A_{ii}$ for $A \in \mathbb{S}^n$. For $A, B \in \mathbb{S}^n$, we write $A \succeq B$ if $A - B$ is positive semidefinite, and define their Frobenius inner product as $A \bullet B = \mathrm{tr}(AB)$. We are interested in solving the following trace-bounded semidefinite program (SDP) and its dual:

$$P_* := \quad \min_X \quad \{C \bullet X \quad : \quad \mathcal{A}X = b, \quad X \in \Delta_\tau^n\} \tag{P}$$

and

$$D_* := \quad \max_{p \in \mathbb{R}^m, \theta \in \mathbb{R}_+} \quad \{-b^\top p - \tau^2 \theta \quad : \quad S := C + \mathcal{A}^*(p) + \theta I \succeq 0\} \tag{D}$$

where $b \in \mathbb{R}^m$, $C \in \mathbb{S}^n$, and $\mathcal{A} : \mathbb{S}^n \to \mathbb{R}^m$ and $\mathcal{A}^* : \mathbb{R}^m \to \mathbb{S}^n$ are the linear operators defined as

$$\mathcal{A}X := \begin{bmatrix} A_1 \bullet X \\ A_2 \bullet X \\ \vdots \\ A_m \bullet X \end{bmatrix}, \quad \mathcal{A}^*(p) := \sum_{\ell=1}^m p_\ell A_\ell, \quad \text{where } A_\ell \in \mathbb{S}^n \text{ for } \ell = 1, \dots m. \tag{1}$$

*__Funding__: Jacob M. Aguirre is supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-2039655. Diego Cifuentes is supported by U.S. Office of Naval Research, N00014-23-1-2631. Renato D.C. Monteiro is supported by AFOSR Grant FA9550-25-1-0131.

†H. M. Stewart School of Industrial and Systems Engineering, Georgia Tech, Atlanta, GA, 30332-0205. aguirre@gatech.edu
‡H. M. Stewart School of Industrial and Systems Engineering, Georgia Tech, Atlanta, GA, 30332-0205. dfc3@gatech.edu
§School of Applied Mathematics, FGV, Praia de Botafogo, Rio de Janeiro, Brazil. vincent.guigues@fgv.br
¶School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA, 30332-0205. monteiro@isye.gatech.edu
‖School of Applied Mathematics, FGV, Praia de Botafogo, Rio de Janeiro, Brazil. nascimento.victor.1@fgv.edu.br
**Department of Combinatorics and Optimization, University of Waterloo, Waterloo, ON, N2L 3G1. a3sujana@uwaterloo.ca

Authors are listed in alphabetical order.

We also define Δ_τ^n to be the spectraplex

$$\Delta_\tau^n := \{X \in \mathbb{S}^n : \mathrm{tr}(X) \leq \tau^2, X \succeq 0\}, \tag{2}$$

where the parameter $\tau > 0$ is supplied as part of the problem data.

Semidefinite programming arises across a wide spectrum of applications, notably in machine learning, computational sciences, and various engineering disciplines. Despite its versatility, efficiently solving large-scale SDPs remains computationally demanding. Interior-point methods, although widely employed, are often constrained by computational complexity as they often rely on solving a large linear system using a Newton-type method, which involves an expensive inversion of a large matrix [6, 3, 38, 56, 77]. These computational bottlenecks have sparked interest in developing efficient first-order methods for solving large-scale problems. Unlike interior-point methods, first-order methods scale well with the dimension of the problem instance and predominantly utilize sparse matrix-vector multiplications to perform function and gradient evaluations. For an extensive survey and analysis of recent developments in using first-order methods for solving SDPs, we refer the reader to the comprehensive literature available in [2, 20, 21, 23, 24, 28, 31, 40, 32, 47, 41, 60, 64, 67, 71, 83, 86, 87, 89, 92, 93].

Due to GPU's great capabilities in parallelizing matrix-vector multiplications, they have been shown to work extremely well in speeding up first-order methods in practice [16, 37, 48, 52, 51, 50, 69]. In this paper, we introduce an enhanced Julia-based GPU implementation, referred to as cuHALLaR, of the first-order HALLaR method developed in [55] for solving huge-scale SDPs. Our work reinforces the viability of using GPU architectures for efficient parallelization in semidefinite programming solvers. Our numerical experiments demonstrate substantial speedups on matrix completion, maximum stable set, and phase retrieval SDP relaxations. For these problem classes, suitable trace bounds τ can be computed from the problem data and must be supplied to the current implementation, which does not adjust τ automatically.

1.1 Related literature

We divide our discussion into four parts: first, the emerging literature on utilizing GPUs for mathematical optimization, second, the challenges in solving large SDPs, third, recent techniques in solving large SDPs using low-rank approaches, and fourth, a comparison between cuHALLaR, a recent GPU SDP solver cuLoRADS developed in [37], and the CPU version of HALLaR.

GPUs for Mathematical Programming GPUs offer massive parallelism that accelerates mathematical programming through parallel linear algebra operations. Recent GPU-based solvers for linear programming and quadratic programming [50, 52, 51] have achieved performance that outperforms traditional CPU-based methods. First-order methods map well to GPU architectures, unlike simplex or interior-point methods which rely on difficult-to-parallelize factorizations. For linear programming, cuPDLP.jl incorporates GPU-accelerated sparse matrix operations to match commercial solver performance. Quadratic programming solvers like cuOSQP [69] and CuClarabel [16] similarly exploit GPU parallelism for significant speedups.

Most recently, [48] proposed a new GPU primal-dual conic programming solver, namely, PDCS, which utilizes the primal-dual hybrid gradient (PDHG) method for solving several classes of cone optimization problems, including semidefinite programs; however, the authors of the paper remark that due to PDHG requiring Euclidean projections, their method for solving SDP problems is rather inhibited and slow. Thus, their code does not allow users to input SDP data, and so we cannot compare against PDCS.

Challenges in large-scale semidefinite programming Solving large-scale SDPs in practice presents greater challenges compared to solving large-scale linear programs and quadratic programs due to the reliance of many SDP solvers on full eigendecomposition, inversion, and storage of large dense matrices. To circumvent several of these issues, pre-processing techniques such as facial reduction, chordal conversion, and symmetry reduction have been proposed which can potentially convert the original SDP (P) into a smaller one or one with a more favorable structure [78, 30, 33, 91, 94, 90, 39, 7, 25, 73, 63]. Interior-point methods are then often applied to the smaller reformulated SDP. Another popular approach that is commonly used in practice to solve large-scale SDPs and which avoids storing and inverting large dense matrices is the low-rank approach first proposed by Burer and Monteiro [10, 11]. The next paragraph describes low-rank first-order methods for solving SDPs in more detail.

Low-rank first-order methods for SDPs The low-rank approach proposed by Burer and Monteiro [10, 11] converts SDP (P) into a nonconvex quadratic program with quadratic constraints (QCQP) through the transformation $X = UU^\top$, where $U \in \mathbb{R}^{n\times r}$ and $r \ll n$. The main advantage of this approach is that the QCQP has $n \times r$ variables,

while (P) has $n(n+1)/2$ variables. Also, the PSD constraint $X \succeq 0$ has been eliminated through the low-rank transformation so methods do not need to perform eigendecompositions of dense matrices to approximately solve the QCQP. Burer and Monteiro proposed an efficient first-order limited-memory BFGS augmented Lagrangian (AL) method, which only relied on matrix-vector multiplications and storage of $n \times r$ matrices, to find an approximate local minimizer of the QCQP. Recent landscape results have shown that if the factorization parameter r is taken large enough, then finding local minimizers of the QCQP is actually equivalent to finding global minimizers of the SDP (P) [17, 18, 8, 9, 5, 65, 79]. For a more extensive list of papers which characterize the optimization landscape of nonconvex formulations obtained through Burer-Monteiro factorizations, see [59, 27, 53, 49, 58, 22, 54, 46]. For recent advances and developments in accelerating low-rank first-order methods for solving SDPs, we refer the reader to the extensive literature available in [28, 36, 37, 55, 74, 75, 76, 81, 82, 88, 80, 87, 89].

The following comparison highlights the main differences between HALLaR, first developed in [55], and the LoRADS method developed in [36]. HALLaR utilizes an inexact AL method to solve (P). It solves a sequence of AL subproblems of the form $\arg\min_{X \in \Delta_r^n} \; C \bullet X + p^\top(\mathcal{A}X - b) + \beta\|\mathcal{A}X - b\|^2/2$ by first restricting them to the space of matrices of the form $X = UU^\top$, where the number of columns r of U is significantly smaller than n, and then applying a nonconvex solver to the reformulation. It uses a low-rank Frank-Wolfe step to escape from a possible spurious stationary point obtained by the nonconvex solver, a step which generally adds an additional column to U.

LoRADS, on the other hand, adopts a two-stage approach. In their first stage, they utilize the low-rank limited-memory BFGS AL method of Burer and Monteiro as a warm-start strategy for their second stage. The second stage of LoRADS also utilizes an inexact AL method applied to (P) but the major difference being the way the AL subproblems are reformulated. In contrast to the $X = UU^\top$ change of variables used by HALLaR, LoRADS reformulates them by replacing X by $UV^\top$ and adding the constraint $U = V$. It then approximately solves these nonconvex reformulations of the AL subproblems by an alternating minimization penalty-type approach which penalizes $U = V$ constraint and alternately minimizes with respect to the blocks U and V using the conjugate gradient method. Similar to HALLaR, the inexact stationary solutions of the nonconvex AL reformulations may not be near global solutions of the AL subproblems as they are not necessarily equivalent. In contrast to HALLaR, LoRADS uses a heuristic that increases the number of columns of U and V to attempt to escape from these spurious solutions.

cuHALLaR versus cuLoRADS and HALLaR Like cuLoRADS, which is a GPU accelerated version of LoRads developed in [37], cuHALLaR is also a GPU accelerated version of HALLaR. Computational results reported in this paper show that cuHALLaR is 2 to 25 times faster than cuLoRADS on large-scale instances of some important SDP classes. cuHALLaR can efficiently solve massive problems to 10^{-5} relative precision in just a few minutes. For example, cuHALLaR is able to solve a matrix completion SDP instance with matrix variable of size 8 million, and approximately 300 million constraints, in just 142 seconds.

cuHALLaR also achieves massive speedup compared to its CPU counterpart, HALLaR. cuHALLaR achieves speedups of 30-140x on large matrix completion SDP instances, up to 135x speedup on large maximum stable set SDP instances, and 15-47x speedup on phase retrieval SDP instances. These results demonstrate the strong performance of cuHALLaR on the large-scale structured SDPs considered in this paper.

1.2 Notation

Let $\mathbb{R}^n$ be the space of n dimensional vectors, $\mathbb{R}^{n\times r}$ the space of $n \times r$ matrices, and $\mathbb{S}^n$ the space of symmetric $n \times n$ matrices. Let $\mathbb{R}^n_{++}$ (resp., $\mathbb{R}^n_+$) denote the convex cone in $\mathbb{R}^n$ of vectors with positive (resp., nonnegative) entries and $\langle \cdot, \cdot \rangle$ (resp. $\|\cdot\|$) be the Euclidean inner product (resp. norm) on $\mathbb{R}^n$. The minimum eigenvalue of a matrix $Q \in \mathbb{S}^n$ is denoted by $\lambda_{\min}(Q)$, and $v_{\min}(Q)$ denotes a corresponding eigenvector of unit norm.

For a given $\epsilon \geq 0$, the ϵ-normal cone of a closed convex set $\mathcal{K}$ at $z \in \mathcal{K}$, denoted by $N^\epsilon_{\mathcal{K}}(z)$, is

$$N^\epsilon_{\mathcal{K}}(z) := \{\xi \in \mathbb{R}^n : \langle \xi, u - z \rangle \leq \epsilon, \quad \forall u \in \mathcal{K}\}.$$

The normal cone of a closed convex set $\mathcal{K}$ at $z \in \mathcal{K}$ is denoted by $N_{\mathcal{K}}(z) = N^0_{\mathcal{K}}(z)$. Finally, we define the Frobenius ball of radius d in $\mathbb{R}^{n\times s}$ space to be

$$B^s_d := \{U \in \mathbb{R}^{n\times s} : \|U\|_F \leq d\}. \tag{3}$$

We denote the i-th row of a matrix U by $U_{i,:}$. Given a matrix $X \in \mathbb{R}^{m\times n}$, we define

$$\|X\|_* = \sum_{i=1}^{\min(m,n)} \sigma_i(X) \tag{4}$$

as the nuclear norm, where $\sigma_i(X)$ represents the i-th singular value of X.

Finally, let $\mathbb{S}^n(\mathbb{C})$ denote the real vector space of complex Hermitian matrices:

$$\mathbb{S}^n(\mathbb{C}) = \{X \in \mathbb{C}^{n\times n} : X = X^H\},$$

where X^H denotes the conjugate transpose of X. For a complex scalar, vector, or matrix, $\overline{(\cdot)}$ denotes entrywise complex conjugation.

2 Overview of HALLaR

This section provides a minimal review of HALLaR, the hybrid low-rank augmented Lagrangian method that was first developed in [55]. The procedures described in this section are common to both HALLaR and the GPU accelerated version cuHALLaR.

As mentioned in the introduction, HALLaR is an inexact augmented Lagrangian method that approximately solves the pair of semidefinite programs (P) and (D). Given a tolerance pair $(\epsilon_p, \epsilon_{pd}) \in \mathbb{R}^2_{++}$, the method aims to find an $(\epsilon_p, \epsilon_{pd})$-optimal solution, i.e., a triple $(\bar{X}, \bar{p}, \bar{\theta}) \in \Delta^n_\tau \times \mathbb{R}^m \times \mathbb{R}_+$ that satisfies

$$\|\mathcal{A}\bar{X} - b\| \le \epsilon_p, \quad |C \bullet \bar{X} + b^\top \bar{p} + \tau^2\bar{\theta}| \le \epsilon_{pd}, \quad C + \mathcal{A}^*\bar{p} + \bar{\theta}I \succeq 0, \quad \bar{\theta} \ge 0. \tag{5}$$

When the tolerances are zero, these relations reduce to the standard optimality conditions for the primal-dual pair. Note that the final two inequalities in (5) enforce dual feasibility, a property that HALLaR maintains at every iteration. Being an inexact AL method, HALLaR generates sequences $\{X_t\}$ and $\{p_t\}$ according to the following updates

$$X_t \quad \approx \quad \arg\min_X \quad \{\mathcal{L}_{\beta_t}(X; p_{t-1}) \ : \ X \in \Delta^n_\tau\}, \tag{6a}$$

$$p_t \quad = \quad p_{t-1} + \beta_t(\mathcal{A}X_t - b) \tag{6b}$$

where

$$\mathcal{L}_\beta(X; p) \quad := \quad C \bullet X + p^\top(\mathcal{A}X - b) + \frac{\beta}{2}\|\mathcal{A}X - b\|^2 = [C + \mathcal{A}^*(p)] \bullet X + \frac{\beta}{2}\|\mathcal{A}X - b\|^2 \tag{7}$$

is the AL function, whose gradient with respect to X is given by

$$\nabla_X \mathcal{L}_\beta(X; p) := C + \mathcal{A}^* \left(p_{t-1} + \beta_t \left(\mathcal{A}X - b\right)\right).$$

The key part of HALLaR is an efficient method, called the hybrid low-rank method (HLR), for finding a near-optimal solution X_t of the AL subproblem (6a). HLR never forms X_t but outputs a low-rank factor $U_t \in B^{s_t}_\tau$ such that $X_t = U_t U_t^\top$, where $B^{s_t}_\tau$ is as in (3) and hopefully $s_t \ll n$. Given a pair $(s, \tilde{Y}) \in \mathbb{Z}_+ \times B^s_\tau$, which is initially set to $(s, \tilde{Y}) = (s_{t-1}, U_{t-1})$, a general iteration of HLR performs the following steps:

i) it uses a nonlinear solver to compute a suitable stationary point Y of the nonconvex reformulation of (6a) obtained through the change of variable $X = UU^\top$, namely:

$$\min_U \quad \left\{g_t(U) := \mathcal{L}_{\beta_t}(UU^\top; p_{t-1}) \quad : \quad \|U\|_F \le \tau, \quad U \in \mathbb{R}^{n\times s}\right\}; \tag{8}$$

ii) it verifies if the point $X = YY^\top$ corresponds to a near-global minimizer of the original convex AL subproblem (6a). This check is performed by computing the gradient G of the AL function at $X = YY^\top$ and using its minimum eigenpair to compute a Frank-Wolfe optimality gap. If this gap is below a prescribed tolerance, the point is accepted as a suitable solution of (8), and HLR stops, setting $(s_t, U_t) = (s, Y)$;

iii) otherwise, if the gap is not small then this indicates that Y is a spurious stationary point of (8). The minimum eigenvector computed in the previous step furnishes a descent direction for the convex problem (6a). Using this direction, a Frank-Wolfe step is performed in the X-space (but implemented in the U-space) to escape the spurious point Y. This step results in a rank-one update that produces a new iterate pair $(s, \tilde{Y})$.

