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[**]

June 12, 2025

Abstract

This paper introduces cuHALLaR, a GPU-accelerated implementation of the HALLaR method proposed in [47] for solving large-scale semidefinite programming (SDP) problems. We demonstrate how our Julia-based implementation efficiently uses GPU parallelism through optimization of simple, but key, operations, including linear maps, adjoints, and gradient evaluations. Extensive numerical experiments across three problem classes—maximum stable set, matrix completion, and phase retrieval show significant performance improvements over both CPU implementations and existing GPU-based solvers. For the largest instances, cuHALLaR achieves speedups of 30-140x on matrix completion problems, up to 135x on maximum stable set problems for Hamming graphs with 8.4 million vertices, and 15-47x on phase retrieval problems with dimensions up to 3.2 million. Our approach efficiently handles massive problems with dimensions up to $(n, m) \approx (8 \times 10^6, 3 \times 10^8)$ at 10^{-5} relative precision, solving matrix completion instances with over 8 million rows and columns in just 142 seconds. These results establish cuHALLaR as a very promising GPU-based method for solving large-scale semidefinite programs.

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)

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

and

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

where $b \in \mathbb{R}^m$, $C \in \mathbb{S}^n$, $\mathcal{A} : \mathbb{S}^n \to \mathbb{R}^m$ is a linear map, $\mathcal{A}^* : \mathbb{R}^m \to \mathbb{S}^n$ is its adjoint, and Δ^n is the spectraplex

$$\Delta^n := \{X \in \mathbb{S}^n : \operatorname{Tr}(X) \leq 1, X \succeq 0\}. \tag{1}$$

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

[*]**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. (Email: `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. vhn@fgv.br

[**]Department of Combinatorics and Optimization, University of Waterloo, Waterloo, ON, N2L 3G1. a3sujana@uwaterloo.ca

Authors are listed in alphabetical order.

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 [2, 5, 31, 46, 65]. 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 [1, 16, 17, 19, 20, 23, 25, 26, 33, 34, 38, 51, 54, 56, 58, 71, 74, 76, 78, 81, 82].

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 [12, 30, 39, 41–43, 57]. In this paper, we introduce an enhanced Julia-based GPU implementation, referred to as cuHALLaR, of the first-order HALLaR method developed in [47] 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, 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 [41–43] 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 [57] and CuClarabel [12] similarly exploit GPU parallelism for significant speedups. Most recently, [39] proposed a new GPU conic programming solver that utilizes the primal-dual hybrid gradient (PDHG) method for solving several classes of cone optimization problems, including semidefinite programs.

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 [6, 21, 24, 27, 32, 53, 61, 66, 79, 80, 83]. 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 [9, 10]. 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 [9, 10] converts SDP (P) into a nonconvex quadratic program with quadratic constraints (QCQP) through the transformation $X = UU^T$, 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) [4, 7, 8, 14, 15, 55, 67]. For a more extensive list of papers which characterize the optimization landscape of nonconvex formulations obtained through Burer-Monteiro factorizations, see [18, 22, 37, 40, 44, 45, 49, 50]. 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 [23, 29, 30, 47, 62–64, 68–70, 76–78].

We now give a brief comparison between HALLaR, which was first developed in [47] and the LoRADS method developed in [29]. HALLaR utilizes an inexact AL method to solve (P). It solves each of its AL subproblems over the spectraplex $\arg\min_{X\in\Delta^n} \quad C \bullet X + p^\top(\mathcal{A}(X)-b)+\frac{\beta}{2}\|\mathcal{A}X-b\|^2$ by first restricting them to the space of matrices of rank at most r through the transformation $X = UU^T$ 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 also potentially increases the rank of the current iterate.

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 $X = UU^T$ change of variables used by HALLaR, LoRADS reformulates them by replacing X by UV^T 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, the cuHALLaR method proposed in this paper is a GPU accelerated version of HALLaR. It is displayed on several important classes of SDPs that cuHALLaR is 2 to 25 times faster than cuLoRads on many large instances. 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 where the size of the matrix variable is 8 million and the number of constraints is approximately 300 million 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 $N^\epsilon_C(z)$, is

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

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

$$B^s_r := \{U \in \mathbb{R}^{n\times s} : \|U\|_F \leq r\}. \tag{2}$$

2 Overview of HALLaR

This section provides a minimal review of HALLaR, the hybrid low-rank augmented Lagrangian method that was first developed in [47].

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

$$\|\mathcal{A}\bar{X} - b\| \leq \epsilon_p, \quad |C \bullet \bar{X} - (b^T\bar{p} - \bar{\theta})| \leq \epsilon_{pd}, \quad C + \mathcal{A}^*p + \bar{\theta}I \succeq 0, \quad \bar{\theta} \geq 0. \tag{3}$$

Being an inexact AL method, HALLaR generates sequences $\{X_t\}$ and $\{p_t\}$ according to the following updates

$$\begin{aligned} X_t &\approx \arg\min_X \ \{\mathcal{L}_{\beta_t}(X; p_{t-1}) \ : \ X \in \Delta^n\}, && \text{(4a)} \\ p_t &= p_{t-1} + \beta_t(\mathcal{A}(X_t) - b) && \text{(4b)} \end{aligned}$$

where

$$\mathcal{L}_\beta(X;p) \quad := \quad C \bullet X + p^\top(\mathcal{A}(X)-b) + \frac{\beta}{2}\|\mathcal{A}X-b\|^2 \tag{5}$$

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 (4a). HLR never forms X_t but outputs a low-rank factor $U_t \in B_1^{s_t}$ such that $X_t = U_t U_t^T$ where $B_1^{s_t}$ is as in (2) and hopefully $s_t \ll n$. The next paragraph briefly describes HLR in more detail for solving (4a).

