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)

Abstract

This paper presents an SDP solver, cuHALLaR, which is a GPU-accelerated implementation of the hybrid low-rank augmented Lagrangian method, HALLaR, proposed earlier by three of the authors. The proposed Julia-based implementation is designed to exploit massive parallelism by performing all core operator evaluations—linear maps, their adjoints, and gradients—entirely on the GPU. Extensive numerical experiments are conducted on three problem classes: matrix completion, maximum stable set, and phase retrieval. The results demonstrate substantial performance gains over both optimized CPU implementations and existing GPU-based solvers. On the largest instances, cuHALLaR achieves speedups of up to 165x, solving a matrix completion SDP with a matrix variable of size $2 \times 10^6 \times 2 \times 10^6$ and over 2.6×10^8 constraints to a relative precision of 10^{-5} in 53 seconds. Similarly, speedups of up to 135x are observed for maximum stable set problems on graphs with over one million vertices. These results establish cuHALLaR as a state-of-the-art solver for extremely large-scale SDPs.

Keywords: semidefinite programming, augmented Lagrangian, low-rank methods, gpu acceleration, Frank-Wolfe method

1 Introduction

Let $\mathbb{S}^n$ be the set of $n \times n$ symmetric matrices. The notation $A \succeq B$ means that $A - B$ is positive semidefinite. We are interested in solving the primal-dual pair of semidefinite programs (SDPs)

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

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 $\bullet$ is the Frobenius inner product, $b \in \mathbb{R}^m$, $C \in \mathbb{S}^n$, and $\mathcal{A} : \mathbb{S}^n \to \mathbb{R}^m$ and $\mathcal{A}^* : \mathbb{R}^m \to \mathbb{S}^n$ are linear maps such that

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

*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

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

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, 34, 50, 67]. 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, 18, 19, 21, 22, 25, 28, 36, 29, 41, 37, 54, 57, 59, 61, 73, 76, 77, 79, 82, 83].

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 [13, 33, 42, 46, 45, 44, 60]. In this paper, we introduce an enhanced Julia-based GPU implementation, referred to as cuHALLaR, of the first-order HALLaR method developed in [49] for solving huge-scale SDPs. Our work reinforces the viability of using GPU architectures for efficient parallelization in semidefinite programming solvers. Moreover, our experiments show potential for excellent speedups when using GPUs for optimization solvers.

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 [33], 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 [44, 46, 45] have achieved performance which 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 [60] and CuClarabel [13] similarly exploit GPU parallelism for significant speedups.

Most recently, [42] 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 [68, 27, 30, 81, 84, 80, 35, 7, 23, 63, 56]. 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) [15, 16, 8, 9, 5, 58, 69]. For a more extensive list of papers which characterize the optimization landscape of nonconvex formulations obtained through Burer-Monteiro factorizations, see [53, 24, 47, 43, 52, 20, 48, 40]. 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 [25, 32, 33, 49, 64, 65, 66, 71, 72, 78, 70, 77, 79].

We now give a brief comparison between HALLaR, which was first developed in [49] and the LoRADS method developed in [32]. 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 the these spurious solutions.

cuHALLaR versus cuLoRADS and HALLaR Like cuLoRads, which is a GPU accelerated version of LoRads developed in [33], 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 numerical results show that cuHALLaR is a very promising GPU-based method for solving extremely large SDP instances.

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 $\mathbb{R}^n$. Given matrices $A_1, \ldots, A_m \in \mathbb{S}^n$, let $\mathcal{A} : \mathbb{S}^n \to \mathbb{R}^m$ denote the operator defined as $[\mathcal{A}(X)]_i = A_i \bullet X$ for every $i = 1, \ldots, m$. The adjoint operator $\mathcal{A}^* : \mathbb{R}^m \to \mathbb{S}^n$ of $\mathcal{A}$ is given by $\mathcal{A}^*(\lambda) = \sum_{i=1}^m \lambda_i A_i$. 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 C at $z \in C$, denoted by $N_C^\epsilon(z)$, is

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

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

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

We denote the i-th row of a matrix U by $U_{i,:}$.

2 Overview of HALLaR

This section provides a minimal review of HALLaR, the hybrid low-rank augmented Lagrangian method that was first developed in [49].

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

$$\|\mathcal{A}\bar{X} - b\| \le \epsilon_p, \quad |C \bullet \bar{X} + b^\top \bar{p} + \tau^2 \bar{\theta}| \le \epsilon_{pd}, \quad C + \mathcal{A}^* \bar{p} + \bar{\theta} I \succeq 0, \quad \bar{\theta} \ge 0. \tag{4}$$

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 (4) 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

$$X_t \quad \approx \quad \arg\min_X \quad \{\mathcal{L}_{\beta_t}(X; p_{t-1}) \ : \ X \in \Delta^n_\tau\}, \tag{5a}$$

$$p_t \quad = \quad p_{t-1} + \beta_t(\mathcal{A}(X_t) - b) \tag{5b}$$

where

$$\mathcal{L}_\beta(X; p) \quad := \quad C \bullet X + p^\top(\mathcal{A}(X) - b) + \frac{\beta}{2}\|\mathcal{A}X - b\|^2 \tag{6}$$

is the AL function. 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 (5a). 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 $s_t \ll n$. The next paragraph briefly describes HLR in more detail for solving (5a).