We now give more details about the above steps. First, we consider step i). Given a tolerance $\rho > 0$ and $\tilde{Y} \in \mathbb{R}^{n\times s}$, this step is implemented with the aid of a generic nonlinear solver which, started from $\tilde{Y}$, computes a stationary solution $(Y; R) \in \mathbb{R}^{n\times s} \times \mathbb{R}^{n\times s}$ of (8) satisfying

$$R \in \nabla g_t(Y) + N_{B^s_\tau}(Y), \quad \|R\|_F \le \rho, \quad g_t(Y) \le g_t(\tilde{Y}). \tag{9}$$

In view of (7) and the definition of $g_t(\cdot)$ in (8), $\nabla g_t(\cdot)$ is given by

$$\nabla g_t(U) = 2[\nabla_X \mathcal{L}_\beta(UU^\top; p)]U = 2[C + \mathcal{A}^*(p_{t-1} + \beta_t(\mathcal{A}(UU^\top) - b))]U, \quad \forall U \in \mathbb{R}^{n\times s}. \tag{10}$$

Appendix A describes a method that is able to carry out step i), namely, ADAP-AIPP, whose exact formulation and analysis can be found in [55, 72], and also in earlier works (see e.g. [15, 61, 42, 43]) in other related forms. ADAP-AIPP typically requires several evaluations of both $g_t(U)$ and $\nabla g_t(U)$. The main computational effort in these evaluations lies in computing $[C + \mathcal{A}^*(p)]U$ and $\mathcal{A}(UU^\top)$, where $p := p_{t-1} + \beta_t(\mathcal{A}(UU^\top) - b)$.

The details of steps ii) and iii) in the U-space are described in the formal description of the algorithm given below. It is worth observing that at the end of step iii), the low-rank parameter s is set to either $s = 1$ (unlikely case) or $s = s + 1$ (usual case), due to the rank-one update nature of the Frank-Wolfe step (see line 12 of HALLaR stated below). Hence, in the usual case, the number of columns of $\tilde{Y}$ increases by one. In our experiments, however, HLR solves many AL subproblems in a single iteration, stopping at step (ii). The number of columns of Y remains fixed at s throughout steps (i) and (ii), so in these cases U_t has the same number of columns as U_{t-1}. This helps explain why the iterates generated by our method tend to have low rank. The description of HALLaR below assumes that the computation of the minimum eigenpairs are exact.

Algorithm 1 HALLaR.

1: **Input:** An integer $s_0 \geq 1$; initial points $(U_0, p_0) \in B^{s_0}_\tau \times \mathbb{R}^m$; a tolerance pair $(\epsilon_{\rm p}, \epsilon_{\rm pd}) \in \mathbb{R}^2_{++}$; a sequence $\{\beta_t\}_{t\geq 1}$ of positive reals; and a decreasing sequence $\{\epsilon_t\}_{t\geq 1}$ of positive reals converging to zero.

2: **Output:** Upon termination, a triple $(\bar{U}, \bar{p}, \bar{\theta})$ such that, with $\bar{X} := \bar{U}\bar{U}^\top$, $(\bar{X}, \bar{p}, \bar{\theta}) \in \Delta^n_\tau \times \mathbb{R}^m \times \mathbb{R}_+$ is an $(\epsilon_{\rm p}, \epsilon_{\rm pd})$-solution of (P) and (D), in the sense of (5).

3: **for** $t = 1, 2, \ldots$ **do**

4: set $(\tilde{Y}, s) \leftarrow (U_{t-1}, s_{t-1})$;

5: **while true do** ▷ **HLR method**

6: call a nonconvex solver, initialized at $\tilde{Y}$, to compute (Y, R), with $Y \in B^s_\tau$, satisfying (9) with $\rho = \epsilon_t$ for

$$\min_{U\in\mathbb{R}^{n\times s}} \left\{ g_t(U) := \mathcal{L}_{\beta_t}(UU^\top; p_{t-1}) \ : \ \|U\|_F \leq \tau \right\}; \tag{11}$$

7: compute a minimum eigenpair $(\lambda_{\min}(G), v_{\min}(G))$ with $\|v_{\min}(G)\| = 1$ of

$$G = G(Y) := \nabla_X \mathcal{L}_{\beta_t}(YY^\top; p_{t-1}) \in \mathbb{S}^n, \tag{12}$$

and the quantities

$$\gamma = G \bullet (YY^\top), \quad \theta := \max\{-\lambda_{\min}(G), 0\}, \qquad y := \begin{cases} v_{\min}(G), & \text{if } \theta > 0, \\ 0, & \text{otherwise;} \end{cases} \tag{13}$$

8: **if** $\gamma + \tau^2\theta \leq \epsilon_t$ **then**

9: **break the HLR loop**;

10: **end if**

11: compute

$$\kappa = \beta_t \|\mathcal{A}(YY^\top) - \tau^2 \mathcal{A}(yy^\top)\|^2$$

12: and set

$$\alpha := \begin{cases} \min\left\{ \dfrac{\gamma + \tau^2\theta}{\kappa}, 1 \right\}, & \text{if } \kappa > 0, \\ 1, & \text{otherwise}, \end{cases} \tag{14}$$

$$(\tilde{Y}, s) \leftarrow \begin{cases} (\tau y, 1), & \text{if } \alpha = 1, \\ \left(\left[\sqrt{1-\alpha}\, Y, \ \sqrt{\alpha}\, \tau y \right], s + 1 \right), & \text{otherwise;} \end{cases} \tag{15}$$

13: **end while**

14: set $(U_t, \theta_t, s_t) \leftarrow (Y, \theta, s)$;

15: set

$$p_t := p_{t-1} + \beta_t \left(\mathcal{A}(U_t U_t^\top) - b \right); \tag{16}$$

16: if

$$\|\mathcal{A}(U_t U_t^\top) - b\| \leq \epsilon_{\rm p}, \qquad \left| C \bullet (U_t U_t^\top) + b^\top p_t + \tau^2 \theta_t \right| \leq \epsilon_{\rm pd}, \tag{17}$$

then **return** $(\bar{U}, \bar{p}, \bar{\theta}) = (U_t, p_t, \theta_t)$;

17: **end for**

Several remarks about each of the steps of HALLaR are now given. First, t is the outer iteration count for HALLaR. Second, the while loop in lines 5–13 of HALLaR corresponds to the HLR method, which HALLaR uses to approximately solve the AL subproblems in (6a). Third, the main computational costs of each HLR iteration are the gradient evaluations required by the nonconvex solver in step 6 and the minimum eigenvector (MEV) computation in step 7. Fourth, after the HLR method approximately solves an AL subproblem, HALLaR updates the Lagrange multiplier in line 15 and checks for the termination condition in line 16. Fifth, the termination condition does not check for near dual feasibility since it can be shown that for every $t \geq 1$,

$$S_t := C + \mathcal{A}^* p_t + \theta_t I \succeq 0, \quad \theta_t \geq 0, \tag{18}$$

and hence that HALLaR always generates feasible dual iterates (p_t, θ_t, S_t). Finally, the penalty and tolerance sequences β_t and ϵ_t are kept general in the presentation of HALLaR. In practice, we choose the penalty sequence $\{\beta_t\}_{t\geq 1}$ adaptively, drawing on ideas from the LANCELOT method [19].

We conclude this section by summarizing the convergence guarantees of HALLaR established in [55]. Suppose that strong duality holds between (P) and (D), i.e., they have the same finite optimal value and (D) has an optimal solution. The compactness of the feasible region of (P) implies that (P) also has an optimal solution. Given a tolerance $\epsilon > 0$ and setting $\beta_t = \beta$ for all $t \geq 1$ where $\beta = \mathcal{O}(1/\epsilon)$, it is shown in Corollary 3.3 that HALLaR obtains a triple $(\bar{U}, \bar{p}, \bar{\theta})$ satisfying

$$\begin{aligned}
&(\epsilon\text{-primal feasibility}) && \|\mathcal{A}(\bar{U}\bar{U}^\top) - b\| \leq \epsilon, \quad \mathrm{tr}(\bar{U}\bar{U}^\top) \leq \tau^2, \quad \bar{U}\bar{U}^\top \succeq 0, && (19)\\
&(\text{dual feasibility}) && \bar{S} := C + \mathcal{A}^*\bar{p} + \bar{\theta} I \succeq 0, \quad \bar{\theta} \geq 0, && (20)\\
&(\epsilon\text{-complementarity}) && \langle \bar{U}\bar{U}^\top, \bar{S}\rangle + \bar{\theta}(\tau^2 - \mathrm{tr}(\bar{U}U^\top)) \leq \epsilon
\end{aligned}$$

in $\mathcal{O}(1/\epsilon^2)$ total number of HLR iterations and MEV computations. Moreover, it follows from Lemma 3.5 of [55] that the multiplier sequence $\{p_t\}$ is bounded. Using the above two observations, it can be shown that HALLaR also requires $\mathcal{O}(1/\epsilon^2)$ total number of HLR iterations and MEV computations to find a triple $(\bar{U}, \bar{p}, \bar{\theta}) = (U_t, p_t, \theta_t)$ satisfying (17) with $\epsilon_{\mathrm{p}} = \epsilon_{\mathrm{pd}} = \epsilon$.

3 Computational Aspects of cuHALLaR

GPU architecture is designed to perform many operations through massive parallelism, using a single instruction multiple data (SIMD) model. Unlike CPUs, which have fewer cores and are optimized for sequential execution and control, GPUs have thousands of simpler cores that are organized in streaming multiprocessors (SMs). GPUs possess a memory hierarchy similar to CPUs with fast and small registers for each SM, a shared memory between blocks of SMs, and a slower and bigger global GPU memory.

In this section, we discuss the key computational factors that allow cuHALLaR to achieve significant speedups over HALLaR through the use of GPU programming. This section is divided into two subsections. The first subsection discusses effectively dividing tasks between the CPU and the GPU. The second subsection describes our new way of efficiently reading and inputting SDP data that is suitable for GPU programming.

3.1 Effective Division of CPU and GPU Tasks

The primary computational concern in GPU programming is effectively dividing tasks between the CPU and GPU to exploit the advantages of both architectures. Highly parallelizable operations, such as matrix, vector, and tensor operations, should be implemented on the GPU as functions that are called kernels. On the other hand, serial tasks should be implemented as regular functions in the CPU. Hence, the primary computational bottleneck in modern GPUs is the possibly large number of data transfers that need to be performed between the CPU and the GPU.

Another important computational consideration in GPU programming is that, like in the CPU, launching threads in the GPU also incurs an overhead due to thread management and communication. Therefore, an efficient GPU implementation must minimize the number of data transfers and the number of kernels launched.

To minimize CPU and GPU communication overhead, cuHALLaR pre-allocates essential problem data from (P) on the GPU, including the objective matrix C, the linear operator $\mathcal{A}$, and the right-hand side vector b. All subsequent computations involving large matrices and vectors are performed exclusively on the GPU.

3.2 Reading Data Efficiently and Input Format

SDPA format. This format, introduced in [84, 85, 29], allows for the specification of SDPs using sparse matrix representations of the cost and constraint matrices. Each matrix is stored by explicitly listing all nonzero entries

in its upper triangular part, which is efficient for problems where all data matrices are sparse. However, the SDPA format is not suitable for handling instances with dense but well-structured data matrices. For example, if C is the matrix of all ones, then SDPA format requires the storage of $n(n+1)/2$ entries, and hence does not take advantage of the fact that such a matrix can be represented by either a single scalar or a single n-vector. Nevertheless, cuHALLaR also accepts input in the sparse SDPA format, given its widespread use among SDP solvers.

HSLR format. To address the issues of the SDPA format and to improve handling of large-scale problems, we introduce a new input format, namely, Hybrid Sparse Low-Rank (HSLR) format, where each constraint matrix A_ℓ (with the convention $C = A_0$) can be decomposed as:

$$A_\ell = A_\ell^{\text{sp}} + A_\ell^{\text{lr}}, \qquad A_\ell^{\text{lr}} = K_\ell D_\ell K_\ell^\top, \qquad \ell = 0, 1, \ldots, m \tag{21}$$

where $A_\ell^{\text{sp}} \in \mathbb{S}^n$ is a sparse symmetric matrix, $K_\ell \in \mathbb{R}^{n\times r_\ell}$, and $D_\ell \in \mathbb{S}^{r_\ell}$ is a symmetric matrix that is not necessarily diagonal. This structure allows us to reformulate the core operations of cuHALLaR in terms of sparse matrix algebra and small dense matrix multiplications. In particular, cuHALLaR never has to form any $n \times n$ matrices. If the input is provided in SDPA format, the sparse matrices are stored and handled as $A_\ell = A_\ell^{\text{sp}}$. For examples on how to construct HSLR format for several structured SDP instances such as the SDP relaxations of the matrix completion and maximum stable set problems, the reader should refer to our user manual [1].

The advantages of HSLR over sparse SDPA format are twofold:

1. **Storage Efficiency**: In SDPA format, a dense $n \times n$ matrix is stored explicitly using $\mathcal{O}(n^2)$ entries, even when it admits a low-rank representation. HSLR stores the low-rank component A_ℓ^{lr} through its factors K_ℓ and D_ℓ, requiring $\mathcal{O}(nr_\ell + r_\ell^2)$ entries in total. If D_ℓ is diagonal, only its r_ℓ diagonal entries are stored, giving a total storage cost of $\mathcal{O}(nr_\ell + r_\ell)$ for the low-rank component. The sparse component A_ℓ^{sp} is stored in a coordinate (COO)-style format that records the constraint index, row index, column index, and value of each nonzero entry. A sparse component with n_ℓ nonzero entries therefore requires $\mathcal{O}(n_\ell)$ storage.

2. **Computational Efficiency**: During cuHALLaR's execution, matrix-vector products and trace evaluations are computed by separately handling the sparse and low-rank components of each data matrix A_ℓ. Since cuHALLaR handles these components separately, it never has to form the possibly dense matrix A_ℓ and calls different kernels based on the specific structure of each A_ℓ.