Given a pair $(s,\tilde{Y}) \in \mathbb{Z}_+ \times B_1^s$, 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 (4a) obtained through the change of variable $X = UU^T$, namely:

$$\min_U \quad \left\{ g_t(U) = \mathcal{L}_{\beta_t}(UU^\top; p_{t-1}) \quad : \quad U \in B_1^s \right\}; \tag{L$_r$}$$

(ii) checks whether $X = YY^\top$, is a reasonable approximate solution of (4a); if so it stops and sets $(s_t, U_t) = (s, Y)$; and; iii) else, it performs a Frank-Wolfe step in the X-space (but implemented in the U-space) to obtain a new 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

$$R \in \nabla g_t(Y) + N_{B_1^s}(Y), \quad \|R\|_F \le \rho, \quad g_t(Y) \le g_t(\tilde{Y}). \tag{6}$$

It is easy to verify that (6), in terms of the SDP data, is equivalent to

$$R \in 2[C + \mathcal{A}^*(p_{t-1} + \beta_t(\mathcal{A}(YY^T) - b))]Y + N_{B_1^s}(Y), \quad \|R\|_F \le \rho.$$

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 [47, 60], and also in earlier works (see e.g. [11, 35, 36, 52]) 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. **Output**: $(\bar{X}, \bar{p}, \bar{\theta}) \in \Delta^n \times \mathbb{R}^m \times \mathbb{R}_+$, an $(\epsilon_{\rm p}, \epsilon_{\rm pd})$-solution of the pair of SDPs (P) and (D).

```
Require: Let initial points $(U_0, p_0) \in B_1^{s_0} \times \mathbb{R}^m$, and tolerance pair $(\epsilon_{\rm p}, \epsilon_{\rm pd}) \in \mathbb{R}^2_{++}$ be given, and let $\{\epsilon_t\}_{t\geq 1}$ and $\{\beta_t\}_{t\geq 1}$ be a decreasing and increasing sequence of positive integers, respectively.
1: set $t \leftarrow 1$;
2: set $\tilde{Y} = U_{t-1}$, $s = s_{t-1}$, $Y = 0_{n\times s}$, $G = 0_{n\times n}$, and $\theta = \infty$;
3: while $(GY) \bullet Y + \theta > \epsilon_t$ do                                   ▷ (HLR method)
4:     call a nonconvex solver with initial point $\tilde{Y}$, tolerance $\epsilon_t$, and function $\mathcal{L}_{\beta_t}(UU^\top; p_{t-1}) + \delta_{B_1^s}(U)$ to find a point $Y \in B_1^s$ that is an $\epsilon_t$-approximate stationary solution (according to the criterion in (6)) of
         $$\min_U \quad \left\{ \mathcal{L}_{\beta_t}(UU^\top; p_{t-1}) \quad : \quad \|U\|_F \leq 1, \quad U \in \mathbb{R}^{n\times s} \right\}; \tag{7}$$
5:     compute
         $$G := \nabla(\mathcal{L}_{\beta_t}(YY^T; p_{t-1})) \in \mathbb{S}_n \tag{8}$$
       and a minimum eigenpair of $G$, i.e., $(\lambda_{\min}(G), v_{\min}(G)) \in \mathbb{R} \times \mathbb{R}^n$, and set
         $$\theta := \max\{-\lambda_{\min}(G), 0\}, \qquad y := \begin{cases} v_{\min}(G) & \text{if } \theta > 0 \\ 0 & \text{otherwise;} \end{cases} \tag{9}$$
6:     if $(GY) \bullet Y + \theta \leq \epsilon_t$ then
7:         break while loop and go to step 11 below;
8:     end if
9:     compute
         $$\alpha = \min\left\{ \frac{Y \bullet [C + \mathcal{A}^*\left(p_{t-1} + \beta_t(\mathcal{A}(YY^\top) - b)\right)]Y + \theta}{\beta_t \|\mathcal{A}(YY^\top) - \mathcal{A}(yy^\top)\|^2}, \ 1 \right\} \tag{10}$$
       and set
         $$(\tilde{Y}, s) = \begin{cases} (y, 1) & \text{if } \alpha = 1 \\ \left(\left[\sqrt{1-\alpha}\, Y, \sqrt{\alpha}\, y\right], s+1\right) & \text{otherwise;} \end{cases} \tag{11}$$
10: end while
11: set $(U_t, \theta_t, s_t) = (Y, \theta, s)$;
12: set
         $$p_t = p_{t-1} + \beta(\mathcal{A}(U_t U_t^\top) - b); \tag{12}$$
13: if
         $$\|\mathcal{A}(U_t U_t^\top) - b\| \leq \epsilon_{\rm p}, \quad |CU_t \bullet U_t - (-b^\top p_t - \theta_t)| \leq \epsilon_{pd} \tag{13}$$
    then return $(\bar{U}, \bar{p}, \bar{\theta}) = (U_t, p_t, \theta_t)$;
14: set $t = t + 1$ and go to step 2.
```