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 computes a suitable stationary point Y of the nonconvex reformulation of (5a) obtained through the change of variable $X = UU^\top$, namely:

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

(ii) it verifies if the point $X = YY^\top$ corresponds to a near-global minimizer of the original convex AL subproblem (5a). 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 (L_r), 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 (L_r). The minimum eigenvector computed in the previous step furnishes a descent direction for the convex problem (5a). 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). This step is implemented with the aid of a generic nonlinear solver which, started from $\tilde{Y}$, computes a $(\rho; \tilde{Y})$-stationary solution of (L_r), i.e., a triple $(Y; R, \rho)$ such that

$$R \in \nabla g_t(Y) + N_{B^s_\tau}(Y), \quad \|R\|_F \le \rho, \quad g_t(Y) \le g_t(\tilde{Y}). \tag{7}$$

It is easy to verify that (7), in terms of the SDP data, is equivalent to

$$R \in 2[C + \mathcal{A}^*(p_{t-1} + \beta_t(\mathcal{A}(YY^\top) - b))]Y + N_{B^s_\tau}(Y), \quad \|R\|_F \le \rho.$$

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 [49, 62], and also in earlier works (see e.g. [12, 55, 38, 39]) in other related forms.

The details of steps ii) and iii) implemented in the X-space are quite standard. The details of its implementation in the U-space are described in the formal description of the algorithm given below. The only point worth observing at this stage is that, due to the rank-one update nature of the Frank-Wolfe method, 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). Hence, in the usual case, the number of columns of $\tilde{Y}$ increases by one.

We now formally state HALLaR below.

Algorithm 1 HALLaR.

1: **Input:** Let initial points $(U_0, p_0) \in B_\tau^{s_0} \times \mathbb{R}^m$, tolerance pair $(\epsilon_{\rm p}, \epsilon_{\rm pd}) \in \mathbb{R}^2_{++}$, let $\{\beta_t\}_{t\geq 1}$ be a sequence of positive reals, and let $\{\epsilon_t\}_{t\geq 1}$ be a decreasing sequence of positive reals converging to 0.
2: **Output:** $(\bar{X}, \bar{p}, \bar{\theta}) \in \Delta_\tau^n \times \mathbb{R}^m \times \mathbb{R}_+$, an $(\epsilon_{\rm p}, \epsilon_{\rm pd})$-solution of the pair of SDPs (P) and (D).
3: set $t \leftarrow 1$;
4: set $\tilde{Y} = U_{t-1}$, $s = s_{t-1}$, $Y = 0_{n\times s}$, $G = 0_{n\times n}$, and $\theta = \infty$;
5: **while** $G \bullet (YY^\top) + \tau^2\theta > \epsilon_t$ **do** ▷ **(HLR method)**
6: call a nonconvex solver with initial point $\tilde{Y}$, tolerance ϵ_t, and function $\mathcal{L}_{\beta_t}(UU^\top; p_{t-1}) + \delta_{B_\tau^s}(U)$ to find a point $Y \in B_\tau^s$ that is an ϵ_t-approximate stationary solution (according to the criterion in (7)) of

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

7: compute

$$G := \nabla\left[\mathcal{L}_{\beta_t}(\,\cdot\,; p_{t-1})\right]\left(YY^\top\right) \in \mathbb{S}^n \tag{9}$$

and a minimum eigenpair of G, i.e., $(\lambda_{\min}(G), v_{\min}(G)) \in \mathbb{R} \times \mathbb{R}^n$, and set

$$\theta := \max\{-\lambda_{\min}(G), 0\}, \qquad y := \begin{cases} v_{\min}(G) & \text{if } \theta > 0 \\ 0 & \text{otherwise;} \end{cases} \tag{10}$$

8: **if** $G \bullet (YY^\top) + \tau^2\theta \leq \epsilon_t$ **then**
9: **break while loop** and go to step 12 below;
10: **end if**
11: compute

$$\alpha = \min\left\{\frac{G \bullet (YY^\top) + \tau^2\theta}{\beta_t\|\mathcal{A}(YY^\top) - \mathcal{A}(\tau^2 yy^\top)\|^2},\ 1\right\} \tag{11}$$

and set

$$(\tilde{Y}, s) = \begin{cases} (\tau y, 1) & \text{if } \alpha = 1 \\ \left(\left[\sqrt{1-\alpha}\,Y, \sqrt{\alpha}\,\tau\, y\right], s+1\right) & \text{otherwise;} \end{cases} \tag{12}$$

12: **end while**
13: set $(U_t, \theta_t, s_t) = (Y, \theta, s)$;
14: set

$$p_t = p_{t-1} + \beta_t(\mathcal{A}(U_t U_t^\top) - b); \tag{13}$$

15: if

$$\|\mathcal{A}(U_t U_t^\top) - b\| \leq \epsilon_{\rm p}, \quad |C \bullet (U_t U_t^\top) - (-b^\top p_t - \tau^2\theta_t)| \leq \epsilon_{\rm pd} \tag{14}$$

then **return** $(\bar{U}, \bar{p}, \bar{\theta}) = (U_t, p_t, \theta_t)$;
16: set $t = t + 1$ and **go to** step **3.**