4 Efficient Implementation of cuHALLaR

This section describes how cuHALLaR exploits the sparse and low-rank components of (21) to perform its basic operations without forming dense $n \times n$ matrices. We first formalize the constraint and adjoint evaluations, and then describe the minimum eigenpair computation.

To treat the objective and constraint evaluations in a unified manner, set $A_0 := C$ and augment the operators in (1) by defining $\tilde{\mathcal{A}} : \mathbb{S}^n \to \mathbb{R}^{m+1}$ and its adjoint $\tilde{\mathcal{A}}^* : \mathbb{R}^{m+1} \to \mathbb{S}^n$ as

$$\tilde{\mathcal{A}}X := \begin{bmatrix} C \bullet X \\ \mathcal{A}X \end{bmatrix}, \qquad \tilde{\mathcal{A}}^*(\tilde{p}) := \sum_{\ell=0}^{m} \tilde{p}_\ell A_\ell. \tag{22}$$

Clearly, for any $p \in \mathbb{R}^m$, (22) implies that $\tilde{\mathcal{A}}^*(\tilde{p}) = C + \mathcal{A}^*(p)$ where $\tilde{p} = (1, p) \in \mathbb{R}^{m+1}$. Recall from the low-rank reformulation (8) that cuHALLaR represents each primal iterate as $X = UU^\top$ where $U \in \mathbb{R}^{n\times r}$, and operates directly on the factor U.

Every cuHALLaR iteration has to perform the operations

$$\tilde{\mathcal{A}}(UU^\top), \qquad \tilde{\mathcal{A}}^*(\tilde{p})U, \tag{23}$$

for pairs $(U, \tilde{p}) \in \mathbb{R}^{n\times r} \times \mathbb{R}^{m+1}$, possibly with $r = 1$. Subsections 4.1 and 4.2 describe how cuHALLaR performs the two operations in (23). cuHALLaR uses the first one to evaluate the quantity $g_t(U)$ in (8) and both of them to compute $\nabla g_t(U)$ at the U-iterates generated by ADAP-AIPP. Specifically, it follows from (10) and the remark following (22) that $\nabla g_t(U)$ can be computed as

$$\nabla g_t(U) \overset{(22)}{=} 2\,\tilde{\mathcal{A}}^*(\tilde{p})U, \qquad \tilde{p} = \big(1,\, p_{t-1} + \beta_t(\mathcal{A}(UU^\top) - b)\big) \in \mathbb{R}^{m+1},$$

and hence without the need to compute the matrix $G(U) \in \mathbb{S}^n$ as in (12).

The minimum eigenpairs are calculated using the Hermitian Lanczos method implemented in the KrylovKit Julia package [35], with absolute residual tolerance 10^{-10}. KrylovKit computes a minimum eigenpair of the matrix $G = G(Y)$ in (12) by performing matrix–vector products Gv for several generated vectors v. Observe that $Gv = \tilde{\mathcal{A}}^*(\tilde{p})v$ is a special case of the second operation in (23) with U consisting of the single column vector v. Table 8 in Appendix B.1 presents a performance profile of the MEV computations for four problem instances from Section 5. These computations account for 54.7–79.9% of the total execution time of cuHALLaR.

4.1 Efficient Computation of $\tilde{\mathcal{A}}(UU^\top)$

This subsection describes how cuHALLaR efficiently computes $\tilde{\mathcal{A}}(UU^\top)$, where $U \in \mathbb{R}^{n\times r}$ and $\tilde{A}(\cdot)$ is as in (23). It follows from (21) and the first relation in (1) that $T = \tilde{\mathcal{A}}(UU^\top)$ can be computed as $T = T^{sp} + T^{lr}$ where the ℓ-th components of T^{sp} and T^{lr} are defined as:

$$T^{sp}_\ell = A^{sp}_\ell \bullet (UU^\top), \quad T^{lr}_\ell = A^{lr}_\ell \bullet (UU^\top), \quad \ell = 0, 1, \ldots, m. \tag{24}$$

In practice, cuHALLaR implements efficient subroutines to separately compute T^{sp} and T^{lr}. The rest of this subsection discusses in more detail how cuHALLaR computes these two quantities efficiently.

Sparse Component. This paragraph describes how cuHALLaR efficiently computes $T^{sp} \in \mathbb{R}^{m+1}$. Let $B = UU^\top$. It is easy to see that the entries of B can be computed as $B_{ij} = \langle U_{i,:}, U_{j,:}\rangle$, where $U_{i,:}$ and $U_{j,:}$ are the i-th and j-th rows of U, respectively. The ℓ-th component of $T^{\rm sp}$, $T^{\rm sp}_\ell = A^{\rm sp}_\ell \bullet (UU^\top)$, is then computed as:

$$T^{\rm sp}_\ell = \sum_{i=1}^{n}\sum_{j=1}^{n} (A^{\rm sp}_\ell)_{ij} B_{ij}. \tag{25}$$

Since $A^{\rm sp}_\ell$ and B are symmetric matrices, this summation can be rewritten by defining the weights

$$\omega_{ij,\ell} := (2 - \delta_{ij})(A^{\rm sp}_\ell)_{ij}, \text{ where } \delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \tag{26}$$

Using (26), we see that $T^{\rm sp}_\ell = \sum_{i\le j} \omega_{ij,\ell} B_{ij}$.

To avoid redundant computations of B_{ij} across different constraints ℓ, we reorganize this computation by using the union of the supports of A^{sp}_ℓ. This union $\mathcal{S}$ is formally defined as:

$$\mathcal{S} := \bigcup_{\ell=0}^{m} \operatorname{supp}(A^{\rm sp}_\ell) \cap \{(i,j) : 1 \le i \le j \le n\}. \tag{27}$$

For each pair $(i,j) \in \mathcal{S}$, we define $\mathcal{L}_{ij}$ to be the set of indices ℓ for which the entry (i,j) is active:

$$\mathcal{I}_{ij} := \{\ell : (i,j) \in \operatorname{supp}(A^{\rm sp}_\ell)\}. \tag{28}$$

To compute T^{sp}, we then compute the quantity B_{ij} over each pair $(i,j) \in \mathcal{S}$ and update the corresponding entries T^{sp}_ℓ for all $\ell \in \mathcal{I}_{ij}$. This procedure is formally described in Algorithm 2 below.

Algorithm 2 Computation of $\tilde{\mathcal{A}}(UU^\top)$ for the sparse component, where $\tilde{\mathcal{A}}(\cdot)$ is as in (23).

Input: $U \in \mathbb{R}^{n\times r}$, data matrices $\{A^{\rm sp}_\ell\}_{\ell=0}^m \subseteq \mathbb{S}^n$, and precomputed structures $\mathcal{S}$ and $\{\mathcal{I}_{ij}\}$ from (27)–(28).
Output: $T^{sp} \in \mathbb{R}^{m+1}$ where $T^{sp}_\ell = A^{sp}_\ell \bullet UU^\top$.
Initialize $T^{sp} \leftarrow 0 \in \mathbb{R}^{m+1}$.
for all $(i,j) \in \mathcal{S}$ in parallel **do**
    $B_{ij} \leftarrow \langle U_{i,:}, U_{j,:}\rangle$
    for all $\ell \in \mathcal{I}_{ij}$ **do**
        $\omega_{ij,\ell} \leftarrow (2 - \delta_{ij})(A^{\rm sp}_\ell)_{ij}$
        $T^{sp}_\ell \leftarrow T^{sp}_\ell + \omega_{ij,\ell} B_{ij}$ ▷ Atomic update
    end for
end for
return T^{sp}

In Algorithm 2, the outer loop over the set $\mathcal{S}$ is parallelized. More specifically, each unique index pair $(i,j) \in \mathcal{S}$ is assigned to a separate GPU thread. Each thread performs a loop over $r \ll n$ to compute B_{ij} and a loop over $|\mathcal{I}_{ij}| \ll m+1$ that updates T_ℓ^{sp}. Since different pairs (i,j) may contribute to the same entry T_ℓ^{sp}, the update in Line 8 is performed atomically. By this, we mean that the operation

$$T_\ell^{sp} \leftarrow T_\ell^{sp} + \omega_{ij,\ell} B_{ij}. \tag{29}$$

is carried out by the hardware as a protected read–add–write operation: a thread reads the current value of T_ℓ^{sp}, adds its contribution, and writes back the updated value without another thread being allowed to modify the same memory location in the middle of this operation. Thus, if several threads attempt to update the same entry T_ℓ^{sp} concurrently, their additions are effectively serialized for that entry, ensuring that no contribution is overwritten or lost. Threads acting on different entries, and the remaining computations in Algorithm 2, can still be executed in parallel.

Low-Rank Component. This paragraph describes how cuHALLaR efficiently computes $T^{\mathrm{lr}} \in \mathbb{R}^{m+1}$. Using the second relation in (21), the ℓ-th component of T^{lr}, $T_\ell^{\mathrm{lr}} = A_\ell^{\mathrm{lr}} \bullet (UU^\top)$, can be computed as

$$T_\ell^{\mathrm{lr}} = \mathrm{tr}(K_\ell D_\ell K_\ell^\top (UU^\top)) = \mathrm{tr}(D_\ell (K_\ell^\top U)(U^\top K_\ell)). \tag{30}$$

Defining $Y_\ell := K_\ell^\top U \in \mathbb{R}^{r_\ell \times r}$, relation (30) further simplifies to:

$$T_\ell^{\mathrm{lr}} = \mathrm{tr}(D_\ell Y_\ell Y_\ell^\top). \tag{31}$$

Algorithm 3 below formally describes our efficient procedure for computing T^{lr}.

Algorithm 3 Computation of $\tilde{\mathcal{A}}(UU^\top)$ for the low-rank component, where $\tilde{\mathcal{A}}(\cdot)$ is as in (23)

1: **Input:** $U \in \mathbb{R}^{n\times r}$ and problem data $\{K_\ell, D_\ell\}_{\ell=0}^m$.
2: **Output:** $T^{\mathrm{lr}} \in \mathbb{R}^{m+1}$ where $T_\ell^{\mathrm{lr}} = A_\ell^{\mathrm{lr}} \bullet UU^\top$.
3: Initialize $T^{\mathrm{lr}} \leftarrow 0 \in \mathbb{R}^{m+1}$.
4: **for** $\ell = 0$ to m **do**
5: $\quad Y_\ell \leftarrow K_\ell^\top U$ ▷ in parallel
6: $\quad T_\ell^{\mathrm{lr}} \leftarrow 0 + \mathrm{tr}(D_\ell Y_\ell Y_\ell^\top)$
7: **end for**
return T^{lr}

In Algorithm 3, the loop over $\ell = 0, \ldots, m$ is performed in serial, while the computations of Y_ℓ and $\mathrm{tr}(D_\ell Y_\ell Y_\ell^\top)$ are performed in parallel. In our numerical experiments, we found that parallelizing these computations was very efficient.

4.2 Efficient Computation of $\tilde{\mathcal{A}}^*(\tilde{p})U$

This subsection details how, for a given $(U,p) \in \mathbb{R}^{n\times r} \times \mathbb{R}^m$, cuHALLaR efficiently computes the quantity $W = \tilde{\mathcal{A}}^*(\tilde{p})U$ as in (23), where $\tilde{p} = (1,p) \in \mathbb{R}^{m+1}$. It follows from (21) and the second identity in (23) that W is given by

$$W = \underbrace{\left(\sum_{\ell=0}^m \tilde{p}_\ell A_\ell^{\mathrm{sp}}\right) U}_{W^{\mathrm{sp}}} + \underbrace{\sum_{\ell=0}^m \tilde{p}_\ell (K_\ell D_\ell K_\ell^\top) U}_{W^{\mathrm{lr}}}. \tag{32}$$

In cuHALLaR, efficient subroutines compute W^{sp} and W^{lr} separately.

Sparse Component. To compute W^{sp} as in (32), we first allocate a sparse symmetric matrix M with nonzero entries at indices (i,j), where $\mathcal{I}_{ij}$ in (28) is nonempty. After doing this allocation, we then update the values of the nonzero entries of M_{ij} in parallel using the formula:

$$M_{ij} = \sum_{\ell \in \mathcal{I}_{ij}} \tilde{p}_\ell (A_\ell^{\mathrm{sp}})_{ij}. \tag{33}$$

The matrix W^{sp} is then computed as $W^{\mathrm{sp}} = MU$. This procedure is formally described in Algorithm 4 below.

Algorithm 4 Computation of W^{sp} as in (32).

1: **Input:** $U \in \mathbb{R}^{n\times r}$, $\tilde{p} = (1,p) \in \mathbb{R}^{m+1}$ where $p \in \mathbb{R}^m$, $\{A_\ell^{\rm sp}\}_{\ell=0}^m \subseteq \mathbb{S}^n$, and precomputed structures $\mathcal{S}$, $\{\mathcal{I}_{ij}\}$ from (27)–(28).
2: **Output:** $W^{sp} \in \mathbb{R}^{n\times r}$ where $W^{sp} = (\sum_{\ell=0}^m \tilde{p}_\ell A_\ell^{\rm sp})\, U$.
3: Initialize $M \in \mathbb{S}^n$ with pattern $\mathcal{S} \cup \{(j,i) : (i,j) \in \mathcal{S}\}$.
4: **for all** $(i,j) \in \mathcal{S}$, in parallel **do**
5: $\quad w \leftarrow 0$
6: $\quad$ **for all** $\ell \in \mathcal{I}_{ij}$ **do**
7: $\quad\quad w \leftarrow w + \tilde{p}_\ell (A_\ell^{\rm sp})_{ij}$
8: $\quad$ **end for**
9: $\quad M_{ij} \leftarrow w$
10: $\quad$ **if** $i \neq j$ **then**
11: $\quad\quad M_{ji} \leftarrow w$
12: $\quad$ **end if**
13: **end for**
14: $W^{sp} \leftarrow MU$
return W^{sp}

Several remarks about Algorithm 4 are now given. The matrix M, which is initialized in step 3, is stored in sparse format with symmetric pattern $\mathcal{S} \cup \{(j,i) : (i,j) \in \mathcal{S}\}$, hence it contains at most $2|\mathcal{S}| - |\{\, i : (i,i) \in \mathcal{S}\,\}|$ nonzero entries. Also, it is worthwhile to note that the outer for loop over $\mathcal{S}$ is parallelized. More specifically, each unique index pair $(i,j) \in \mathcal{S}$ is assigned to a separate GPU thread. Each thread then performs a loop over $|\mathcal{I}_{ij}| \ll m+1$ to compute $w = \sum_{\ell\in\mathcal{I}_{ij}} \tilde{p}_\ell (A_\ell^{\rm sp})_{ij}$ and updates the corresponding entries in M.

Low-Rank Component. This paragraph describes how cuHALLaR efficiently computes $W^{lr} \in \mathbb{R}^{n\times r}$. Using the second quantity in (32) and defining $Y_\ell = K_\ell^\top U \in \mathbb{R}^{r_\ell \times r}$, it follows that $W^{\rm lr}$ can be computed as

$$W^{\rm lr} = \sum_{\ell=0}^m \tilde{p}_\ell K_\ell (D_\ell Y_\ell). \tag{34}$$

Algorithm 5 below formally describes our efficient procedure for computing $W^{\rm lr}$.

Algorithm 5 Computation of W^{lr} as in (32).