Several remarks about each of the steps of HALLaR are now given. First, t is the iteration count for HALLaR. Second, each iteration of HALLaR performs a HLR call and each call is performed in the while loop from steps 3 to 10. In line 2, the triple (Y, G, θ) is set as $(0_{n\times s}, 0_{n\times n}, \infty)$, which means the HLR call in the while loop is always performed during each iteration of HALLaR. Third, after a HLR call is finished, HALLaR updates the Lagrange multiplier in line 12 and checks for termination in line 13. Lastly, the termination condition on this line does not check for near dual feasibility since it can be shown that for every $t \geq 1$,

$$S_t := C + \mathcal{A}^* p_t + \theta_t I \succeq 0, \quad \theta_t \geq 0, \tag{14}$$

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

We now comment on the steps performed within the while loop (lines 3 to 10) in light of the three steps of the HALLaR outline given just before its formal description. Step 4 is implementing step i) of the HALLaR outline. Steps 5 and 6 are implementing step ii). Finally, step 9 implements step iii).

We now give an interpretation of step 9 of HALLaR relative to the X-method. The stepsize α computed in (10) corresponds to solving

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

Moreover, the iterate $\tilde{Y}$ computed in equation (11) has the property that $\tilde{Y}\tilde{Y}^\top$ is equal to the usual FW iterate $\alpha yy^T + (1-\alpha)YY^T$ in the X-space; hence $\tilde{Y}$ is an U-factor for the Frank-Wolfe iterate in the X-space.

3 Accelerating Hallar

In this section, we discuss how to translate the initial CPU implementation of HALLaR efficiently to a scalable and well-optimized GPU implementation.

3.1 GPU architecture, threads, and minimizing CPU/GPU communication

GPUs execute threads in a Single Instruction, Multiple Data (SIMD) fashion: groups of 32 threads, called warps, run the same instruction in lockstep. These warps are organized into blocks, each mapped onto a streaming multiprocessor (SM). Within a block, threads can coordinate via barrier synchronizations (e.g. "__syncthreads") and share data through fast on-chip shared memory. Kernels (functions designed to run in parallel in the GPU) launch a grid of blocks to cover the entire data set, so that thousands of warps can run concurrently, hiding memory latency and maximizing arithmetic throughput.

The GPU memory hierarchy further shapes performance. Global memory, while large, has high latency and must be accessed coalesced to achieve high bandwidth. Shared memory resides on each SM and offers very low latency when threads within a block reuse it. Finally, each thread has its own registers for the fastest access; spilling to local memory incurs significant cost. Effective GPU kernels minimize global loads/stores, exploit shared memory for data reuse, and balance register usage to avoid spilling.

Figure 1, adapted from the NVIDIA documentation [48], displays the NVIDIA Hopper architecture used in modern GPUs like the H200. The SMs are the main units of execution inside the GPU. They are grouped into the TPC (Texture Processing Clusters) which are in turn grouped into GPC (GPU Processing Clusters). This hierarchy of clusters allows the GPU to improve the management and distribution of the workload between the SMs, improving scalability. The number of available GPCs and TPCs depends on the GPU model. Inside each SM, the warp scheduler, dispatch unit and the L0 instruction cache work as a control unit, delivering instructions to the warps. Next, we see the set of registers each SM has access to, followed by execution components: CUDA cores, tensor cores, LD/ST and SFU. The CUDA cores execute scalar aritmetic operations, while the tensor cores are specialized to execute multiplication and addition $C = D * A + C$ in batches of data. The LD/ST units performs load and store operations on the data. The SFU (Special Function Units) are specialized for transcendental and trigonometric operations. The TMA (Tensor Memory Acceleration) provides asynchronous transfers of large blocks of data between the global and shared memory. Next, we see the L1-cache that optimizes transfers between the shared memory and the CUDA/tensor cores. Back to the GPU, we have the L2-cache that optimizes transfers from the global GPU memory to all SMs. Access to L2-cache by the SMs is controlled by the GPC.

Before launching a kernel, we must transfer the input dataset from the CPU (also known as host in GPU programming) to the GPU (also called device) and define the number of blocks and threads used. The memory allocated in the CPU is not accessible by the GPU and vice-versa. When the kernel is launched, the workload is distributed among the GPCs that, in turn, distribute the workload among their TPCs. The TPCs forward the thread blocks among its SMs. The SMs divide the thread blocks into warps and prepare the shared memory and registers. The warp scheduler and the dispatch unit select the warps ready to run and dispatch the instructions to execution by the CUDA/tensor cores, the LD/ST or the SFU. After the kernel is finished, a transfer from the device back to the host may be necessary.