Several remarks about each of the steps of HALLaR are now given. First, t is the iteration count for HALLaR. Second, in practice we choose the sequence $\{\beta_t\}_{t\geq 1}$ in an adaptive manner based on some of the ideas of the LANCELOT method [17]. Third, the while loop in lines 5 to 12 of HALLaR corresponds to the HLR method, which is used by HALLaR to approximately solve its AL subproblems in (5a). In line 4, the triple (Y, G, θ) is set as $(0_{n\times s}, 0_{n\times n}, \infty)$ to ensure that the HLR method is always executed during each iteration of HALLaR. Fourth, the nonconvex solver in line 6 of HALLaR will typically make several evaluations of $\nabla_U \mathcal{L}_{\beta_t}(UU^\top; p_{t-1})$, which is easily seen to be equal to

$$\nabla_U \mathcal{L}_{\beta_t}(UU^\top; p_{t-1}) := 2\left[C + \mathcal{A}^*\left(p_{t-1} + \beta_t\left(\mathcal{A}(UU^\top) - b\right)\right)\right]U. \tag{15}$$

Fifth, it follows from the definition of $\mathcal{L}_{\beta_t}(\cdot; p_{t-1})$ that G in (9) can be written as

$$G := C + \mathcal{A}^*\left(p_{t-1} + \beta_t\left(\mathcal{A}(YY^\top) - b\right)\right). \tag{16}$$

Sixth, after the HLR method approximately solves an AL subproblem, HALLaR updates the Lagrange multiplier in line 14 and checks for termination in line 15. Lastly, the termination condition on this line does not check for near dual feasibility since it can be shown that for every $t \geq 1$,

$$S_t := C + \mathcal{A}^* p_t + \theta_t I \succeq 0, \quad \theta_t \geq 0, \tag{17}$$

and hence that HALLaR always generates feasible dual iterates (p_t, θ_t, S_t).

We now comment on the steps performed within the while loop (lines 5 to 12) in light of the three steps of the HALLaR outline given just before its formal description. Step 6 implements step i) of the outline. Steps 7 through 9 implement step ii). Finally, step 11 implements step iii).

We now give an interpretation of step 11 of HALLaR relative to the X-space. The Frank-Wolfe procedure generates a new iterate by moving from the current point $YY^\top$ towards an extreme point of the feasible set Δ_τ^n, which is of the form $\tau^2 yy^\top$. The stepsize α computed in (11) corresponds to solving

$$\arg\min_{\hat{\alpha}\in[0,1]} \left\{ \mathcal{L}_{\beta_t}\left((1-\hat{\alpha})YY^\top + \hat{\alpha}(\tau yy^\top); p_{t-1}\right)\right\}.$$

Moreover, the iterate $\tilde{Y}$ computed in equation (12) has the property that $\tilde{Y}\tilde{Y}^\top$ is equal to the usual Frank-Wolfe iterate $(1-\alpha)YY^\top + \alpha(\tau^2 yy^\top)$ in the X-space; hence $\tilde{Y}$ is a valid U-factor for the Frank-Wolfe iterate.

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 in the GPU as functions which 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. Introduced in [74, 75, 26], the SDPA format 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, for SDP instances with a dense but well-structured data matrix (e.g., the matrix of all ones or the identity matrix plus a low-rank matrix), it becomes prohibitively expensive to store and handle their input data in SDPA format. 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.

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:

$$A_\ell = A_\ell^{\text{sp}} + A_\ell^{\text{lr}}, \qquad A_\ell^{\text{lr}} = K_\ell D_\ell K_\ell^\top, \qquad \ell = 0, 1, \ldots, m \tag{18}$$

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. 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**: HSLR format has a significantly lower memory footprint than SDPA format. Specifically, if a data matrix A_ℓ is such that $A_\ell^{\text{lr}} \neq 0$ and D_ℓ is diagonal, then it requires $n_\ell + r_\ell n + r_\ell$ entries (resp., n(n+1)/2) to be stored in HSLR (resp., SDPA) format, where n_ℓ is the number of nonzeros of A_ℓ^{sp}.

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 $\tilde{\mathcal{A}}(UU^\top)$ and $\tilde{\mathcal{A}}^*(\tilde{p})U$

The efficient evaluations of the linear maps $\mathcal{A}(X) : \mathbb{S}^n \to \mathbb{R}^m$ and $\mathcal{A}^*(p) : \mathbb{R}^m \to \mathbb{S}^n$ as in (1) are essential to the performance of cuHALLaR. To avoid the formation of a possibly dense $n \times n$ matrix X, cuHALLaR forms a low-rank factor $U \in \mathbb{R}^{n\times r}$ of X that satisfies $X = UU^\top$. At every iteration, cuHALLaR has to evaluate for a given $(U, p) \in \mathbb{R}^{n\times r} \times \mathbb{R}^m$ the quantities $\tilde{\mathcal{A}}(UU^\top)$ and $\tilde{\mathcal{A}}^*(\tilde{p})U$ where $\tilde{p} = (1, p) \in \mathbb{R}^{m+1}$, and $\tilde{\mathcal{A}} : \mathbb{S}^n \to \mathbb{R}^{m+1}$ and $\tilde{\mathcal{A}}^* : \mathbb{R}^{m+1} \to \mathbb{S}^n$ are given by