1: **Input:** $U \in \mathbb{R}^{n\times r}$, $\tilde{p} = (1,p) \in \mathbb{R}^{m+1}$ where $p \in \mathbb{R}^m$, and problem data $\{K_\ell, D_\ell\}_{\ell=0}^m$.
2: **Output:** $W^{lr} \in \mathbb{R}^{n\times r}$ where $W^{lr} = \left(\sum_{\ell=0}^m \tilde{p}_\ell A_\ell^{\rm lr}\right) U$.
3: Initialize $W^{lr} \leftarrow 0_{n\times r}$.
4: **for** $\ell = 0$ to m **do**
5: $\quad Y_\ell \leftarrow K_\ell^\top U$
6: $\quad Z_\ell \leftarrow D_\ell Y_\ell$ ▷ in parallel
7: $\quad W^{lr} \leftarrow W^{lr} + \tilde{p}_\ell (K_\ell Z_\ell)$
8: **end for**
return W^{lr}

In Algorithm 5, the loop over $\ell = 0, \ldots, m$ is performed in serial, while the computations of $Y_\ell = K_\ell^\top U$ and $Z_\ell = D_\ell Y_\ell$ used to update W^{lr} are performed in parallel.

5 Numerical Experiments

In this section, we present extensive numerical experiments comparing cuHALLaR, HALLaR, and cuLoRaDs on three important classes of SDPs: matrix completion, maximum stable set, and phase retrieval. CuClarabel [16] was not included in our numerical comparisons because its GPU implementation supports only PSD cones of dimension at most 32, whereas our experiments involve substantially larger PSD cones.

Solver and Software Details The experiments use precompiled binaries for all three solvers. The HALLaR and cuHALLaR binaries are available from the `https://github.com/OPTHALLaR` repository, and the cuLoRADS binary is available from the `https://github.com/COPT-Public/cuLoRADS` repository. Our cuHALLaR implementation is a Julia module that uses CUDA to perform computations on NVIDIA GPUs. All experiments are conducted with

Julia 1.11.3 and CUDA 12.6.1. The CPU times in all the tables below quantify solve time only and hence do not include the time to precompile and/or read the data.

Hardware Our experiments compare the GPU methods cuHALLaR and cuLoRADS, and the CPU method HALLaR. The GPU experiments are performed on an NVIDIA H200 with 142 GB VRAM deployed on a cluster with an Intel Xeon Platinum 8469C CPU. For CPU benchmarks, we use a single Dual Intel Xeon Gold 6226 CPUs @ 2.7 GHz (24 cores/node). Further details are given in Table 1. We set the memory of the CPU benchmarks at 142GB, which is equal to the maximum memory of the GPU.

Specification	**CPU Node**	**GPU**
Processor(s)	Dual Intel Xeon Gold 6226 @ 2.70 GHz	NVIDIA H200
Cores / SMs	24 cores	80 SMs
Cache (per CPU)	L3: 19.25 MB	
(per core)	L2: 1 MB; L1: 32 KB (D+I)	L1/Shared (per SM), L2
Peak FP64 Perf.	∼2.07 TFLOPS	∼34 TFLOPS
Memory Size	142 GB DDR4	142 GB HBM3e
Memory Bandwidth	∼140.7 GB/s	4.8 TB/s

Table 1: Comparison of CPU and GPU specifications used in experiments.

Termination criteria. Given a tolerance $\epsilon > 0$, both solvers stop when a set of relative error metrics falls below ϵ. For cuHALLaR, an iterate $(X, p, \theta) \in \Delta_\tau^n \times \mathbb{R}^m \times \mathbb{R}_+$ is deemed optimal if it satisfies

$$\max\left\{\frac{\|\mathcal{A}X - b\|_2}{1 + \|b\|_1}, \frac{|\text{pval} - \text{dval}|}{1 + |\text{pval}| + |\text{dval}|}, \frac{\max\{0, -\lambda_{\min}(S)\}}{1 + \|C\|_1}\right\} \leq \epsilon, \tag{35}$$

where $\text{pval} = C \bullet X$ is the primal value, $S = C + \mathcal{A}^* p + \theta I$ is the dual slack matrix, and the dual value is $\text{dval} = -b^\top p - \tau^2 \theta$. The solver cuLoRADS employs a similar set of metrics; however, as it addresses a formulation without the spectraplex constraint, its dual value is defined as $\text{dval} = -b^\top p$.

Input format. cuHALLaR binary accepts two different input formats: HSLR and SDPA formats. For SDP data prepared in SDPA format, the user must also supply the trace bound $\tau > 0$ in (2) for the constraint $\text{tr}(X) \leq \tau^2$ externally via the command-line interface or a parameter file. On the other hand, HSLR embeds the trace bound τ directly within the file itself, so that the user does not need to supply it separately. For more details and examples of preparing data in SDPA and HSLR format, the reader should refer to cuHALLaR's user manual [1].

Our computational experiments also report results for on-the-fly HALLaR and cuHALLaR versions (not available to the public) where the data are generated in the memory. Specifically, instead of reading the problem data from an input file, these versions use subroutines for computing $\tilde{\mathcal{A}}(UU^\top)$ and $\tilde{\mathcal{A}}^*(\tilde{p})U$ for any $\tilde{p} = (1, p) \in \mathbb{R}^{m+1}$ and $U \in \mathbb{R}^{n \times r}$. Although these versions are not crucial for our experiments for the matrix completion and maximum stable set problems, they play an important role in the context of the phase-retrieval experiments as they significantly save data storage (see Section 5.3 for a more detailed discussion).

Initialization. Both cuHALLaR and HALLaR are initialized with the dual variable $p_0 = 0$ and a primal factor $U_0 \in B_\tau^1$ with a unit single column. The binary versions of HSLR and SDPA set $U_0 = e/\|e\|$ where $e \in \mathbb{R}^n$ is the vector of all-ones, and the on-the-fly versions set U_0 to a randomly generated vector.

Table format. All tables follow a common format, with columns describing the problem instances, reporting the solution times of each method, and presenting ratios between these times. Computational results are reported as $T/\bar{r}$, where T is the solution time in seconds and $\bar{r}$ is the rank of the solution obtained. Reported times exclude data loading and precompilation. If a method fails to reach the prescribed tolerance within the time limit, its result is reported as $*/\bar{\epsilon}$, where $\bar{\epsilon}$ denotes the final error according to the left-hand side of (35). The ratios compare the solution times of the methods indicated in the corresponding column headers.

5.1 Experiments for matrix completion

Given integers $n_2 \geq n_1 \geq 1$, consider the problem of retrieving a low rank matrix $M \in \mathbb{R}^{n_1 \times n_2}$ by observing a subset $\{M_{ij} : (i,j) \in \Omega\}$ of its entries. A standard approach to tackle this problem is by considering the nuclear norm relaxation:

$$\min_{Y \in \mathbb{R}^{n_1 \times n_2}} \quad \{\|Y\|_* \ : \ Y_{ij} = M_{ij}, \ \forall\, (i,j) \in \Omega\}\,,$$

where $\|\cdot\|_*$ as in (4). The above problem can be rephrased as the following SDP:

$$\min_{X \in \mathbb{S}^{n_1+n_2}} \left\{ \frac{1}{2}\operatorname{tr}(X) : X = \begin{pmatrix} W_1 & Y \\ Y^\top & W_2 \end{pmatrix} \succeq 0\,, \quad Y_{i,j} = M_{i,j}\,, \ \forall\, (i,j) \in \Omega \right\}. \tag{36}$$

Matrix completion instances are generated randomly, using the following procedure: given $r \leq n_1 \leq n_2$, the hidden solution matrix M is the product $M_1 M_2^\top$, where the matrices $M_1 \in \mathbb{R}^{n_1 \times r}$ and $M_2 \in \mathbb{R}^{n_2 \times r}$ have independent standard Gaussian random variables as entries. Afterward, m independent and uniformly random entries from M are taken, where $m = \lceil \bar{\gamma}\ r(n_1 + n_2 - r) \rceil$, and $\bar{\gamma} = r \log(n_1 + n_2)$ is the oversampling ratio.

We now discuss how a trace constraint is added to (36) so that it becomes an instance of (P). Let $Y_0 \in \mathbb{R}^{n_1 \times n_2}$ be the trivial completion, which agrees with M_{ij} on the observed entries and is zero elsewhere. Since (36) is a reformulation of the nuclear-norm problem above and its objective is $0.5 \operatorname{tr}(X)$, every optimal solution X_* of (36) satisfies

$$\operatorname{tr}(X_*) = 2 \min_{Y \in \mathbb{R}^{n_1 \times n_2}} \{\|Y\|_* : \ Y_{ij} = M_{ij}, \ \forall\, (i,j) \in \Omega\} \leq 2\|Y_0\|_* \leq 2\sqrt{n_1}\, \|Y_0\|_F, \tag{37}$$

where the first and second inequalities are due to the feasibility of Y_0 for the nuclear-norm problem and the Cauchy–Schwarz inequality applied to the n_1 singular values of Y_0, respectively. Even though adding a trace constraint where $\tau^2 = 2\|Y_0\|_*$ does not discard any optimal solution of (36), it requires the computation of an SVD of Y_0 to evaluate $\|Y_0\|_*$. Instead, the experiments choose τ satisfying $\tau^2 = 2\sqrt{n_1}\|Y_0\|_F$, which is easier to compute.

Table 2 compares HALLaR, cuHALLaR, and cuLoRADS on matrix completion problems. For $n_1 = 3000$, $n_2 = 7000$ and for each $r \in \{3, 5\}$, we generated two instances using different seeds. The second block of columns demonstrates the tremendous speedups our GPU-accelerated solver cuHALLaR achieves over HALLaR using on-the-fly-problem generation. On many instances, cuHALLaR achieves speedups of 10x to 188x compared to HALLaR. For the largest instance considered with $(n_1, n_2) = (800{,}000, 1{,}200{,}000)$, cuHALLaR solves the problem in 53 seconds, while HALLaR requires nearly 2.5 hours.

The last block of columns compares our binary implementation of cuHALLaR, which is run with both HSLR and SDPA input formats, with cuLoRADS, which is run with SDPA format. With HSLR format, cuHALLaR is faster than cuLoRADS on 24 of the 26 instances, with speedup factors ranging from 1.2 to 37.1. On the remaining two instances, both with $(n_1, n_2) = (3{,}000, 7{,}000)$ and $r = 5$, cuLoRADS is faster by a factor of approximately 1.1. For the seven largest instances, listed in the lower portion of the table, cuHALLaR with HSLR input achieves speedups of $2\times$ to $4\times$ over cuLoRADS. The near-optimal solutions obtained by cuHALLaR also have lower rank than those returned by cuLoRADS.

Problem Instance		HALLaR vs cuHALLaR (on-the-fly)			cuLoRADS vs cuHALLaR			
Size $(n_1, n_2; m)$	r	CPU	GPU	Ratio	cuHALLaR (HSLR)	cuHALLaR (SDPA)	cuLoRADS (SDPA)	Ratio*
3,000, 7,000; 828,931	3	7.63/3	0.77/3	9.9	3.33/3	3.22/3	123.71/19	**37.1**
3,000, 7,000; 828,931	3	7.52/3	0.70/3	10.7	3.20/3	3.25/3	11.42/19	**3.6**
3,000, 7,000; 2,302,586	5	34.52/5	1.28/5	27.0	3.37/5	2.49/5	3.19/19	0.9
3,000, 7,000; 2,302,586	5	34.42/5	1.09/5	31.6	2.45/5	2.68/5	2.26/19	0.9
10,000, 21,623; 2,948,996	3	38.88/3	1.22/3	31.9	3.31/3	2.68/3	5.14/21	**1.5**
10,000, 21,623; 2,948,996	3	34.80/3	1.11/3	31.4	4.10/3	2.94/3	5.65/21	**1.4**
10,000, 21,623; 8,191,654	5	141.17/5	2.05/5	68.9	4.02/5	7.38/5	4.71/21	**1.2**
10,000, 21,623; 8,191,654	5	140.55/5	1.95/5	72.1	4.02/5	7.74/5	9.00/21	**2.2**
25,000, 50,000; 3,367,574	2	43.52/2	1.20/2	36.3	4.40/2	3.96/2	10.77/23	**2.4**
25,000, 50,000; 7,577,040	3	79.93/3	1.82/3	43.9	4.19/3	3.68/3	21.31/23	**5.1**
30,000, 70,000; 4,605,171	2	60.57/2	1.63/2	37.2	4.43/2	3.48/2	17.02/24	**3.8**
30,000, 70,000; 10,361,633	3	145.76/3	2.57/3	56.7	5.12/3	4.54/3	18.45/24	**3.6**
50,000, 100,000; 7,151,035	2	121.63/2	1.89/2	64.4	12.36/2	5.18/2	23.67/24	**1.9**
50,000, 100,000; 16,089,828	3	242.00/3	3.68/3	65.8	7.04/3	5.69/3	27.95/24	**4.0**
80,000, 120,000; 9,764,859	2	237.38/2	2.45/2	96.9	5.63/2	6.05/2	20.66/25	**3.7**
80,000, 120,000; 21,970,931	3	490.07/3	5.02/3	97.6	6.92/3	6.27/3	32.28/25	**4.7**
120,000, 180,000; 34,051,152	3	1023.0/4	7.35/4	139.2	9.85/4	17.62/4	41.24/26	**4.2**
160,000, 240,000; 46,437,192	3	1419.26/4	8.79/4	161.5	12.24/3	12.12/3	40.65/26	**3.3**
240,000, 360,000; 71,845,299	3	1949.8/3	12.82/3	152.1	28.63/3	20.76/3	67.11/27	**2.3**
320,000, 480,000; 97,865,043	3	2754.83/3	17.41/3	158.2	36.76/3	31.78/3	136.43/28	**3.7**
400,000, 600,000; 124,339,596	3	3472.13/3	22.31/3	155.6	63.39/3	42.01/3	176.81/28	**2.8**
480,000, 720,000; 151,176,587	3	4403.04/3	28.13/3	156.5	71.02/3	46.21/3	159.7/28	**2.2**
560,000, 840,000; 178,314,984	3	6400.22/3	34.72/3	184.3	95.53/3	71.34/3	192.75/29	**2.0**
640,000, 960,000; 205,711,405	3	7374.16/3	39.25/3	187.9	96.83/3	100.77/3	319.17/29	**3.3**
720,000, 1,080,000; 233,333,416	3	8003.14/3	48.85/3	163.8	119.29/3	109.05/3	258.16/29	**2.2**
800,000, 1,200,000; 261,155,840	3	8771.27/3	53.3/3	164.6	114.11/3	110.11/3	293.65/30	**2.6**

Table 2: Performance comparison between HALLaR, cuHALLaR, and cuLoRADS for matrix completion problems using a tolerance of $\epsilon = 10^{-5}$. The time limit is set at three hours.

5.2 Experiments for maximum stable set

Given a graph $\mathcal{G} = ([n], E)$, the maximum stable set problem consists of finding a subset of vertices of largest cardinality such that no two vertices are connected by an edge. Lovász introduced the ϑ-function, which upper bounds the value of the maximum stable set. The ϑ-function is the value of the following SDP relaxation

$$\max_{X \in \mathbb{S}^n} \quad \{ee^\top \bullet X \ : \ X_{ij} = 0, \ ij \in E, \ \mathrm{tr}(X) \leq 1, \ X \succeq 0\} \tag{38}$$

where $e = (1, 1, \ldots, 1)^\top \in \mathbb{R}^n$ is the all ones vector. Written as a minimization problem, (38) is an instance of (P) with $C = -ee^\top$ and trace bound $\tau = 1$. It was shown in [34] that the ϑ-function agrees exactly with the stability number for perfect graphs.