The GPU programming paradigm is meant to integrate CPU and GPU to obtain the performance gains of both architectures. Sections of the code that allow high parallelization, like matrix, vector and tensor operations are implemented as kernels, while serial tasks are implemented as regular functions in the CPU. This CPU/GPU integration sometimes requires data transfers between the CPU and GPU. Modern GPUs compute faster than they can feed data from memory, so these transfers are often the bottleneck of applications. Therefore we must minimize data transfers to achieve maximum performance.

cuHALLaR stores on the GPU, at the beginning, key data such as the primal/dual solutions, the right-hand side vector b, and the data used to compute C and $\mathcal{A}$. Any other auxiliary data is allocated directly on the GPU. So all operations that involve arrays with huge size are done in the GPU, including the dual feasibility computations, which was computed in the CPU by other GPU solvers like [30].

Another important performance aspect is that, just like in the CPU, launching threads in the GPU incurs an overhead due to the grid, context and communication management. Therefore it is also important to minimize the number of kernels launched, merging trivial kernels and avoiding to launch them inside loops when possible.

In HALLaR, three operations are critical to performance: the linear map $\mathcal{A}$, the adjoint $\mathcal{A}^*$ and the gradient ∇g. These operations are performed at every ADAP-FISTA iteration, and the gradient is also computed at the Frank-Wolfe step of HLR. For the maximum stable set and matrix completion problems, CuHallar solves $\mathcal{A}$ and $\mathcal{A}^*$ in a single kernel, and the gradient is solved with two kernels. In the phase retrieval problem, due to the computation

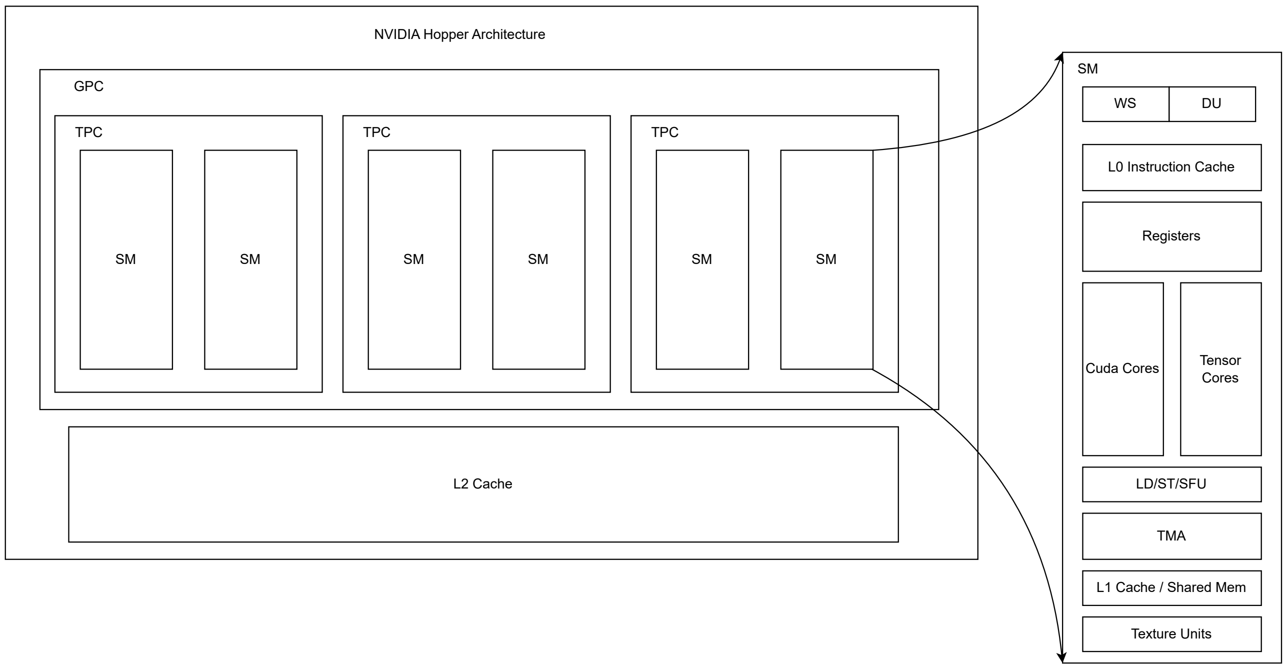

Figure 1: GPU architecture, adapted from the NVIDIA documentation [48], displaying the general GPU design and the components of each SM.

of the discrete Fourier transform, all operations are solved with r kernel calls, where r is the rank of the current solution.

In cuHALLaR, we use the CUDA.jl library, the Julia interface for the CUDA runtime library. CUDA.jl provides high level implementations of many cuBLAS, cuSPARSE and cuFFT functions. The transfer of data between the CPU and GPU is as simple as converting a variable between the Julia Array type and the CUDA.jl CuArray type. CUDA.jl also provides macros so users can implement their own kernels. With the help of Julia broadcast fusion feature, CUDA.jl is able to merge kernels in broadcasted expressions. This allows cuHallar to reduce the number of kernels invoked, specially in the ADAP-FISTA method.