$$\tilde{\mathcal{A}}(X) := \begin{bmatrix} A_0 \bullet X \\ A_1 \bullet X \\ \vdots \\ A_m \bullet X \end{bmatrix} := \begin{bmatrix} A_0 \bullet X \\ \mathcal{A}(X) \end{bmatrix}, \qquad \tilde{\mathcal{A}}^*(\tilde{p}) := \sum_{\ell=0}^{m} \tilde{p}_\ell A_\ell = A_0 + \sum_{\ell=1}^{m} p_\ell A_\ell = A_0 + \mathcal{A}^*(p) \tag{19}$$

where $A_0 := C$ and $\mathcal{A}(\cdot)$ and $\mathcal{A}^*(\cdot)$ are as in (1). It follows from the definitions of $\nabla_U \mathcal{L}_{\beta_t}(UU^\top; p_{t-1})$ and G in (15) and (16), respectively, that cuHALLaR has to evaluate $\tilde{\mathcal{A}}(UU^\top)$ in steps 6, 7, 11, and 14, and $\tilde{\mathcal{A}}^*(\tilde{p})U$ in steps 6 and 11.

This section presents the details of the efficient implementation of the operations in (19) in the GPU, using the fact that the data matrices A_ℓ are assumed to be the sum of a sparse and a low-rank matrix as in (18).

4.1 Constraint Evaluation $\tilde{\mathcal{A}}(UU^\top)$

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 (19). It follows from (18) and the first relation in (1) that $T = \tilde{\mathcal{A}}(UU^\top)$ can be computed as $T = T^{sp} + T^{\text{lr}}$ where the ℓ-th components of T^{sp} and T^{lr} are defined as:

$$T_\ell^{sp} = A_\ell^{sp} \bullet (UU^\top), \quad T_\ell^{\text{lr}} = A_\ell^{\text{lr}} \bullet (UU^\top), \quad \ell = 0, 1, \ldots, m. \tag{20}$$

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^{sp}, $T_\ell^{\text{sp}} = A_\ell^{\text{sp}} \bullet (UU^\top)$, is then computed as:

$$T_\ell^{\text{sp}} = \sum_{i=1}^{n} \sum_{j=1}^{n} (A_\ell^{\text{sp}})_{ij} B_{ij}. \tag{21}$$

Since A_ℓ^{sp} and B are symmetric matrices, this summation can be rewritten by defining the weights

$$\omega_{ij,\ell} := (2 - \delta_{ij})(A_\ell^{\text{sp}})_{ij}, \text{ where } \delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \tag{22}$$

Using (22), we see that $T_\ell^{\text{sp}} = \sum_{i \leq j} \omega_{ij,\ell} B_{ij}$.

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

$$\mathcal{S} := \bigcup_{\ell=0}^{m} \operatorname{supp}(A_\ell^{\text{sp}}) \cap \{(i,j) : 1 \le i \le j \le n\}. \tag{23}$$

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:

$$\mathcal{L}_{ij} := \{\ell : (i,j) \in \operatorname{supp}(A_\ell^{\text{sp}})\}. \tag{24}$$

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_ℓ^{sp} for all $\ell \in \mathcal{L}_{ij}$. 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 (19).

Input: $U \in \mathbb{R}^{n\times r}$, data matrices $\{A_\ell^{\text{sp}}\}_{\ell=0}^m \subseteq \mathbb{S}^n$, and precomputed structures $\mathcal{S}$ and $\{\mathcal{L}_{ij}\}$ from (23)–(24).
Output: $T^{sp} \in \mathbb{R}^{m+1}$ where $T_\ell^{sp} = A_\ell^{sp} \bullet UU^\top$.
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 $\ell \in \mathcal{L}_{ij}$ **do**
        $\omega_{ij,\ell} \leftarrow (2-\delta_{ij})(A_\ell^{\text{sp}})_{ij}$
        $T_\ell^{sp} \leftarrow T_\ell^{sp} + \omega_{ij,\ell} B_{ij}$
    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 $|\mathcal{L}_{ij}| \ll m+1$ that updates T_ℓ^{sp}.

Low-Rank Component. This paragraph describes how cuHALLaR efficiently computes $T^{\text{lr}} \in \mathbb{R}^{m+1}$. Using the second relation in (18), the ℓ-th component of T^{lr}, $T_\ell^{\text{lr}} = A_\ell^{\text{lr}} \bullet (UU^\top)$, can be computed as

$$T_\ell^{\text{lr}} = \operatorname{Tr}(K_\ell D_\ell K_\ell^\top (UU^\top)) = \operatorname{Tr}(D_\ell (K_\ell^\top U)(U^\top K_\ell)). \tag{25}$$

Defining $Y_\ell := K_\ell^\top U \in \mathbb{R}^{r_\ell \times r}$, relation (25) further simplifies to:

$$T_\ell^{\text{lr}} = \operatorname{Tr}(D_\ell Y_\ell Y_\ell^\top). \tag{26}$$

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 (19)

Input: $U \in \mathbb{R}^{n\times r}$ and problem data $\{K_\ell, D_\ell\}_{\ell=0}^m$.
Output: $T^{lr} \in \mathbb{R}^{m+1}$ where $T_\ell^{lr} = A_\ell^{lr} \bullet UU^\top$.
Initialize $T^{lr} \leftarrow 0 \in \mathbb{R}^{m+1}$.
for $\ell = 0$ to m **do**
    $Y_\ell \leftarrow K_\ell^\top U$ ▷ in parallel
    $T_\ell^{lr} \leftarrow 0 + \operatorname{Tr}(D_\ell Y_\ell Y_\ell^\top)$
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 $\operatorname{Tr}(D_\ell Y_\ell Y_\ell^\top)$ are performed in parallel. In our numerical experiments, we found that parallelizing these computations was very efficient.