Problem Instance			Runtime (seconds) / rank		
Problem Size $(n; m)$	Graph	Dataset	HALLaR	cuHALLaR	Ratio
10,937; 75,488	wing_nodal	DIMACS10	3,826.34/130	246.71/127	**15.5**
16,384; 49,122	delaunay_n14	DIMACS10	643.90/36	102.43/34	**6.3**
16,386; 49,152	fe-sphere	DIMACS10	21.74/3	17.82/3	**1.2**
22,499; 43,858	cs4	DIMACS10	749.95/15	588.46/115	**1.3**
25,016; 62,063	hi2010	DIMACS10	2,607.24/15	132.85/21	**19.6**
25,181; 62,875	ri2010	DIMACS10	1,911.10/17	1,951.28/190	0.98
32,580; 77,799	vt2010	DIMACS10	2,952.84/18	196.59/60	**15.0**
48,837; 117,275	nh2010	DIMACS10	7,694.06/20	2,186.74/172	**3.5**
24,300; 34,992	aug3d	GHS_indef	30.10/1	2.08/1	**14.5**
32,430; 54,397	ia-email-EU	Network Repo	598.76/5	52.86/4	**11.3**
11,806; 32,730	Oregon-2	SNAP	1,581.11/23	467.75/55	**3.4**
21,363; 91,286	ca-CondMat	SNAP	8,521.04/59	348.59/77	**24.4**
31,379; 65,910	as-caida_G_001	SNAP	2,125.41/8	428.41/27	**5.0**
26,518; 65,369	p2p-Gnutella24	SNAP	312.75/4	55.99/10	**5.6**
22,687; 54,705	p2p-Gnutella25	SNAP	242.42/4	88.53/13	**2.7**
36,682; 88,328	p2p-Gnutella30	SNAP	506.26/5	61.84/10	**8.2**
62,586; 147,892	p2p-Gnutella31	SNAP	1,484.71/5	245.33/29	**6.0**
49,152; 69,632	cca	AG-Monien	63.09/2	12.30/2	**5.1**
49,152; 73,728	ccc	AG-Monien	14.52/2	20.62/2	0.7
49,152; 98,304	bfly	AG-Monien	15.72/2	10.35/2	**1.5**
16,384; 32,765	debr_G_12	AG-Monien	250.40/10	189.57/10	**1.3**
32,768; 65,533	debr_G_13	AG-Monien	471.80/9	638.54/13	0.7
65,536; 131,069	debr_G_14	AG-Monien	474.22/9	29.51/9	**16.1**
131,072; 262,141	debr_G_15	AG-Monien	521.88/10	28.07/10	**18.6**
262,144; 524,285	debr_G_16	AG-Monien	1,333.31/12	84.28/13	**15.8**
524,288; 1,048,573	debr_G_17	AG-Monien	6,437.05/12	171.81/13	**37.4**
1,048,576; 2,097,149	debr_G_18	AG-Monien	16,176.30/13	119.50/13	**135.4**

Table 3: Performance comparison for the Maximum stable set problem. A relative tolerance of $\epsilon = 10^{-5}$ is set.

The results presented in Table 3 reveal a compelling scaling advantage of cuHALLaR over its CPU counterpart, HALLaR, particularly as the dimensions of the SDP instance increase. This trend is most pronounced within the AG-Monien dataset, where for de Bruijn graphs, the performance ratio grows significantly with the number of vertices. For instance, the speedup escalates from 16.1x for `debr_G_14` to a remarkable 135.4x for `debr_G_18`, an instance with over one million vertices and over two million edges. For these large-scale problems, both solvers converge to solutions of nearly identical rank, indicating that the speedup is a direct consequence of the GPU's superior computational throughput for cuHALLaR's core linear algebra operations.

Performance patterns across other graph families confirm the robustness of this advantage. For the SNAP dataset, cuHALLaR delivers consistent and substantial improvements, with speedups ranging from 3.4x to 24.4x. Similarly, for the DIMACS10 instances, cuHALLaR achieves significant accelerations up to 19.6x compared to HALLaR.

Conversely, the results also show scenarios where GPU acceleration offers limited benefits. For smaller or computationally less demanding graphs, such as `fe-sphere` or `ccc`, the overhead associated with GPU kernel launches and memory management outweighs the parallelism benefits, leading to modest speedups or even slowdowns. Furthermore, on certain instances like `ri2010` and `nh2010`, cuHALLaR converges to solutions of substantially higher rank than HALLaR e.g., 190 vs. 17 for `ri2010`. Even in these cases, the runtimes remain competitive, highlighting cuHALLaR's capacity in efficiently managing high-rank factors.

Table 4 compares cuLoRADS with the on-the-fly versions of HALLaR and cuHALLaR on Hamming graphs $H_{10,2}$ through $H_{14,2}$. Our solvers use the HSLR input format while cuLoRADS uses SDPA. We observe that cuLoRADS is slower and terminates with solutions of ranks between 14 and 20, whereas HALLaR and cuHALLaR obtain solutions with rank 2.

Table 5 extends the comparison between HALLaR and cuHALLaR to larger instances, up to $H_{23,2}$. We omit cuLoRADS from this comparison because the storage costs of the SDPA input format prevent its use on the larger

Problem Instance	**HALLaR (on-the-fly)**	**cuHALLaR (on-the-fly)**	**cuLoRADS**	**HALLaR/cuHALLaR**	**cuLoRADS/cuHALLaR**
Graph(n; $\|E\|$)	Time/Rank	Time/Rank	Time/Rank	Ratio	Ratio
$H_{10,2}$(1,024; 5,120)	0.153/2	1.735/2	7.83/14	0.088	**4.5**
$H_{11,2}$(2,048; 11,264)	0.260/2	1.627/2	3.95/16	0.160	**2.4**
$H_{12,2}$(4,096; 24,576)	0.606/2	2.033/2	4.47/17	0.298	**2.2**
$H_{13,2}$(8,192; 53,248)	1.852/2	3.106/2	6.78/19	0.596	**2.2**
$H_{14,2}$(16,384; 114,688)	3.843/2	3.170/2	79.04/20	**1.212**	**24.9**

Table 4: Performance comparison for the Maximum Stable Set problem on Hamming graphs. A relative tolerance of $\epsilon = 10^{-5}$ is set. The HALLaR and cuHALLaR benchmarks were performed using on-the-fly problem generation.

Problem Instance	**On-the-fly Generation**			**HSLR format**
Graph(n; $\|E\|$)	HALLaR	cuHALLaR	Ratio	cuHALLaR (HSLR)
$H_{15,2}$(32,768; 245,760)	8.240/2	3.223/2	**2.557**	2.701/2
$H_{16,2}$(65,536; 524,288)	17.235/2	4.457/2	**3.869**	3.83/2
$H_{17,2}$(131,072; 1,114,112)	33.342/2	3.904/2	**8.541**	2.51/2
$H_{18,2}$(262,144; 2,359,296)	46.436/2	2.627/2	**17.681**	3.43/2
$H_{19,2}$(524,288; 4,980,736)	105.408/2	5.281/2	**19.958**	10.70/2
$H_{20,2}$(1,048,576; 10,485,760)	249.701/2	8.109/2	**30.786**	6.62/2
$H_{21,2}$(2,097,152; 22,020,096)	592.696/2	16.966/2	**34.934**	12.96/2
$H_{22,2}$(4,194,304; 46,137,344)	1,702.255/2	37.145/2	**45.826**	18.842/2
$H_{23,2}$(8,388,608; 96,468,992)	4,205.741/2	83.905/2	**50.125**	51.75/2

Table 5: Performance comparison for the Maximum Stable Set problem on Hamming graphs. A relative tolerance of $\epsilon = 10^{-5}$ is set.

instances, rather than because of an inherent limitation of the solver itself. As discussed in Subsection 3.2, the SDPA format, unlike HSLR, does not exploit the rank-one structure of the dense objective matrix $ee^\top$.

The results for the GSET instances in Table 6, obtained via on-the-fly problem generation, show that performance is strongly correlated with graph structure. For sparse graphs that admit low-rank solutions (e.g., G11, G12, and G32), the CPU-based HALLaR is 30–100x faster than cuHALLaR. Conversely, for dense graphs yielding high-rank solutions, such as G58, G59, and G64, cuHALLaR achieves a 4–5x speedup.

Problem Instance	On-the-fly Generation			HSLR	SDPA	
Graph(n; $\|E\|$)	HALLaR	cuHALLaR	Ratio	cuHALLaR (HSLR)	cuLoRADS (SDPA)	Ratio
G1(800; 19,176)	98.25/98	45.19/98	2.17	53.63/97	340.66	**6.35**
G10(800; 19,176)	80.51/97	40.52/97	1.99	53.11/95	330.18	**6.22**
G11(800; 1,600)	0.14/2	4.23/2	0.03	7.29/2	7.78	**1.07**
G12(800; 1,600)	0.09/2	2.59/2	0.03	6.60/2	8.32	**1.26**
G14(800; 4,694)	45.37/73	214.73/74	0.21	211.04/55	484.41	**2.30**
G20(800; 4,672)	129.73/124	359.40/166	0.36	*/3.0e-05	518.64	0.2 <
G43(1000; 9,990)	34.20/60	35.84/60	0.95	43.27/58	383.83	**8.87**
G51(1,000; 5,909)	167.03/141	273.84/128	0.61	289.18/127	540.96	**1.87**
G23(2,000; 19,990)	94.81/76	52.59/76	1.80	74.39/76	766.91	**10.31**
G31(2,000; 19,990)	122.07/77	58.23/76	2.10	73.34/76	841.21	**11.47**
G32(2,000; 4,000)	0.35/2	25.34/3	0.01	40.89/2	13.45	0.33
G34(2,000; 4,000)	2.00/2	25.29/3	0.08	47.80/2	16.67	0.35
G35(2,000; 11,778)	397.32/133	254.68/108	1.56	689.21/142	14.55	0.02
G41(2,000; 11,785)	343.84/143	286.82/139	1.20	432.96/189	1527.71	**3.53**
G48(3,000; 6,000)	2.29/2	18.95/2	0.12	40.62/2	42.57	**1.05**
G55(5,000; 12,498)	152.03/45	81.11/45	1.87	102.45/39	309.48	**3.02**
G56(5,000; 12,498)	151.99/45	80.83/45	1.88	102.66/39	299.69	**2.92**
G57(5,000; 10,000)	8.71/2	42.55/2	0.20	50.91/2	26.09	0.51
G58(5,000; 29,570)	2,191.49/112	504.55/130	4.35	388.23/67	3178.00	**8.19**
G59(5,000; 29,570)	2,167.27/111	539.28/121	4.02	386.05/65	3159.04	**8.18**
G60(7,000; 17,148)	262.35/50	96.89/50	2.71	145.95/52	455.19	**3.12**
G62(7,000; 14,000)	7.80/3	25.18/3	0.31	21.61/2	34.88	**1.61**
G64(7,000; 41,459)	1,719.79/51	334.12/51	5.15	489.03/72	1655.44	**3.38**
G66(9,000; 18,000)	11.31/3	27.91/3	0.41	73.86/2	37.28	0.50
G67(10,000; 20,000)	2.39/2	5.83/2	0.41	57.05/2	69.30	**1.21**
G72(10,000; 20,000)	2.41/2	5.81/2	0.41	65.19/2	33.56	0.51
G77(14,000; 28,000)	27.34/3	11.81/2	2.31	57.71/2	52.19	0.90
G81(20,000; 40,000)	53.85/3	64.20/3	0.84	66.28/2	71.37	**1.08**

Table 6: Performance comparison for the Maximum Stable Set problem on GSET instances. Tolerances are set to 10^{-5}. Time limit was set to 1 hour. Ratio in the last column is computed as cuLoRADS (SDPA) / cuHALLaR (HSLR).

The last block of columns of Table 6 demonstrates a clear performance advantage of cuHALLaR (with HSLR format) over cuLoRADS, which utilizes standard SDPA format. Our method cuHALLaR achieves significant speedups over cuLoRADS on medium-to-large sized graphs with substantial edge density. For instance, on graphs G23 and G31, cuHALLaR achieves speedups of 10.3x and 11.5x, respectively. This trend continues for larger graphs like G58 and G59, where the speedup is consistently over 8x. However, the performance dynamic reverses for SDP instances where the underlying graph is very sparse. On graphs such as G32 and G34, where the number of edges is only twice the number of vertices, cuLoRADS is approximately three times faster than cuHALLaR. Similarly, for G57, G66, and G72, cuLoRADS is roughly two times faster than cuHALLaR. This performance dichotomy suggests that the architectural benefits of the HSLR format and the associated kernels in cuHALLaR are most pronounced when the computational workload per iteration is high. For highly sparse problems, the lower overhead of processing the simpler SDPA format in cuLoRADS proves more efficient, indicating that the relative performance of the solvers is strongly dependent on the underlying graph structure.

5.3 Experiments for phase retrieval

For coded diffraction measurements, the masks $d_1, \ldots, d_L \in \mathbb{C}^n$ are given data which define measurement matrices according to

$$B_\ell = n^{-1/2} F \mathrm{Diag}(\overline{d_\ell}), \quad \ell = 1, \ldots, L, \tag{39}$$

where $F \in \mathbb{C}^{n \times n}$ is the unnormalized discrete Fourier transform matrix defined as

$$F_{st} = e^{-2\pi \mathrm{i}(s-1)(t-1)/n}, \quad s, t = 1, \ldots, n. \tag{40}$$

The phase retrieval problem seeks to recover a signal $x \in \mathbb{C}^n$, up to a global phase, from measurements

$$|B_\ell x|^2 = b_\ell, \qquad \ell = 1, \dots, L, \tag{41}$$

where squared magnitudes are taken entrywise and $b_\ell \in \mathbb{R}^n_+$. Set $m = nL$, $b = [b_1; \dots; b_L] \in \mathbb{R}^m_+$, and define the linear operator $\mathcal{A} : \mathbb{S}^n(\mathbb{C}) \to \mathbb{R}^m$ by

$$\mathcal{A}(X) = \begin{bmatrix} \operatorname{diag}(B_1 X B_1^H) \\ \vdots \\ \operatorname{diag}(B_L X B_L^H) \end{bmatrix}. \tag{42}$$

With $X = xx^H$, (41) can be written as $\mathcal{A}(X) = b$, where $X \succeq 0$ and $\operatorname{rank}(X) \leq 1$. Dropping the rank constraint and minimizing the trace gives the PhaseLift SDP relaxation [14]

$$\min_{X \in \mathbb{S}^n(\mathbb{C})} \{\operatorname{tr}(X) : \mathcal{A}(X) = b, \quad X \succeq 0\}. \tag{43}$$

The adjoint $\mathcal{A}^* : \mathbb{R}^m \to \mathbb{S}^n(\mathbb{C})$ is the linear operator

$$\mathcal{A}^*(\hat{p}) = \sum_{\ell=1}^{L} B_\ell^H \operatorname{Diag}(\hat{p}_\ell) B_\ell, \tag{44}$$

where $\hat{p} = [\hat{p}_1; \dots; \hat{p}_L]$ and $\hat{p}_\ell \in \mathbb{R}^n$.

On-the-fly operators. For $U = [u_1 \cdots u_r] \in \mathbb{C}^{n \times r}$, HALLaR and cuHALLaR loop over the columns of U to compute the two quantities $\mathcal{A}(UU^H)$ and $\mathcal{A}^*(\hat{p})U$ as

$$\mathcal{A}(UU^H) = \sum_{j=1}^{r} \mathcal{A}(u_j u_j^H), \quad \mathcal{A}^*(\hat{p})U = \big[\mathcal{A}^*(\hat{p})u_1 \ \cdots \ \mathcal{A}^*(\hat{p})u_r\big].$$

Hence, to compute the above quantities, it suffices to describe how to compute $\mathcal{A}(uu^H)$ and $\mathcal{A}^*(\hat{p})u$ efficiently for a single vector $u \in \mathbb{C}^n$, which is the main focus of the next three paragraphs.