4 Numerical Experiments

In this section, we provide provide extensive experiments comparing cuHALLaR, HALLaR, and cuLoRADS. The experiments concern SDP instances for the matrix completion, the maximum stable set, and the phase retrieval problems. These problems were also used in the benchmark of the original HALLaR paper [47].

4.1 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 [3] module. Our implementation uses the CUDA language [13] for interfacing with NVIDIA CUDA GPUs. The experiments are performed running 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 $\epsilon > 0$, all methods stop when an iterate $(X,p) \in \Delta^n \times \mathbb{R}^m$ satisfying

$$\max\left\{\frac{\|\mathcal{A}X - b\|_2}{1+\|b\|_1}, \frac{|\text{pval} - \text{dval}|}{1+|\text{pval}|+|\text{dval}|}, \frac{\max\{0, -\lambda_{\min}(C+\mathcal{A}^*p)\}}{1+\|C\|_1}\right\} \le \epsilon. \tag{15}$$

is obtained, where $\text{pval} = C \bullet X$ and $\text{dval} = -b^\top p$ denote the primal and dual values of (X,p), respectively.

Scaling. We have assumed in Section 1 that the trace of X is bounded by one. However, cuHALLaR and HALLaR allow $\operatorname{tr} X \le 1$ to be replaced by the more general constraint $\operatorname{tr}(X) \le \tau$ for some $\tau > 0$. This is done by reformulating the SDP as in (P) using the changes of variables $X \leftarrow X/\tau$ and changing the rhs as $b \leftarrow b/\tau$. (15) is used in the experiments of phase retrieval and matrix completion.

Initialization. HALLaR and cuHALLaR are initialized using an initial random $n \times 1$ matrix with entries generated independently from the Gaussian distribution over $\mathbb{R}$ (or over $\mathbb{C}$ for complex-valued SDPs) which is then scaled to be a unit vector. The dual initial point p_0 is taken to be the zero vector.

Input format. HALLaR and cuHALLaR require the following user inputs to describe the SDP instance:

(i) The vector $b \in \mathbb{R}^m$ and the scalar $\tau > 0$. Here, τ represents the upper bound for the trace of the primal matrix variable $X \in \mathbb{S}^n_+$, such that $\operatorname{Tr}(X) \le \tau$.

(ii) A routine for evaluating CU for arbitrary matrices U of compatible dimensions.

(iii) A routine for evaluating $(\mathcal{A}^*p)U$ for arbitrary vectors p and matrices U of compatible dimensions.

(iv) A routine for evaluating $\mathcal{A}(UU^\top)$ for arbitrary matrices U of compatible dimensions.

We are aware that the above operator-based input format for SDPs is not prevalent as matrix-based representations such as the SDPA format [72, 73] or the SEDUMI format [59]. Matrix-based representations have been widely adopted by numerous SDP solvers, and are quite effective for SDPs involving sparse matrices. However, these representations might not be effective for SDPs involving dense matrices. In particular, the stable set and the phase retrieval SDPs involve fully dense matrices, and any code that requires input in a matrix-based format (e.g., SDPA format) can take a long time to read and manipulate their data. In contrast, our operator-based input format enables us to efficiently handle extremely large stable set and phase retrieval SDP instances.

Ratio: In all tables below, the entry labeled “Ratio” reports the running time of the CPU implementation divided by the running time of the GPU implementation (or, when comparing two solvers, the time of the first solver over the time of the second). For instance, in Table 2 the first row shows cuHALLaR runtime of 0.77 sec and a cuLoRADS runtime of 123.71 sec, yielding a ratio of 160.662. This means that cuLoRADS took 160.662 times longer to solve that instance than cuHALLaR.

4.2 Experiments for matrix completion

Given integers $n_2 \ge n_1 \ge 1$, 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:

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

The above problem can be rephrased as the following SDP:

$$\min_{X \in \mathbb{S}^{n_1+n_2}} \left\{ \frac{1}{2}\operatorname{tr}(X) : X = \begin{pmatrix} W_1 & Y \\ Y^\top & W_2 \end{pmatrix} \succeq 0\,, \quad Y_{i,j} = M_{i,j}\,,\ \forall\,(i,j) \in \Omega). \right\} \tag{16}$$

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.

Problem Instance		**Runtime (seconds) / rank**		
Size $(n_1, n_2; m)$	r	cuHALLaR	cuLoRADS	Ratio
3,000, 7,000; 828,931	3	0.77/3	123.71/19	160.662
3,000, 7,000; 828,931	3	0.7/3	11.42/19	16.314
3,000, 7,000; 2,302,586	5	1.28/5	3.19/19	2.492
3,000, 7,000; 2,302,586	5	1.09/5	2.26/19	2.073
10,000, 21,623; 2,948,996	3	1.22/3	5.14/21	4.213
10,000, 21,623; 8,191,654	5	1.95/5	4.71/21	2.415
25,000, 50,000; 3,367,574	2	1.2/2	10.77/23	8.975
25,000, 50,000; 7,577,040	3	1.82/3	21.31/23	11.708
30,000, 70,000; 4,605,171	2	1.63/2	17.02/24	10.441
30,000, 70,000; 10,361,633	3	2.57/3	18.45/24	7.178
50,000, 100,000; 7,151,035	2	1.89/2	23.67/24	12.523
50,000, 100,000; 16,089,828	3	3.68/3	27.95/24	7.595
80,000, 120,000; 9,764,859	2	2.45/2	20.66/25	8.432
80,000, 120,000; 21,970,931	3	5.02/3	32.28/25	6.430
80,000, 120,000; 7,996,791	3	2.80/3	17.94/25	6.407
120,000, 180,000; 11,996,886	3	3.74/3	36.47/25	9.751
160,000, 240,000; 15,996,836	3	4.90/3	52.61/25	10.736
240,000, 360,000; 23,996,846	3	7.03/3	86.80/25	12.347