4.2 Adjoint $\tilde{\mathcal{A}}^*(\tilde{p})U$

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 (19), where $\tilde{p} = (1, p) \in \mathbb{R}^{m+1}$. It follows from (18) and the second identity in (19) that W is given by

$$W = \underbrace{\left(\sum_{\ell=0}^{m} \tilde{p}_\ell A_\ell^{\text{sp}}\right) U}_{W^{\text{sp}}} + \underbrace{\sum_{\ell=0}^{m} \tilde{p}_\ell (K_\ell D_\ell K_\ell^\top) U}_{W^{\text{lr}}} . \tag{27}$$

The remaining part of this subsection discusses how cuHALLaR implements efficient subroutines to compute W^{sp} and W^{lr} separately.

Sparse Component. To compute W^{sp} as in (27), we first allocate a sparse symmetric matrix M with nonzero entries at indices (i, j), where $\mathcal{L}_{ij}$ in (24) is nonempty. After doing this allocation, we then update the values of the nonzero entries of M_{ij} in parallel using the formula:

$$M_{ij} = \sum_{\ell\in\mathcal{L}_{ij}} \tilde{p}_\ell (A_\ell^{\text{sp}})_{ij}. \tag{28}$$

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

Algorithm 4 Computation of W^{sp} as in (27).

1: **Input:** $U \in \mathbb{R}^{n\times r}$, $\tilde{p} = (1, p) \in \mathbb{R}^{m+1}$ where $p \in \mathbb{R}^m$, $\{A_\ell^{\text{sp}}\}_{\ell=0}^m \subseteq \mathbb{S}^n$, and precomputed structures $\mathcal{S}$, $\{\mathcal{L}_{ij}\}$ from (23)–(24).
2: **Output:** $W^{sp} \in \mathbb{R}^{n\times r}$ where $W^{sp} = (\sum_{\ell=0}^m \tilde{p}_\ell A_\ell^{\text{sp}})\, U$.
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** $\ell \in \mathcal{L}_{ij}$ **do**
7: $\quad\quad w \leftarrow w + \tilde{p}_\ell (A_\ell^{\text{sp}})_{ij}$
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 $|\mathcal{L}_{ij}| \ll m + 1$ to compute $w = \sum_{\ell\in\mathcal{L}_{ij}} \tilde{p}_\ell (A_\ell^{\text{sp}})_{ij}$ 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 (27) and defining $Y_\ell = K_\ell^\top U \in \mathbb{R}^{r_\ell\times r}$, it follows that W^{lr} can be computed as

$$W^{\text{lr}} = \sum_{\ell=0}^{m} \tilde{p}_\ell K_\ell (D_\ell Y_\ell). \tag{29}$$

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

Algorithm 5 Computation of W^{lr} as in (27).

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 $\{K_\ell, D_\ell\}_{\ell=0}^m$.
Output: $W^{\mathrm{lr}} \in \mathbb{R}^{n\times r}$ where $W^{\mathrm{lr}} = \left(\sum_{\ell=0}^m \tilde{p}_\ell A_\ell^{\mathrm{lr}}\right) U$.
Initialize $W^{\mathrm{lr}} \leftarrow 0_{n\times r}$.
for $\ell = 0$ to m **do**
    $Y_\ell \leftarrow K_\ell^\top U$
    $Z_\ell \leftarrow D_\ell Y_\ell$ ▷ in parallel
    $W^{\mathrm{lr}} \leftarrow W^{\mathrm{lr}} + \tilde{p}_\ell (K_\ell Z_\ell)$
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.

4.3 Kernels for problems in SDPA format

This subsection describes cuHALLaR's efficient approach for dealing with SDP instances given in sparse SDPA format. The parser reads each data matrix A_ℓ, for $\ell = 0, ..., m$ and computes its density as

$$d_\ell = \frac{nnz(A_\ell)}{n^2}. \tag{30}$$

The kernels for SDPA format separate constraints into sparse or dense. Let A_ℓ^{dn} be the representation of data matrix A_ℓ in dense format. If $d_\ell > 10^{-1}$, cuHALLaR sets $A_\ell = A_\ell^{dn}$; otherwise it sets $A_\ell = A_\ell^{\mathrm{sp}}$. The quantities $\mathcal{S}$ and $\{\mathcal{L}_{ij}\}$ in (23) and (24) are precomputed from the data matrices which are determined to be sparse. The evaluations of $\tilde{\mathcal{A}}(UU^\top)$ and $\tilde{\mathcal{A}}^*(\tilde{p})U$ use similar approaches as the ones described in Section 4. The sparse components of these evaluations are performed using Algorithms 2 and 4 while the dense components of these evaluations are evaluated using adaptations of Algorithms 3 and 5, where A_ℓ^{dn} replaces $A_\ell^{lr} = K_\ell D_\ell K_\ell^\top$.

5 Numerical Experiments

In this section, we provide extensive experiments comparing cuHALLaR, HALLaR, and cuLoRADS on three problem classes: matrix completion, maximum stable set, and phase retrieval. These problems were also used in the benchmark of the original HALLaR paper [49]. The pre-compiled binary solver can be found at `https://github.com/OPTHALLaR`.