For $u \in \mathbb{C}^n$, both HALLaR and cuHALLaR evaluate $\mathcal{A}(uu^H)$ and $\mathcal{A}^*(\hat{p})u$ using FFTs and entrywise products, without forming any $n \times n$ matrices. In particular it follows from (42) that,

$$\mathcal{A}(uu^H) = \begin{bmatrix} |B_1 u|^2 \\ \vdots \\ |B_L u|^2 \end{bmatrix} = \frac{1}{n} \begin{bmatrix} |F(\overline{d_1} \odot u)|^2 \\ \vdots \\ |F(\overline{d_L} \odot u)|^2 \end{bmatrix}, \tag{45}$$

where $\odot$ denotes entrywise multiplication. Since $F^H F = nI$, the inverse DFT satisfies $F^{-1} = n^{-1} F^H$. Thus, (44) implies

$$\mathcal{A}^*(\hat{p})u = \sum_{\ell=1}^{L} d_\ell \odot F^{-1}\Big(\hat{p}_\ell \odot F(\overline{d_\ell} \odot u)\Big). \tag{46}$$

The products Fv and $F^{-1}v$, where $v \in \mathbb{C}^n$, can be computed using the Julia routines `fft` and `ifft`, respectively. Evaluating $\mathcal{A}(uu^H)$ requires L forward FFTs, while evaluating $\mathcal{A}^*(\hat{p})u$ requires L forward and L inverse FFTs. Each evaluation therefore costs $O(Ln \log n)$ arithmetic operations. Since $m = nL$, the on-the-fly implementation requires $O(Ln) = \mathcal{O}(m)$ storage for the L masks $d_\ell \in \mathbb{C}^n$ and the measurement vector $b \in \mathbb{R}^m$. An HSLR representation would be substantially more costly for large n, since storing b and the L measurement matrices $B_\ell \in \mathbb{C}^{n \times n}$ explicitly requires $O(Ln^2 + Ln)$ storage.

cuHALLaR uses the cuFFT library [57] to compute (45) and (46). Both operators are evaluated through a sequence of GPU operations. First, the vectors $v_\ell = \overline{d_\ell} \odot u$, $\ell = 1, \dots, L$, are computed in parallel. Second, their Fourier transforms $Fv_1, \dots, Fv_L$ are evaluated in a single batched operation using cuFFT, which internally parallelizes the computation on the GPU. Then, for $\mathcal{A}(uu^H)$, the squared moduli $n^{-1}|Fv_\ell|^2$ are computed and accumulated entrywise in parallel. For $\mathcal{A}^*(\hat{p})u$, the forward FFTs are followed by entrywise weighting $\hat{p}_\ell \odot Fv_\ell$, computation of the inverse FFTs, multiplication by the masks d_l, and a GPU reduction over the masks.

Handling complex data Although cuHALLaR is presented for real symmetric SDPs, our implementation also supports complex Hermitian SDPs using native complex arithmetic in Julia. We use the factorization $X = YY^H$ with $Y \in \mathbb{C}^{n\times r}$, where Y^H is the conjugate transpose of Y. For complex matrices $U, V \in \mathbb{C}^{n\times r}$, we use the real inner product

$$\langle U, V\rangle_{\mathbb{R}} = \operatorname{Re}\operatorname{tr}(U^H V),$$

where Re denotes the real part of a complex number. With respect to this inner product, the gradient of $\tilde{g}(Y) = \mathcal{L}_\beta(YY^H; p)$, where $\mathcal{L}_\beta(\cdot; p) : \mathbb{S}^n(\mathbb{C}) \to \mathbb{R}$, is

$$\nabla\tilde{g}(Y) = 2\nabla_X \mathcal{L}_\beta(YY^H; p)Y. \tag{47}$$

Thus, the gradient has the same form as in the real case, with the transpose replaced by the conjugate transpose. The evaluations of on-the-fly operators mentioned in the previous paragraph use Julia's FFT routines, which also support complex data. Our implementations of cuHALLaR and HALLaR support complex matrices and Hermitian eigenvalue computations, so the phase retrieval experiments do not require modifications to these routines or conversion to a real symmetric SDP.

Experimental setup Synthetic instances are generated using the coded diffraction model in [89]. The entries of the $L = 12$ masks are sampled independently. Each entry is the product of two independent random variables: a phase drawn uniformly from $\{1, -1, \mathrm{i}, -\mathrm{i}\}$ and an amplitude equal to $\sqrt{3}$ with probability $1/5$ or $1/\sqrt{2}$ with probability $4/5$. A signal $\hat{x} \in \mathbb{C}^n$ is drawn from the standard complex Gaussian distribution, resampling until $\|\hat{x}\|^2 \le 3n$. The noiseless measurements are then generated as

$$b_\ell = |B_\ell \hat{x}|^2, \qquad \ell = 1, \dots, L.$$

Since $\hat{x}\hat{x}^H$ is feasible for (43), every optimal solution X_* satisfies

$$\operatorname{tr}(X_*) \le \operatorname{tr}(\hat{x}\hat{x}^H) = \|\hat{x}\|^2 \le 3n.$$

Thus, adding the trace constraint $\operatorname{tr}(X) \le \tau^2$ with $\tau^2 = 3n$ preserves all optimal solutions of (43).

In these experiments, HALLaR and cuHALLaR use their Julia implementations with the on-the-fly operators described above. Only the masks d_ℓ and vectors b_ℓ, $\ell = 1, \dots L$, are generated once and stored into memory. Neither F, the matrices B_ℓ, nor the individual SDP constraint matrices are formed. The precompiled binaries are not used for these experiments. These binaries read the SDP data from HSLR or SDPA files. SDPA stores matrix entries, while HSLR can also store low-rank factors. For phase retrieval, storing all constraint matrices B_ℓ or their factors in HSLR or SDPA format requires substantially more memory than storing the masks. The on-the-fly implementation avoids this storage by using Julia's FFT routines. The available cuHALLaR binary also accepts only real-valued inputs and therefore cannot directly handle these complex instances.

Problem Instance (n; m)	HALLaR (on-the-fly)	cuHALLaR (on-the-fly)	Ratio
10,000; 120,000	53.642/5	40.866/5	**1.31**
10,000; 120,000	18.384/2	5.529/4	**3.32**
10,000; 120,000	23.644/4	4.347/3	**5.44**
10,000; 120,000	14.957/3	3.462/3	**4.32**
31,623; 379,476	149.416/5	29.016/5	**5.15**
31,623; 379,476	119.499/4	30.964/4	**3.86**
31,623; 379,476	153.16/5	29.279/5	**5.23**
31,623; 379,476	127.10/4	30.964/5	**4.11**
100,000; 1,200,000	303.986/5	10.401/4	**29.24**
100,000; 1,200,000	330.00/6	8.641/4	**38.18**
100,000; 1,200,000	434.10/5	9.279/4	**46.78**
100,000; 1,200,000	380.05/6	9.334/4	**40.75**
316,228; 3,794,736	600.62/2	29.016/6	**20.70**
316,228; 3,794,736	699.193/2	30.964/6	**22.58**
316,228; 3,794,736	507.60/2	33.738/7	**15.05**
316,228; 3,794,736	620.56/2	34.124/8	**18.19**
3,162,278; 37,947,336	*/4.1e-03	583.782/14	> 74.0

Table 7: Performance comparison for the phase retrieval problem. A relative tolerance of $\epsilon = 10^{-5}$ is used, and the time limit for HALLaR is 12 hours.

Table 7 shows that cuHALLaR achieves substantial acceleration over HALLaR for SDP instances of the phase retrieval problem. Interestingly, the speedup that cuHALLaR achieves over HALLaR is generally more significant as the size of the SDP instance increases. For instances with dimension $n = 31{,}623$, cuHALLaR achieves speedups of 3.86x to 5.23x over HALLaR. For larger instances with dimension $n = 100,000$, the speedup that cuHALLaR achieves over HALLaR increases dramatically to 29.24x–46.78x. For phase retrieval instances considered with dimension $n = 316,228$, cuHALLaR achieves speedups from 15.05x to 22.58x over HALLaR.

Most significantly, cuHALLaR can successfully solve a massive phase retrieval instance with dimension $n = 3,162,278$ in under 10 minutes. In stark contrast, HALLaR fails to solve this instance within the 12-hour time limit. Hence, for this huge problem instance, cuHALLaR is at least 74 times faster than HALLaR.

5.4 General comments and conclusions about numerical results

Initialization and rank growth. In HALLaR and cuHALLaR, the user-specified integer $s_0 \geq 1$ determines the number of columns in the initial point. In all experiments, we set $s_0 = 1$ for both methods and initialize with a randomly generated single column. Starting from either the origin or a rank-one initialization is the most natural course of action if no information is known about the ranks of the optimal solutions. Moreover, such an initialization keeps the storage and computational cost low for problems with low-rank solutions and also keeps these costs low during the initial iterations for problems with high-rank solutions. Users may choose $s_0 > 1$, in which case step 6 of the very first call of the HLR subroutine returns a factor Y with s_0 columns. Since the ranks of the iterates typically remain constant or increase from that point onward, an overly large value of s_0 may lead to iterates of unnecessarily high rank, increasing computational costs.

Several of the problem classes considered here admit low-rank optimal solutions under suitable assumptions. It is well known (see e.g. [12, 66]) that a nuclear-norm matrix minimization problem guarantees exact recovery of a rank-r ground-truth matrix under suitable sampling and incoherence assumptions and that a corresponding SDP reformulation admits an optimal solution of rank r. For the maximum stable set problem, the Lovász theta SDP relaxation admits a rank-one optimal solution whenever the relaxation is tight. The relaxation is tight for perfect graphs, which include the binary Hamming graphs $H_{d,2}$ considered in Tables 4 and 5. For graphs that are not perfect, the associated SDP relaxations can have high-rank solutions (see e.g., Tables 3 and 6). Moreover, it is well-known (see e.g. [13, 14]) that, under suitable measurement assumptions, the ground-truth solution of the original phase-retrieval problem is a rank-one optimal solution of the PhaseLift SDP relaxation (43).

Although theoretical results [62, 70, 4] for general SDPs establish the existence of optimal SDP solutions satisfying certain rank bounds, these bounds are often conservative. Neither these results nor our convergence analysis guarantee that the iterates generated or the solutions returned by HALLaR and cuHALLaR have low rank. Empirically, the final ranks obtained by HALLaR and cuHALLaR for the matrix completion, phase retrieval, and Hamming maximum

stable set instances are small.

Summary of the numerical results. To assess the benefit of GPU acceleration, we compare cuHALLaR with its CPU counterpart, HALLaR, using on-the-fly operators in both implementations. For matrix completion with $n_1 \geq 80{,}000$, cuHALLaR achieves speedup factors of 96.9–187.9 (Table 2). For maximum stable set problems, the speedup factors are 15.8–135.4 for `debr_G_14`–`debr_G_18` and 1.2–50.1 for $H_{14,2}$–$H_{23,2}$. For phase retrieval with $n \geq 100{,}000$, cuHALLaR achieves speedup factors of 15.05–46.78 on the instances that both methods solve within the time limit. Moreover, cuHALLaR solves the largest phase retrieval instance, whereas HALLaR fails to do so within the 12-hour time limit.

The CPU-based HALLaR is faster on smaller instances, including $H_{10,2}$–$H_{13,2}$ and the sparse GSET graphs G11, G12, and G32. On these instances, the work per iteration may be too small to offset the overhead of launching GPU kernels. Differences in factor rank between the two implementations can also affect the observed speedups, since higher ranks increase the computational cost of each iteration.

Achieved accuracy. Table 10 in Appendix B.3 reports the final primal residual, relative primal–dual gap and dual infeasibility of HALLaR and cuHALLaR on selected instances of the three problem classes.

6 Concluding Remarks

This paper presents cuHALLaR, a GPU implementation of the hybrid low-rank augmented Lagrangian method HALLaR of [55] for SDPs with a trace bound $\mathrm{tr}(X) \leq \tau^2$. Our approach exploits the parallel computing capabilities of modern GPUs to overcome computational bottlenecks inherent in handling massive SDP instances, while maintaining the advantages of low-rank factorization-based methods.

Across the three structured problem classes considered in our experiments, cuHALLaR achieves substantial speedups over the CPU-based HALLaR on large instances and is also faster than cuLoRADS on most instances tested with both solvers. For matrix completion, cuHALLaR with HSLR input achieves speedups of 1.2× to 37.1× over cuLoRADS on 24 of the 26 instances in Table 2. The final ranks range from 2 to 5 for cuHALLaR and from 19 to 30 for cuLoRADS. On the largest matrix completion instance, with approximately 2.6×10^8 constraints, cuHALLaR using on-the-fly input terminates in 53.3 seconds, compared with 8,771 seconds (about 2.4 hours) for HALLaR.

For maximum stable set problems, the advantage of cuHALLaR over HALLaR becomes more pronounced as problem size and graph density increase. Speedups reach 50× for the largest Hamming graph problems and 135× for million-node AG-Monien instances. On smaller instances, such as $H_{10,2}$ through $H_{13,2}$ and the sparse GSET graphs G11, G12, and G32, however, the CPU-based HALLaR is faster. For phase retrieval problems, cuHALLaR achieves speedups of 15× to 47× on large instances and solves the largest test case, which has a matrix variable of order exceeding 3 million, in under 10 minutes. In contrast, the CPU-based HALLaR fails to solve the same instance within the 12-hour time limit.

Future work. While cuHALLaR demonstrates exceptional performance across the problems tested, certain structural characteristics of SDPs may impact its efficiency. Problems with inherently high-rank solutions or those requiring extremely high precision may benefit less from our approach. Additionally, our current implementation focuses on single-GPU execution and double-precision arithmetic. Future research directions include developing optimized multi-GPU implementations in C/C++ to further enhance scalability, exploring mixed-precision computation techniques to balance accuracy and performance, and extending our methodology to a broader class of SDP problems such as those with multi-block matrix variables and inequality constraints.

Extending cuHALLaR to handle both affine equality and inequality constraints using a modified augmented Lagrangian function is an important future direction (see, for example, [44, 87, 68]). This extension would require a new analysis to bound the sequence of Lagrange multipliers and a modified rule for choosing the Frank–Wolfe stepsize. Another important direction is extending cuHALLaR to large-scale doubly nonnegative SDP relaxations, which impose entrywise nonnegativity constraints, i.e., $X \geq 0$. Here, the principal challenge is scalability. A direct augmented Lagrangian treatment introduces $\Theta(n^2)$ multipliers associated with the nonnegativity constraints. The storage and update costs of these multipliers can become prohibitive even when the primal matrix admits a low-rank representation. Preserving the memory and computational advantages of cuHALLaR therefore requires a careful treatment of these constraints.

Finally, the current implementation of cuHALLaR requires a valid trace bound τ, either included in the SDP formulation or supplied by the user. In the latter case, an insufficient bound may exclude all optimal solutions of the original SDP, whereas an unnecessarily large bound can make optimality certification more demanding. An important direction for future work is to eliminate the need to supply a valid bound in advance. One possibility is

to increase τ adaptively by introducing an artificial trace constraint $\mathrm{tr}(X) \leq \tau^2$, solving the resulting trace-bounded SDP, and checking if the trace constraint is nearly active. If it is, then τ is increased and the process is repeated with the updated τ. Another possibility is to remove the trace constraint $\mathrm{tr}(X) \leq \tau^2$ and solve the AL subproblems directly over the unbounded cone $\mathbb{S}^n_+$, drawing on the ideas of Laue [45] and Duchi et al. [26]. These developments would extend the approach of [55] to SDPs for which a suitable trace bound is not known in advance or overly conservative.