Table 2: cuHALLaR vs cuLoRADS runtimes for the matrix completion problem. Seed was 0 for all instances. Tolerance is set at 10^{-5}.

Table 2 provides a direct comparison between cuHALLaR and cuLoRADS on matrix completion problems, revealing cuHALLaR's substantial performance advantages. Our GPU implementation consistently outperforms cuLoRADS across all problem instances with speedups ranging from 2x to over 160x. Most notably, cuHALLaR solves these problems at a significantly lower rank, demonstrating its superior ability to find optimal low-rank solutions efficiently. This comparison highlights an advantage of our approach: cuLoRADS struggles with larger problems due to its reliance on the SDPA format and less efficient memory handling, while cuHALLaR's operator-based formulation enables it to tackle much larger instances without encountering the same memory limitations.

Table 3 highlights the speedup achieved by cuHALLaR over its CPU counterpart. The GPU implementation consistently delivers 10-140x faster solution times across standard problem sizes while maintaining identical solution quality. Below the midrule, we showcase truly large-scale instances with dimensions from $(n_1, n_2) = (320{,}000, 480{,}000)$ up to $(3{,}200{,}000, 4{,}800{,}000)$. For these instances, cuHALLaR achieves the most remarkable speedups of 104-143x. Most significantly, for problems with dimensions $(n_1, n_2) = (2{,}400{,}000, 3{,}600{,}000)$ and higher, the CPU implementation times out after 12 hours while cuHALLaR solves them in under 2.5 minutes. These "extra-large" instances couldn't be evaluated on cuLoRADS due to its reliance on the SDPA format, which becomes prohibitively memory-intensive for dense problems of this scale. This demonstrates not only the efficiency of cuHALLaR, but also its ability to handle problem sizes previously deemed computationally intractable and would have taken hours on the original CPU HALLaR implementation.

Note that the experimental results in Table 3 are divided into two distinct sets of problem instances. The first set showcases standard problem dimensions where both CPU and GPU implementations can successfully operate, allowing for direct performance comparison. Below the midrule, we present a set of significantly larger-scale problems with dimensions ranging from $(n_1, n_2) = (320{,}000, 480{,}000)$ up to $(3{,}200{,}000, 4{,}800{,}000)$.

Problem Instance		**Runtime (seconds) / rank**		
Size $(n_1, n_2; m)$	r	HALLaR	cuHALLaR	Ratio
3,000, 7,000; 828,931	3	7.63/3	0.77/3	9.91
3,000, 7,000; 828,931	3	7.52/3	0.70/3	10.74
3,000, 7,000; 2,302,586	5	34.52/5	1.28/5	26.97
3,000, 7,000; 2,302,586	5	34.42/5	1.09/5	31.58
10,000, 21,623; 2,948,996	3	38.88/3	1.22/3	31.87
10,000, 21,623; 2,948,996	3	34.80/3	1.11/3	31.35
10,000, 21,623; 8,191,654	5	141.17/5	2.05/5	68.86
10,000, 21,623; 8,191,654	5	140.55/5	1.95/5	72.08
25,000, 50,000; 3,367,574	2	43.52/2	1.20/2	36.27
25,000, 50,000; 7,577,040	3	79.93/3	1.82/3	43.92
30,000, 70,000; 4,605,171	2	60.57/2	1.63/2	37.16
30,000, 70,000; 10,361,633	3	145.76/3	2.57/3	56.72
50,000, 100,000; 7,151,035	2	121.63/2	1.89/2	64.35
50,000, 100,000; 16,089,828	3	242.00/3	3.68/3	65.76
80,000, 120,000; 9,764,859	2	237.38/2	2.45/2	96.89
80,000, 120,000; 21,970,931	3	490.07/3	5.02/3	97.62
80,000, 120,000; 7,996,791	3	217.97/3	2.80/3	77.85
120,000, 180,000; 11,996,886	3	379.99/3	3.74/3	101.60
160,000, 240,000; 15,996,836	3	430.87/3	4.90/3	87.93
240,000, 360,000; 23,996,846	3	834.17/3	7.03/3	118.66
320,000, 480,000; 31,996,932	3	1,152.68/3	9.60/3	120.07
400,000, 600,000; 39,996,783	3	1,587.08/3	13.05/3	121.62
480,000, 720,000; 47,996,814	3	2,299.54/4	16.06/3	143.18
560,000, 840,000; 55,996,812	3	2,126.11/4	18.23/3	116.63
640,000, 960,000; 63,996,921	3	2,659.69/3	20.53/3	129.55
720,000, 1,080,000; 71,996,873	3	2,904.02/3	26.90/3	107.96
800,000, 1,200,000; 79,996,838	3	2,833.93/3	27.05/3	104.77
1,600,000, 2,400,000; 159,996,869	3	8,364.62/3	60.68/3	137.85
2,400,000, 3,600,000; 239,996,782	3	Timed Out	100.17/3	N/A
3,200,000, 4,800,000; 319,996,871	3	Timed Out	142.27/3	N/A

Table 3: Performance comparison between HALLaR (CPU) and cuHALLaR for matrix completion problems. A relative tolerance of $\epsilon = 10^{-5}$ is used throughout. Ratio is CPU Time / GPU Time. Time limit was set at 12 hours.