We attempted to run CuClarabel [13] on the matrix completion and stable set experiments, but CuClarabel failed even on relatively small SDPA files. Moreover, CuClarabel's lack of native SDPA support precluded straightforward benchmarking, so we omit comparisons against it.

The comparisons between cuHALLaR and cuLoRADS evaluate cuHALLaR using the HSLR format described in Section 4 against cuLoRADS using SDPA format. This comparison highlights the performance and memory efficiency advantages of our HSLR representation and cuHALLaR's implementation.

5.1 Experimental Setup

Hardware and software. 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.

We implemented cuHALLaR as a Julia [4] module. Our implementation uses the CUDA platform [14] for interfacing with NVIDIA CUDA GPUs. The experiments are performed using CUDA 12.6.1 and Julia 1.11.3. The reported running times do not take into account the pre-compilation time.

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

$$\max\left\{\frac{\|\mathcal{A}X - b\|_2}{1 + \|b\|_1}, \frac{|\text{pval} - \text{dval}|}{1 + |\text{pval}| + |\text{dval}|}, \frac{\max\{0, -\lambda_{\min}(S)\}}{1 + \|C\|_1}\right\} \leq \epsilon, \tag{31}$$

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 $\text{dval} = -b^\top p$.

Initialization. Both cuHALLaR and HALLaR are initialized by setting the dual variable to $p_0 = 0$ and the primal factor U_0 to an $n \times 1$ matrix with entries drawn independently from a standard Gaussian distribution.

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 our user manual [1].

Our computational experiments also report results for HALLaR and cuHALLaR versions (not available to the public) where the data is 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 $U \in \mathbb{R}^{n\times r}$. Moreover, these two "on-the-fly" versions have the advantage of being able to run the phase retrieval experiments, which are expensive to store in either SDPA or HSLR.

5.2 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:

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

The above problem can be rephrased as the following SDP:

$$\min_{X\in\mathbb{S}^{n_1+n_2}} \left\{\frac{1}{2}\text{Tr}(X) : X = \begin{pmatrix} W_1 & Y \\ Y^\top & W_2 \end{pmatrix} \succeq 0\,, \quad Y_{i,j} = M_{i,j}\,,\ \forall\,(i,j) \in \Omega)\right\}. \tag{32}$$

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 $UV^\top$, where the matrices $U \in \mathbb{R}^{n_1\times r}$ and $V \in \mathbb{R}^{n_2\times r}$ have independent standard Gaussian random variables as entries. Afterward, m independent and uniformly random entries from M are taken, where $m = \lceil \gamma\, r(n_1 + n_2 - r)\rceil$, and $\gamma = r\log(n_1 + n_2)$ is the oversampling ratio.

Table 2 compares HALLaR, cuHALLaR, and cuLoRADS on matrix completion problems. 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 cuHALLaR, using both HSLR and SDPA formats via our binary, against cuLoRADS, which uses SDPA format. Here, cuHALLaR with HSLR format consistently outperforms cuLoRADS

across all instances, achieving speedups ranging from 0.9x to 37x. Notably, cuHALLaR efficiently finds near-optimal solutions with much smaller ranks than the ones found cuLoRADS. For the largest instances considered in the table below the midrule line, cuHALLaR with HSLR format achieves 2x–4x speedups over cuLoRADS.

Problem Instance		**HALLaR vs cuHALLaR (on-the-fly)**			**cuHALLaR vs cuLoRADS**			
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 $\epsilon = 10^{-5}$. Time limit is set at three hours. Ratio is computed as cuLoRADS (SDPA) / cuHALLaR (HSLR).

5.3 Experiments for maximum stable set

Given a graph $G = ([n], E)$, 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

$$\max_{X \in \mathbb{S}^n} \quad \{ee^\top \bullet X \ : \ X_{ij} = 0, \ ij \in E, \ \mathrm{Tr}(X) \leq 1, \ X \succeq 0\} \tag{33}$$

where $e = (1, 1, \ldots, 1)^\top \in \mathbb{R}^n$ is the all ones vector. It was shown in [31] 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: Runtimes (in seconds) for the Maximum stable set problem. A relative tolerance of $\epsilon = 10^{-5}$ is set. Ratio is computed as HALLaR/ cuHALLaR.

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 HALLaR, cuHALLaR, and cuLoRADS on small Hamming graph instances, with the HALLaR and cuHALLaR benchmarks conducted using on-the-fly problem generation. For small graphs, such as $H_{10,2}$ through $H_{13,2}$, the CPU-based HALLaR outperforms cuHALLaR, reflecting the overhead of GPU parallelization on modest problem sizes. A clear performance transition occurs at $H_{14,2}$, where cuHALLaR begins to demonstrate superior performance with a 1.21x speedup. This advantage grows exponentially as problem size increases, reaching an impressive 50.1x speedup over HALLaR for the $H_{23,2}$ instance, which has over 8 million vertices and 96 million