Acknowledgments

This research was supported in part through research cyberinfrastructure resources and services provided by the Partnership for an Advanced Computing Environment (PACE) at Georgia Tech, Atlanta, Georgia, USA, along with allocation MTH250047 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by U.S. National Science Foundation grants.

Appendix

A ADAP-AIPP

This section presents the ADAP-AIPP method first developed in [55, 72]. HALLaR uses ADAP-AIPP in its step 5 to find an approximate stationary point (according to the criterion in (9)) of a nonconvex problem of the form

$$\min_{U} \quad \left\{ g(U) = \mathcal{L}_\beta(UU^\top; p) \quad : \quad U \in B^s_\tau \right\} \tag{48}$$

where $\mathcal{L}_\beta(X; p)$ is as in (7) and B^s_τ is as in (3).

Given an initial point $\underline{W} \in B^s_\tau$ and a tolerance $\rho > 0$, the goal of ADAP-AIPP is to find a triple $(\overline{W}; \overline{R}, \rho)$ that satisfies

$$\overline{R} \in \nabla g(\overline{W}) + N_{B^s_\tau}(\overline{W}), \quad \|\overline{R}\|_F \leq \rho, \quad g(\overline{W}) \leq g(\underline{W}). \tag{49}$$

The ADAP-AIPP method is now formally presented.

ADAP-AIPP Method

Universal Parameters: $\sigma \in (0, 1/2)$ and $\chi \in (0, 1)$.
Input: a function g as in (48), an initial point $\underline{W} \in B^s_\tau$, an initial prox stepsize $\lambda_0 > 0$, and a tolerance $\rho > 0$.

0. set $W_0 = \underline{W}$, $j = 1$, and

$$\lambda = \lambda_0, \quad \bar{M}_0 = 1; \tag{50}$$

1. choose $\underline{\mathrm{M}}_j \in [1, \bar{M}_{j-1}]$ and call the ADAP-FISTA method in Subsection A.1 with universal input (σ, χ) and inputs

$$x_0 = W_{j-1}, \quad (\mu, L_0) = (1/2, \underline{\mathrm{M}}_j), \tag{51}$$

$$\psi_s = \lambda g + \frac{1}{2}\|\cdot - W_{j-1}\|^2_F, \quad \psi_n = \lambda \delta_{B^s_\tau}; \tag{52}$$

2. if ADAP-FISTA fails or its output (W, V, L) (if it succeeds) does not satisfy the inequality

$$\lambda g(W_{j-1}) - \left[\lambda g(W) + \frac{1}{2}\|W - W_{j-1}\|^2_F\right] \geq V \bullet (W_{j-1} - W), \tag{53}$$

then set $\lambda = \lambda/2$ and go to step 1; else, set $(\lambda_j, \bar{M}_j) = (\lambda, L)$, $(W_j, V_j) = (W, V)$, and

$$R_j := \frac{V_j + W_{j-1} - W_j}{\lambda_j} \tag{54}$$

and go to step 3;

3. if $\|R_j\|_F \leq \rho$, then stop with success and output $(\overline{W}, \overline{R}) = (W_j, R_j)$; else, go to step 4;

4. set $j \leftarrow j+1$ and go to step 1.

Several remarks about ADAP-AIPP are now given.

1. It is shown in [55] that ADAP-AIPP is able to find a triple that satisfies (49) in $\mathcal{O}(1/\rho^2)$ number of iterations.
2. ADAP-AIPP is a double-looped algorithm, whose outer iterations are indexed by j. During its j-th iteration, ADAP-AIPP attempts to solve a proximal subproblem of the form $\min_{U \in B_\tau^s} \lambda g(U) + 0.5\|U - W_{j-1}\|_F^2$, where λ is a prox stepsize and W_{j-1} is the current prox center. If a proximal subproblem cannot suitably be solved, ADAP-AIPP reduces λ and tries to solve the new subproblem.
3. ADAP-AIPP uses the ADAP-FISTA method, which is presented in the next Subsection A.1, to attempt to solve its proximal subproblems. For a more comprehensive treatment of ADAP-FISTA, see Appendix B in [55].

A.1 ADAP-FISTA

This subsection presents the ADAP-FISTA method that ADAP-AIPP invokes to solve proximal subproblems of the form

$$\min_{U \in B_\tau^s} \psi_s(U) := \lambda g(U) + 0.5\|U - W\|_F^2 \tag{55}$$

where g is as in (48). The ADAP-FISTA method is now formally presented.

ADAP-FISTA

Universal Parameters: $\sigma > 0$ and $\chi \in (0,1)$.
Input: a function ψ_s as in (55), an initial point $x_0 \in B_\nu^s$ and scalars $\mu > 0$ and $L_0 > \mu$.

0. set $y_0 = x_0$, $A_0 = 0$, $\nu_0 = 1$, and $i = 0$;

1. set $L_{i+1} = L_i$;

2. compute

$$a_i = \frac{\nu_i + \sqrt{\nu_i^2 + 4\,\nu_i A_i\,(L_{i+1} - \mu)}}{2\,(L_{i+1} - \mu)}, \quad \tilde{x}_i = \frac{A_i\,y_i + a_i\,x_i}{A_i + a_i}, \tag{56}$$

$$y_{i+1} = \arg\min_{u \in B_\nu^s} \left\{ \ell_{\psi_s}(u; \tilde{x}_i) + \tfrac{L_{i+1}}{2}\|u - \tilde{x}_i\|^2 \right\}; \tag{57}$$

if

$$\ell_{\psi_s}(y_{i+1}; \tilde{x}_i) + \tfrac{(1-\chi)\,L_{i+1}}{4}\,\|y_{i+1} - \tilde{x}_i\|^2 \;\geq\; \psi_s(y_{i+1}), \tag{58}$$

go to step 3; else set $L_{i+1} \leftarrow 2\,L_{i+1}$ and repeat step 2;

3. update

$$A_{i+1} = A_i + a_i, \qquad \nu_{i+1} = \nu_i + a_i\,\mu, \tag{59}$$

$$u_{i+1} = (L_{i+1} - \mu)\,(\tilde{x}_i - y_{i+1}), \tag{60}$$

$$x_{i+1} = \frac{1}{\nu_{i+1}}\left[\mu\,a_i\,y_{i+1} + \nu_i\,x_i - a_i\,s_{i+1}\right]; \tag{61}$$

4. if

$$\|y_{i+1} - x_0\|^2 \;\geq\; \chi\,A_{i+1}\,L_{i+1}\,\|y_{i+1} - \tilde{x}_i\|^2, \tag{62}$$

then go to step 5; otherwise stop with **failure**;

5. If the inequality

$$\|u_{i+1}\| \;\leq\; \sigma\,\|y_{i+1} - x_0\| \tag{63}$$

holds, then stop with **success** and output $(y, u, L) = (y_{i+1}, u_{i+1}, L_{i+1})$; otherwise set $i \leftarrow i+1$ and return to step 1.

Several remarks about ADAP-FISTA are now given. ADAP-FISTA is either able to successfully find a suitable solution of (55) or is unable to do so in at most

$$\mathcal{O}\left(\sqrt{L_{\psi_s}}\right)$$

number of iterations, where L_{ψ_s} is the Lipschitz constant of the gradient of the objective in (55). If the objective in (55) is strongly convex, then ADAP-FISTA always succeeds in solving (55). For more technical details on the type of solution that ADAP-FISTA aims to find, see Appendix B in [55].

B Additional Experiments

B.1 Minimum Eigenpair Computations

This subsection measures the contribution of MEV computations to the total runtime of cuHALLaR. We profile four configurations, each repeated three times, on one NVIDIA H200 with eight CPU threads, Julia 1.10.1, CUDA.jl 5.11.2, and KrylovKit 0.9.4. These separate profiling runs use solver and eigenpair tolerances of 10^{-5} and 10^{-10}. Table 8 reports the four configurations for which all three returned solutions satisfy the 10^{-5} accuracy target. The remaining configurations required postprocessing or encountered accuracy or memory failures.

Table 8: Time spent in minimum eigenpair computations. Times are three-run medians in seconds; the MEV share is the median of the per-run percentages.

Problem	Input	Total solve	MEV	MEV (%)
Matrix completion (80,000 × 120,000)	On-the-fly	10.62	8.50	79.9
Matrix completion (80,000 × 120,000)	HSLR	11.53	8.70	75.5
Stable set (`debr_G_14`)	On-the-fly	129.41	82.07	64.6
Phase retrieval ($n = 3{,}162{,}278$)	On-the-fly	54.72	29.99	54.7

MEV time includes all eigensolver calls, with GPU synchronization at each timing boundary. Both times exclude input preparation, compilation warmup, and final verification.

Minimum eigenpair computations account for 54.7–79.9% of the total solve time in these configurations.

B.2 Memory Usage

This subsection reports the memory usage of cuHALLaR on six matrix-completion instances using the on-the-fly representation. These separate runs use one NVIDIA H200, Julia 1.10.1, CUDA.jl 5.11.2, and a solver tolerance of 10^{-5}. All six runs meet the reported accuracy targets. Table 9 lists the largest sampled GPU-memory usage, peak host resident memory (RAM), and maximum factor width reported at the outer iterations.

Table 9: Memory usage for matrix completion with the on-the-fly representation. Each row corresponds to one run; memory is measured in GiB (2^{30} bytes).

Matrix dimensions	Sampled GPU maximum	Host peak	Max. reported factor width
240,000 × 360,000	35.92	16.43	3
320,000 × 480,000	43.58	23.82	3
400,000 × 600,000	51.29	28.18	3
480,000 × 720,000	55.29	32.57	3
640,000 × 960,000	72.64	47.78	3
800,000 × 1,200,000	72.79	56.88	3

GPU usage is sampled approximately every two seconds and includes CUDA context and cached allocations. Both memory measurements cover the entire process, including input preparation and warmup. Sampling may miss transient memory peaks.

Across these instances, sampled GPU-memory usage increases from 35.92 to 72.79 GiB, while the maximum reported factor width remains three. Interestingly, no instance ever reaches the maximum of 144 GiB that is available on the NVIDIA H200 GPU.

B.3 Achieved Accuracy

Table 10 reports the final primal residual, relative primal–dual gap and dual infeasibility, i.e., the three terms in (35), attained by HALLaR and cuHALLaR in the runs of Tables 2, 3, 5 and 7 for selected instances of the three problem classes.

Instance	Method (input)	Time (s)/rank	Primal residual	Relative gap	Dual infeasibility
Matrix completion, Table 2					
800,000 × 1,200,000; $m = 261{,}155{,}840$; $r = 3$	HALLaR (on-the-fly)	8771.27/3	1.3e-08	8.3e-06	6.2e-11
	cuHALLaR (on-the-fly)	53.30/3	1.3e-08	8.3e-06	6.2e-11
	cuHALLaR (HSLR)	114.11/3	9.8e-10	5.0e-06	0
Maximum stable set, Tables 3 and 5					
$H_{23,2}$ (8,388,608; 96,468,992)	HALLaR (on-the-fly)	4,205.741/2	4.0e-09	1.7e-06	0
	cuHALLaR (on-the-fly)	83.905/2	4.0e-09	1.7e-06	0
	cuHALLaR (HSLR)	51.75/2	3.4e-08	7.3e-06	0
`debr_G_18` (1,048,576; 2,097,149)	HALLaR (on-the-fly)	16,176.30/13	2.1e-06	8.9e-06	0
	cuHALLaR (on-the-fly)	119.50/13	1.1e-06	2.1e-06	0
Phase retrieval, Table 7					
$n = 100{,}000$; $m = 1{,}200{,}000$ (first instance)	HALLaR (on-the-fly)	303.986/5	3.0e-08	5.4e-06	0
	cuHALLaR (on-the-fly)	10.401/4	2.1e-08	7.9e-06	3.9e-10
$n = 316{,}228$; $m = 3{,}704{,}736$ (first instance)	HALLaR (on-the-fly)	600.62/2	4.9e-07	1.3e-06	0
	cuHALLaR (on-the-fly)	29.016/6	2.3e-08	9.9e-06	3.0e-10

Table 10: Final relative errors of HALLaR and cuHALLaR for selected instances; A tolerance of $\epsilon = 10^{-5}$ is set. Graph instances are specified by $(n; |E|)$, where n and $|E|$ denote the numbers of vertices and edges, respectively.

References

[1] J. Aguirre, D. Cifuentes, V. Guigues, R. D. Monteiro, V. H. Nascimento, and A. Sujanani, *A user manual for cuhallar: A gpu accelerated low-rank semidefinite programming solver*, arXiv preprint arXiv:2508.15951, (2025).

[2] A. Alfakih, L. W. Jung, M. W. Moursi, and H. Wolkowicz, *Exact solutions for the np-hard wasserstein barycenter problem using a doubly nonnegative relaxation and a splitting method*, arXiv preprint arXiv:2311.05045, (2023).

[3] F. Alizadeh, J.-P. A. Haeberly, and M. L. Overton, *Primal-dual interior-point methods for semidefinite programming: Convergence rates, stability and numerical results*, SIAM Journal on Optimization, 8 (1998), pp. 746–768.

[4] A. I. Barvinok, *Problems of distance geometry and convex properties of quadratic maps*, Discrete & Computational Geometry, 13 (1995), pp. 189–202.

[5] S. Bhojanapalli, N. Boumal, P. Jain, and P. Netrapalli, *Smoothed analysis for low-rank solutions to semidefinite programs in quadratic penalty form*, in Conference On Learning Theory, PMLR, 2018, pp. 3243–3270.

[6] B. Borchers, *Csdp, ac library for semidefinite programming*, Optimization methods and Software, 11 (1999), pp. 613–623.

[7] J. Borwein and H. Wolkowicz, *Regularizing the abstract convex program*, Journal of Mathematical Analysis and Applications, 83 (1981), pp. 495–530.

[8] N. Boumal, V. Voroninski, and A. Bandeira, *The non-convex burer-monteiro approach works on smooth semidefinite programs*, Advances in Neural Information Processing Systems, 29 (2016).

[9] N. Boumal, V. Voroninski, and A. S. Bandeira, *Deterministic guarantees for burer-monteiro factorizations of smooth semidefinite programs*, Communications on Pure and Applied Mathematics, 73 (2020), pp. 581–608.

[10] S. Burer and R. D. Monteiro, *A nonlinear programming algorithm for solving semidefinite programs via low-rank factorization*, Mathematical programming, 95 (2003), pp. 329–357.

[11] ———, *Local minima and convergence in low-rank semidefinite programming*, Mathematical programming, 103 (2005), pp. 427–444.

[12] E. Candes and B. Recht, *Exact matrix completion via convex optimization*, Communications of the ACM, 55 (2012), pp. 111–119.

[13] E. J. Candes, X. Li, and M. Soltanolkotabi, *Phase retrieval from coded diffraction patterns*, Applied and Computational Harmonic Analysis, 39 (2015), pp. 277–299.

[14] E. J. Candes, T. Strohmer, and V. Voroninski, *Phaselift: Exact and stable signal recovery from magnitude measurements via convex programming*, Communications on Pure and Applied Mathematics, 66 (2013), pp. 1241–1274.

[15] Y. Carmon, J. C. Duchi, O. Hinder, and A. Sidford, *Accelerated methods for nonconvex optimization*, SIAM J. Optim., 28 (2018), pp. 1751–1772.