4.3 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 a constant, the ϑ-function, which upper bounds the value of the maximum stable set. The ϑ-function is the value of the SDP

$$\max \quad \{ee^\top \bullet X \ : \ X_{ij} = 0,\ ij \in E,\ \mathrm{Tr}(X) = 1,\ X \succeq 0, \quad X \in \mathbb{S}^n\} \tag{17}$$

where $e = (1, 1, \ldots, 1) \in \mathbb{R}^n$ is the all ones vector. It was shown in [28] that the ϑ-function agrees exactly with the stable set 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	15.51
16,384; 49,122	delaunay_n14	DIMACS10	643.90/36	102.43/34	6.29
16,386; 49,152	fe-sphere	DIMACS10	21.74/3	17.82/3	1.22
22,499; 43,858	cs4	DIMACS10	749.95/15	588.46/115	1.27
25,016; 62,063	hi2010	DIMACS10	2,607.24/15	132.85/21	19.63
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	15.02
48,837; 117,275	nh2010	DIMACS10	7,694.06/20	2,186.74/172	3.52
24,300; 34,992	aug3d	GHS_indef	30.10/1	2.08/1	14.47
32,430; 54,397	ia-email-EU	Network Repo	598.76/5	52.86/4	11.33
11,806; 32,730	Oregon-2	SNAP	1,581.11/23	467.75/55	3.38
21,363; 91,286	ca-CondMat	SNAP	8,521.04/59	348.59/77	24.44
31,379; 65,910	as-caida_G_001	SNAP	2,125.41/8	428.41/27	4.96
26,518; 65,369	p2p-Gnutella24	SNAP	312.75/4	55.99/10	5.59
22,687; 54,705	p2p-Gnutella25	SNAP	242.42/4	88.53/13	2.74
36,682; 88,328	p2p-Gnutella30	SNAP	506.26/5	61.84/10	8.19
62,586; 147,892	p2p-Gnutella31	SNAP	1,484.71/5	245.33/29	6.05
49,152; 69,632	cca	AG-Monien	63.09/2	12.30/2	5.13
49,152; 73,728	ccc	AG-Monien	14.52/2	20.62/2	0.70
49,152; 98,304	bfly	AG-Monien	15.72/2	10.35/2	1.52
16,384; 32,765	debr_G_12	AG-Monien	250.40/10	189.57/10	1.32
32,768; 65,533	debr_G_13	AG-Monien	471.80/9	638.54/13	0.74
65,536; 131,069	debr_G_14	AG-Monien	474.22/9	29.51/9	16.07
131,072; 262,141	debr_G_15	AG-Monien	521.88/10	28.07/10	18.59
262,144; 524,285	debr_G_16	AG-Monien	1,333.31/12	84.28/13	15.82
524,288; 1,048,573	debr_G_17	AG-Monien	6,437.05/12	171.81/13	37.47
1,048,576; 2,097,149	debr_G_18	AG-Monien	16,176.30/13	119.50/13	135.37

Table 4: Runtimes (in seconds) for the Maximum stable set problem. A relative tolerance of $\epsilon = 10^{-5}$ is set.

The results in Table 4 demonstrate a strong scaling advantage for GPU implementation as problem size increases. For the largest instances, cuHALLaR achieves a remarkable 135x speedup over CPU implementation. Performance patterns vary across graph families: DIMACS10 instances show substantial speedups up to 19.6x, SNAP graphs deliver consistent 3-24x improvements, and AG-Monien graphs exhibit dramatic acceleration of 16-135x for larger instances. Some smaller or sparser graphs show modest or even negative speedups, highlighting that GPU acceleration benefits depend on sufficient computational complexity to amortize parallelization overhead. These findings reconfirm that GPU acceleration becomes increasingly advantageous as problem dimensions grow, enabling cuHALLaR to efficiently solve million-scale instances that would be computationally prohibitive on CPU alone.

Table 5 shows the significant performance advantage of GPU acceleration for larger-scale Maximum Stable Set problems on Hamming graphs. For small Hamming graphs, such as $H_{10,2}$ through $H_{13,2}$, HALLaR outperforms the cuHALLaR by factors of 0.088x to 0.596x, reflecting GPU parallelization overhead for modest problem sizes and the fact that HALLaR is not designed for small problems. A clear performance transition occurs at $H_{14,2}$, where cuHALLaR begins demonstrating superior performance with a 1.21x speedup. This advantage grows exponentially as problem size increases, reaching an impressive 50.1x speedup for $H_{23,2}$ with over 8 million vertices. All instances maintain a consistent optimal solution rank of 2, confirming the effectiveness of our low-rank approach. Notably, cuLoRADS encounters memory limitations (OOM) for problems larger than $H_{14,2}$, highlighting the superior memory efficiency of our implementation. For problems it could solve, cuLoRADS shows significantly longer runtimes and higher ranks than both our GPU and CPU implementations.