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: Runtimes (in seconds) 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)
$H_{10,2}$(1,024; 5,120)	0.153/2	1.735/2	0.088	0.798/2
$H_{11,2}$(2,048; 11,264)	0.260/2	1.627/2	0.160	0.92/2
$H_{12,2}$(4,096; 24,576)	0.606/2	2.033/2	0.298	1.83/2
$H_{13,2}$(8,192; 53,248)	1.852/2	3.106/2	0.596	1.10/2
$H_{14,2}$(16,384; 114,688)	3.843/2	3.170/2	**1.212**	1.28/2
$H_{15,2}$(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: Runtimes (in seconds) for the Maximum Stable Set problem on Hamming graphs. A relative tolerance of $\epsilon = 10^{-5}$ is set. Ratio in the fourth column is computed as HALLaR/ cuHALLaR. The last column provides two values in each line: the first one is the run time using the HSLR format and the second one is the rank of the solution.

edges. Notably, cuLoRADS encounters out-of-memory limitations for problems larger than $H_{14,2}$, highlighting the superior memory efficiency of our implementations. For the problems cuLoRADS could solve, it took significantly longer and found solutions with higher ranks than both HALLaR and cuHALLaR.

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: Runtimes (in seconds) for the Maximum Stable Set problem on GSET instances. Tolerances are set to 10^{-5}. Time limit was set to 1 hour (3600 seconds). An entry marked with $*/N$ means the corresponding method finds an approximate solution with relative accuracy strictly larger than the desired accuracy in which case N expresses the maximum of the three relative accuracies in (31). Ratio in the fourth column is computed as HALLaR/ cuHALLaR. Ratio in the last column is computed as cuLoRADS/ cuHALLaR.

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.

5.4 Experiments for phase retrieval

Given m pairs $\{(a_i, b_i)\}_{i=1}^m \subseteq \mathbb{C}^n \times \mathbb{R}_+$, this subsection considers the problem of finding a vector $x \in \mathbb{C}^n$ such that

$$|\langle a_i, x\rangle|^2 = b_i, \quad i = 1, \ldots, m.$$

In other words, the goal is to retrieve x from the magnitude of m linear measurements. By creating the complex Hermitian matrix $X = xx^H$, this problem can be approached by solving the complex-valued SDP relaxation

$$\min_{X \in \mathbb{S}^n(\mathbb{C})} \quad \left\{ \mathrm{Tr}(X) \quad : \quad \langle a_i a_i^H, X \rangle = b_i, \quad X \succeq 0 \right\}, \tag{34}$$

where $\mathbb{S}^n(\mathbb{C})$ denotes the space of $n \times n$ Hermitian matrices. Since the objective function is precisely the trace, any bound on the optimal value can be taken as the trace bound. We use the squared norm of the vector x as the trace bound for our experiments. Even though x is unknown, bounds on its norm are known [78].

The SDP relaxation of the phase retrieval problem is hard because the linear and adjoint operators require the utilization of a fast Fourier transform (FFT) subroutine. On the GPU, this subroutine can be handled through CUDA's cuFFT package [51]. Moreover, the SDP instances are extremely dense, and thus storing the data is expensive. For these reasons, the cuHALLaR binary and cuLoRADS were not tested for this problem class. We only tested HALLaR and cuHALLaR using on-the-fly problem generation.

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; 3,704,736	600.62/2	29.016/6	**20.70**
316,228; 3,704,736	699.193/2	30.964/6	**22.58**
316,228; 3,704,736	507.60/2	33.738/7	**15.05**
316,228; 3,704,736	620.56/2	34.124/8	**18.19**
3,162,278; 37,947,336	*/4.1e-03	583.782/14	> 74.0

Table 7: Runtimes (in seconds) and final rank for the Phase Retrieval problem. A relative tolerance of $\epsilon = 10^{-5}$ is used, with a 12-hour time limit is set for HALLaR. An entry marked with $*/N$ means the corresponding method finds an approximate solution with relative accuracy strictly larger than the desired accuracy in which case N expresses the maximum of the three relative accuracies in (31). The "Ratio" column reports the ratio of HALLaR runtime to cuHALLaR runtime.

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.

6 Concluding Remarks

In this paper, we address the fundamental challenge of developing scalable and efficient solvers for large-scale semidefinite programming problems. We introduce cuHALLaR, a GPU-accelerated implementation of the hybrid low-rank augmented Lagrangian method originally proposed in [49]. 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.

Through extensive numerical experiments across three problem domains, we demonstrate that cuHALLaR achieves remarkable performance improvements over both CPU-based implementations and competing GPU-accelerated

solvers. For matrix completion problems, cuHALLaR consistently outperforms the state-of-the-art GPU solver cuLoRADS by factors of 2x to 25x, while finding solutions with significantly lower ranks. Most impressively, cuHALLaR successfully solves instances with approximately 260 million constraints in under 1 minute, while the CPU implementation, HALLaR, fails to converge within 12 hours.

For maximum stable set problems, our results reveal that cuHALLaR's performance advantage scales dramatically with problem size and graph density. We achieve speedups of 50x for the largest Hamming graph problems and 135x for million-node AG-Monien instances. Similarly, for phase retrieval problems, cuHALLaR achieves speedups of 15x to 47x on large instances, and successfully solves the largest test case—with a matrix variable of size over 3 million—in under 10 minutes. In contrast, the CPU-based HALLaR times out after 12 hours on this same instance, rendering it effectively intractable.

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 variable and inequality constraints. These developments would further advance the approaches introduced in [49] and establish GPU-accelerated low-rank methods as the standard for solving large-scale SDPs in practical applications.