[16] Y. Chen, D. Tse, P. Nobel, P. Goulart, and S. Boyd, *Cuclarabel: Gpu acceleration for a conic optimization solver*, arXiv preprint arXiv:2412.19027, (2024).

[17] D. Cifuentes, *On the Burer–Monteiro method for general semidefinite programs*, Optimization Letters, 15 (2021), pp. 2299–2309.

[18] D. Cifuentes and A. Moitra, *Polynomial time guarantees for the Burer-Monteiro method*, Advances in Neural Information Processing Systems, 35 (2022), pp. 23923–23935.

[19] A. R. Conn, N. I. Gould, and P. Toint, *A globally convergent augmented Lagrangian algorithm for optimization with general constraints and simple bounds*, SIAM Journal on Numerical Analysis, 28 (1991), pp. 545–572.

[20] Q. Deng, Q. Feng, W. Gao, D. Ge, B. Jiang, Y. Jiang, J. Liu, T. Liu, C. Xue, Y. Ye, et al., *An enhanced admm-based interior point method for linear and conic optimization*, arXiv preprint arXiv:2209.01793, (2022).

[21] L. Ding and B. Grimmer, *Revisiting spectral bundle methods: Primal-dual (sub)linear convergence rates*, SIAM Journal on Optimization, 33 (2023), pp. 1305–1332.

[22] L. Ding and S. J. Wright, *On squared-variable formulations for nonlinear semidefinite programming*, arXiv preprint arXiv:2502.02099, (2025).

[23] L. Ding, A. Yurtsever, V. Cevher, J. A. Tropp, and M. Udell, *An optimal-storage approach to semidefinite programming using approximate complementarity*, SIAM Journal on Optimization, 31 (2021), pp. 2695–2725.

[24] A. Douik and B. Hassibi, *Low-rank riemannian optimization on positive semidefinite stochastic matrices with applications to graph clustering*, in International Conference on Machine Learning, PMLR, 2018, pp. 1299–1308.

[25] D. Drusvyatskiy, H. Wolkowicz, et al., *The many faces of degeneracy in conic optimization*, Foundations and Trends® in Optimization, 3 (2017), pp. 77–170.

[26] J. C. Duchi, O. Hinder, A. Naber, and Y. Ye, *Conic descent and its application to memory-efficient optimization over positive semidefinite matrices*, in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 8308–8317.

[27] F. R. Endor and I. Waldspurger, *Benign landscape for burer-monteiro factorizations of maxcut-type semidefinite programs*, arXiv preprint arXiv:2411.03103, (2024).

[28] M. A. Erdogdu, A. Ozdaglar, P. A. Parrilo, and N. D. Vanli, *Convergence rate of block-coordinate maximization burer–monteiro method for solving large sdps*, Mathematical Programming, 195 (2022), pp. 243–281.

[29] K. Fujisawa, Y. Futakata, M. Kojima, S. Matsuyama, S. Nakamura, K. Nakata, and M. Yamashita, *Sdpa-m (semidefinite programming algorithm in matlab) user's manual—version 6.2. 0*, Research Reports on Mathematical and Computing Sciences, Series B: Operation Res., Dep. Math. and Computing Sci., Tokyo Institute of Technol., Japan, 10 (2000).

[30] M. Fukuda, M. Kojima, K. Murota, and K. Nakata, *Exploiting sparsity in semidefinite programming via matrix completion i: General framework*, SIAM Journal on Optimization, 11 (2001), pp. 647–674.

[31] M. Garstka, M. Cannon, and P. Goulart, *Cosmo: A conic operator splitting method for convex conic problems*, Journal of Optimization Theory and Applications, 190 (2021), pp. 779–810.

[32] N. Graham, H. Hu, J. Im, X. Li, and H. Wolkowicz, *A restricted dual peaceman-rachford splitting method for a strengthened dnn relaxation for qap*, INFORMS Journal on Computing, 34 (2022), pp. 2125–2143.

[33] R. Grone, C. R. Johnson, E. M. Sá, and H. Wolkowicz, *Positive definite completions of partial hermitian matrices*, Linear Algebra and its Applications, 58 (1984), pp. 109–124.

[34] M. Grötschel, L. Lovász, and A. Schrijver, *Polynomial algorithms for perfect graphs*, In North-Holland mathematics studies, 88 (1984), pp. 325–356.

[35] J. Haegeman, *Krylovkit*, Feb. 2025.

[36] Q. Han, C. Li, Z. Lin, C. Chen, Q. Deng, D. Ge, H. Liu, and Y. Ye, *A low-rank admm splitting approach for semidefinite programming*, arXiv preprint arXiv:2403.09133, (2024).

[37] Q. Han, Z. Lin, H. Liu, C. Chen, Q. Deng, D. Ge, and Y. Ye, *Accelerating low-rank factorization-based semidefinite programming algorithms on gpu*, arXiv preprint arXiv:2407.15049, (2024).

[38] C. Helmberg, F. Rendl, R. J. Vanderbei, and H. Wolkowicz, *An interior-point method for semidefinite programming*, SIAM Journal on Optimization, 6 (1996), pp. 342–361.

[39] H. Hu, R. Sotirov, and H. Wolkowicz, *Facial reduction for symmetry reduced semidefinite and doubly nonnegative programs*, Mathematical Programming, 200 (2023), pp. 475–529.

[40] W. Huang and X. Zhang, *Solving phaselift by low-rank riemannian optimization methods for complex semidefinite constraints*, SIAM Journal on Scientific Computing, 39 (2017), pp. B840–B859.

[41] S. Kang, X. Jiang, and H. Yang, *Local linear convergence of the alternating direction method of multipliers for semidefinite programming under strict complementarity*, arXiv preprint arXiv:2503.20142, (2025).

[42] W. Kong, J. Melo, and R. Monteiro, *Complexity of a quadratic penalty accelerated inexact proximal point method for solving linearly constrained nonconvex composite programs*, SIAM J. Optim., 29 (2019), pp. 2566–2593.

[43] ———, *An efficient adaptive accelerated inexact proximal point method for solving linearly constrained nonconvex composite problems*, Comput. Optim. Appl., 76 (2019), pp. 305–346.

[44] W. Kong, J. G. Melo, and R. D. Monteiro, *Iteration complexity of a proximal augmented lagrangian method for solving nonconvex composite optimization problems with nonlinear convex constraints*, Mathematics of Operations Research, 48 (2023), pp. 1066–1094.

[45] S. Laue, *A hybrid algorithm for convex semidefinite optimization*, in Proceedings of the 29th International Conference on Machine Learning, 2012, pp. 1083–1090.

[46] E. Levin, J. Kileel, and N. Boumal, *The effect of smooth parametrizations on nonconvex optimization landscapes*, Mathematical Programming, 209 (2025), pp. 63–111.

[47] X. Li, T. K. Pong, H. Sun, and H. Wolkowicz, *A strictly contractive peaceman-rachford splitting method for the doubly nonnegative relaxation of the minimum cut problem*, Computational optimization and applications, 78 (2021), pp. 853–891.

[48] Z. Lin, Z. Xiong, D. Ge, and Y. Ye, *Pdcs: A primal-dual large-scale conic programming solver with gpu enhancements*, arXiv preprint arXiv:2505.00311, (2025).

[49] S. Ling, *Local geometry determines global landscape in low-rank factorization for synchronization*, Foundations of Computational Mathematics, (2025), pp. 1–33.

[50] H. Lu and J. Yang, *cupdlp. jl: A gpu implementation of restarted primal-dual hybrid gradient for linear programming in julia*, arXiv preprint arXiv:2311.12180, (2023).

[51] ———, *A practical and optimal first-order method for large-scale convex quadratic programming*, arXiv preprint arXiv:2311.07710, (2023).

[52] 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).

[53] A. D. McRae, *Benign landscapes for synchronization on spheres via normalized laplacian matrices*, arXiv preprint arXiv:2503.18801, (2025).

[54] A. D. McRae and N. Boumal, *Benign landscapes of low-dimensional relaxations for orthogonal synchronization on general graphs*, SIAM Journal on Optimization, 34 (2024), pp. 1427–1454.

[55] R. D. Monteiro, A. Sujanani, and D. Cifuentes, *A low-rank augmented lagrangian method for large-scale semidefinite programming based on a hybrid convex-nonconvex approach*, arXiv preprint arXiv:2401.12490, (2024).

[56] R. D. C. Monteiro, *Primal–dual path-following algorithms for semidefinite programming*, SIAM Journal on Optimization, 7 (1997), pp. 663–678.

[57] C. NVIDIA, *Programming guide, cusparse, cublas, and cufft library user guides*.

[58] L. O'Carroll, V. Srinivas, and A. Vijayaraghavan, *The burer-monteiro sdp method can fail even above the barvinok-pataki bound*, Advances in Neural Information Processing Systems, 35 (2022), pp. 31254–31264.

[59] W. Ouyang, T. K. Pong, and M.-C. Yue, *Burer-monteiro factorizability of nuclear norm regularized optimization*, arXiv preprint arXiv:2505.00349, (2025).

[60] 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 (2016), pp. 1042–1068.

[61] C. Paquette, H. Lin, D. Drusvyatskiy, J. Mairal, and Z. Harchaoui, *Catalyst for gradient-based nonconvex optimization*, in AISTATS 2018-21st International Conference on Artificial Intelligence and Statistics, 2018, pp. 1–10.

[62] G. Pataki, *On the rank of extreme matrices in semidefinite programs and the multiplicity of optimal eigenvalues*, Mathematics of operations research, 23 (1998), pp. 339–358.

[63] F. Permenter and P. Parrilo, *Partial facial reduction: simplified, equivalent sdps via approximations of the psd cone*, Mathematical Programming, 171 (2018), pp. 1–54.

[64] C. B. Pham, W. Griggs, and J. Saunderson, *A scalable frank-wolfe-based algorithm for the max-cut sdp*, in International Conference on Machine Learning, PMLR, 2023, pp. 27822–27839.

[65] T. Pumir, S. Jelassi, and N. Boumal, *Smoothed analysis of the low-rank approach for smooth semidefinite programs*, Advances in Neural Information Processing Systems, 31 (2018).

[66] B. Recht, *A simpler approach to matrix completion*, Journal of Machine Learning Research, 12 (2011), pp. 3413–3430.

[67] J. Renegar, *Accelerated first-order methods for hyperbolic programming*, Mathematical Programming, 173 (2019), pp. 1–35.

[68] R. T. Rockafellar, *Augmented Lagrangian Methods for Convex Programming*, Tech. Rep. TR 052, University of Washington, Department of Mathematics, 1973. Accessed: 2025-04-30. Related to SIAM J. Control Optim. 14(5), 1976.

[69] M. Schubiger, G. Banjac, and J. Lygeros, *Gpu acceleration of admm for large-scale quadratic programming*, Journal of Parallel and Distributed Computing, 144 (2020), pp. 55–67.

[70] A. Shapiro, *Rank-reducibility of a symmetric matrix and sampling theory of minimum trace factor analysis*, Psychometrika, 47 (1982), pp. 187–199.

[71] N. Shinde, V. Narayanan, and J. Saunderson, *Memory-efficient structured convex optimization via extreme point sampling*, SIAM Journal on Mathematics of Data Science, 3 (2021), pp. 787–814.

[72] A. Sujanani and R. Monteiro, *An adaptive superfast inexact proximal augmented Lagrangian method for smooth nonconvex composite optimization problems*, J. Scientific Computing, 97 (2023).

[73] T. Tang and K.-C. Toh, *Exploring chordal sparsity in semidefinite programming with sparse plus low-rank data matrices*, arXiv preprint arXiv:2410.23849, (2024).

[74] ———, *A feasible method for general convex low-rank sdp problems*, SIAM Journal on Optimization, 34 (2024), pp. 2169–2200.

[75] ———, *A feasible method for solving an sdp relaxation of the quadratic knapsack problem*, Mathematics of Operations Research, 49 (2024), pp. 19–39.

[76] ———, *Solving graph equipartition sdps on an algebraic variety*, Mathematical programming, 204 (2024), pp. 299–347.

[77] M. J. Todd, K. C. Toh, and R. H. Tütüncü, *On the nesterov–todd direction in semidefinite programming*, SIAM Journal on Optimization, 8 (1998), pp. 769–796.

[78] L. Vandenberghe, M. S. Andersen, et al., *Chordal graphs and semidefinite optimization*, Foundations and Trends® in Optimization, 1 (2015), pp. 241–433.

[79] I. Waldspurger and A. Waters, *Rank optimality for the burer–monteiro factorization*, SIAM journal on Optimization, 30 (2020), pp. 2577–2602.

[80] A. L. Wang and F. Kilinç-Karzan, *Accelerated first-order methods for a class of semidefinite programs*, Mathematical Programming, 209 (2025), pp. 503–556.

[81] J. Wang and L. Hu, *Solving low-rank semidefinite programs via manifold optimization*, arXiv preprint arXiv:2303.01722, (2023).

[82] Y. Wang, K. Deng, H. Liu, and Z. Wen, *A decomposition augmented lagrangian method for low-rank semidefinite programming*, SIAM Journal on Optimization, 33 (2023), pp. 1361–1390.

[83] Z. Wen, D. Goldfarb, and W. Yin, *Alternating direction augmented lagrangian methods for semidefinite programming*, Mathematical Programming Computation, 2 (2010), pp. 203–230.

[84] M. Yamashita, K. Fujisawa, M. Fukuda, K. Kobayashi, K. Nakata, and M. Nakata, *Latest Developments in the SDPA Family for Solving Large-Scale SDPs*, Springer US, New York, NY, 2012, pp. 687–713.

[85] M. Yamashita, K. Fujisawa, K. Nakata, M. Nakata, M. Fukuda, K. Kobayashi, and K. Goto, *A high-performance software package for semidefinite programs: Sdpa 7*, (2010).

[86] L. Yang, D. Sun, and K.-C. Toh, *Sdpnal+: a majorized semismooth newton-cg augmented lagrangian method for semidefinite programming with nonnegative constraints*, Mathematical Programming Computation, 7 (2015), pp. 331–366.

[87] A. Yurtsever, O. Fercoq, and V. Cevher, *A conditional-gradient-based augmented lagrangian framework*, in International Conference on Machine Learning, PMLR, 2019, pp. 7272–7281.

[88] A. Yurtsever, Y.-P. Hsieh, and V. Cevher, *Scalable convex methods for phase retrieval*, in 2015 IEEE 6th International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP), IEEE, 2015, pp. 381–384.

[89] A. Yurtsever, J. A. Tropp, O. Fercoq, M. Udell, and V. Cevher, *Scalable semidefinite programming*, SIAM Journal on Mathematics of Data Science, 3 (2021), pp. 171–200.

[90] R. Y. Zhang, *Complexity of chordal conversion for sparse semidefinite programs with small treewidth*, Mathematical Programming, (2024), pp. 1–37.

[91] R. Y. Zhang and J. Lavaei, *Sparse semidefinite programs with guaranteed near-linear time complexity via dualized clique tree conversion*, Mathematical programming, 188 (2021), pp. 351–393.

[92] X.-Y. Zhao, D. Sun, and K.-C. Toh, *A newton-cg augmented lagrangian method for semidefinite programming*, SIAM Journal on Optimization, 20 (2010), pp. 1737–1765.

[93] Y. Zheng, G. Fantuzzi, A. Papachristodoulou, P. Goulart, and A. Wynn, *Fast admm for semidefinite programs with chordal sparsity*, in 2017 American Control Conference (ACC), IEEE, 2017, pp. 3335–3340.

[94] ———, *Chordal decomposition in operator-splitting methods for sparse semidefinite programs*, Mathematical Programming, 180 (2020), pp. 489–532.