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 [49, 62]. HALLaR uses ADAP-AIPP in its step 5 to find an approximate stationary point (according to the criterion in (7)) of a nonconvex problem of the form

$$\min_{U} \quad \left\{ g(U) = \mathcal{L}_\beta(UU^\top; p) \quad : \quad U \in B_\tau^s \right\} \tag{35}$$

where $\mathcal{L}_\beta(X; p)$ is as in (6) and B_τ^s is as in (3).

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

$$\overline{R} \in \nabla g(\overline{W}) + N_{B_\tau^s}(\overline{W}), \quad \|\overline{R}\|_F \le \rho, \quad g(\overline{W}) \le g(\underline{W}). \tag{36}$$

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 (35), an initial point $\underline{W} \in B_\tau^s$, an initial prox stepsize $\lambda_0 > 0$, and a tolerance $\rho > 0$.

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

$$\lambda = \lambda_0, \quad \bar{M}_0 = 1; \tag{37}$$

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

$$x_0 = W_{j-1}, \quad (\mu, L_0) = (1/2, \underline{\mathrm{M}}_j), \tag{38}$$

$$\psi_s = \lambda g + \frac{1}{2}\|\cdot - W_{j-1}\|_F^2, \quad \psi_n = \lambda \delta_{B_\tau^s}; \tag{39}$$

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

$$\lambda g(W_{j-1}) - \left[\lambda g(W) + \frac{1}{2}\|W - W_{j-1}\|_F^2\right] \geq V \bullet (W_{j-1} - W), \tag{40}$$

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

$$R_j := \frac{V_j + W_{j-1} - W_j}{\lambda_j} \tag{41}$$

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 [49] that ADAP-AIPP is able to find a triple that satisfies (36) 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 [49].

A.1 ADAP-FISTA

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

$$\min_{U \in B_\tau^s} \psi_s(U) := \lambda g(U) + 0.5\|U - W\|_F^2 \tag{42}$$

where g is as in (35). 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 (42), an initial point $x_0 \in B_\tau^s$ and scalars $\mu > 0$ and $L_0 > \mu$.

0. set $y_0 = x_0$, $A_0 = 0$, $\tau_0 = 1$, and $i = 0$;

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

2. compute

$$a_i = \frac{\tau_i + \sqrt{\tau_i^2 + 4\,\tau_i A_i\,(L_{i+1} - \mu)}}{2\,(L_{i+1} - \mu)}, \quad \tilde{x}_i = \frac{A_i\,y_i + a_i\,x_i}{A_i + a_i}, \tag{43}$$

$$y_{i+1} = \arg\min_{u \in B_\tau^s} \left\{\ell_{\psi_s}(u; \tilde{x}_i) + \tfrac{L_{i+1}}{2}\|u - \tilde{x}_i\|^2\right\}; \tag{44}$$

if

$$\ell_{\psi_s}(y_{i+1}; \tilde{x}_i) + \tfrac{(1-\chi)\,L_{i+1}}{4}\,\|y_{i+1} - \tilde{x}_i\|^2 \;\geq\; \psi_s(y_{i+1}), \tag{45}$$

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

3. update

$$A_{i+1} = A_i + a_i, \qquad \tau_{i+1} = \tau_i + a_i\,\mu, \tag{46}$$

$$s_{i+1} = (L_{i+1} - \mu)\,(\tilde{x}_i - y_{i+1}), \tag{47}$$

$$x_{i+1} = \frac{1}{\tau_{i+1}}\left[\mu\,a_i\,y_{i+1} + \tau_i\,x_i - a_i\,s_{i+1}\right]; \tag{48}$$

4. if

$$\|y_{i+1} - x_0\|^2 \;\geq\; \chi\, A_{i+1}\, L_{i+1}\, \|y_{i+1} - \tilde{x}_i\|^2, \tag{49}$$

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

5. If the inequality

$$\|s_{i+1}\| \;\leq\; \sigma\, \|y_{i+1} - x_0\| \tag{50}$$

holds, then stop with **success** and output $(y, L) = (y_{i+1}, L_{i+1})$; 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 (42) 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 (42). If the objective in (42) is strongly convex, then ADAP-FISTA always succeeds in solving (42). For more technical details on the type of solution that ADAP-FISTA aims to find, see Appendix B in [49].

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] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, *Julia: A fresh approach to numerical computing*, SIAM review, 59 (2017), pp. 65–98.

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

[13] 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).

[14] J. Choquette, *Nvidia hopper h100 gpu: Scaling performance*, IEEE Micro, 43 (2023), pp. 9–17.

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

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

[17] 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.

[18] 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).

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

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

[21] 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.

[22] 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.

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

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

[25] 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.

[26] 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).

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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).

[33] 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).

[34] 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.

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

[36] 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.

[37] 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).

[38] 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.

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

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

[41] 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.

[42] 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).

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

[44] 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).

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

[46] 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).

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

[48] 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.

[49] 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).

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

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

[52] 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.

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

[54] 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.

[55] 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.

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

[57] 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.

[58] 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).

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

[60] 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.

[61] 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.

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

[63] 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).

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

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

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

[67] 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.

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

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

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

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

[72] 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.

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

[74] 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.

[75] 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).

[76] 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.

[77] 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.

[78] 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.

[79] 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.

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

[81] 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.

[82] 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.

[83] 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.

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