Problem Instance	**HALLaR**	**cuHALLaR**	**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	4.51
$H_{11,2}$(2,048; 11,264)	0.260/2	1.627/2	3.95/16	0.160	2.43
$H_{12,2}$(4,096; 24,576)	0.606/2	2.033/2	4.47/17	0.298	2.20
$H_{13,2}$(8,192; 53,248)	1.852/2	3.106/2	6.78/19	0.596	2.18
$H_{14,2}$(16,384; 114,688)	3.843/2	3.170/2	79.04/20	1.212	24.93
$H_{15,2}$(32,768; 245,760)	8.240/2	3.223/2	OOM*	2.557	N/A
$H_{16,2}$(65,536; 524,288)	17.235/2	4.457/2	OOM*	3.869	N/A
$H_{17,2}$(131,072; 1,114,112)	33.342/2	3.904/2	OOM*	8.541	N/A
$H_{18,2}$(262,144; 2,359,296)	46.436/2	2.627/2	OOM*	17.681	N/A
$H_{19,2}$(524,288; 4,980,736)	105.408/2	5.281/2	OOM*	19.958	N/A
$H_{20,2}$(1,048,576; 10,485,760)	249.701/2	8.109/2	OOM*	30.786	N/A
$H_{21,2}$(2,097,152; 22,020,096)	592.696/2	16.966/2	OOM*	34.934	N/A
$H_{22,2}$(4,194,304; 46,137,344)	1,702.255/2	37.145/2	OOM*	45.826	N/A
$H_{23,2}$(8,388,608; 96,468,992)	4,205.741/2	83.905/2	OOM*	50.125	N/A

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

The GSET benchmark [75] highlights how GPU performance shows a strong correlation with problem structure: for dense graphs with high-rank solutions such as G58, G59, and G64, GPU acceleration yields 4-5x speedups. However, for sparse graphs with very low-rank solutions such as G11, G12, and G32, CPU implementation performs 30-100x faster. This suggests that GPU acceleration is most beneficial when the computational workload involves more complex matrix operations that can effectively utilize parallel processing. Interestingly, the solution ranks sometimes differ between implementations, with GPU occasionally finding different rank solutions than CPU. The performance correlation appears more strongly tied to edge density than vertex count, with the highest speedups observed on instances having both moderate-to-large size and high edge count. These findings indicate that problem-specific characteristics, rather than simply problem size, determine when GPU acceleration provides the most significant performance advantages.

Problem Instance	HALLaR	cuHALLaR	Ratio
G1(800; 19,176)	98.25/98	45.19/98	2.17
G10(800; 19,176)	80.51/97	40.52/97	1.99
G11(800; 1,600)	0.14/2	4.23/2	0.03
G12(800; 1,600)	0.09/2	2.59/2	0.03
G14(800; 4,694)	45.37/73	214.73/74	0.21
G20(800; 4,672)	129.73/124	359.40/166	0.36
G43(1000; 9,990)	34.20/60	35.84/60	0.95
G51(1,000; 5,909)	167.03/141	273.84/128	0.61
G23(2,000; 19,990)	94.81/76	52.59/76	1.80
G31(2,000; 19,990)	122.07/77	58.23/76	2.10
G32(2,000; 4,000)	0.35/2	25.34/3	0.01
G34(2,000; 4,000)	2.00/2	25.29/3	0.08
G35(2,000; 11,778)	397.32/133	254.68/108	1.56
G41(2,000; 11,785)	343.84/143	286.82/139	1.20
G48(3,000; 6,000)	2.29/2	18.95/2	0.12
G55(5,000; 12,498)	152.03/45	81.11/45	1.87
G56(5,000; 12,498)	151.99/45	80.83/45	1.88
G57(5,000; 10,000)	8.71/2	42.55/2	0.20
G58(5,000; 29,570)	2,191.49/112	504.55/130	4.35
G59(5,000; 29,570)	2,167.27/111	539.28/121	4.02
G60(7,000; 17,148)	262.35/50	96.89/50	2.71
G62(7,000; 14,000)	7.80/3	25.18/3	0.31
G64(7,000; 41,459)	1,719.79/51	334.12/51	5.15
G66(9,000; 18,000)	11.31/3	27.91/3	0.41
G67(10,000; 20,000)	2.39/2	5.83/2	0.41
G72(10,000; 20,000)	2.41/2	5.81/2	0.41
G77(14,000; 28,000)	27.34/3	11.81/2	2.31
G81(20,000; 40,000)	53.85/3	64.20/3	0.84

Table 6: Runtimes (in seconds) for the Maximum Stable Set problem on GSET instances. Tolerances are set to 10^{-5}.

4.4 Experiments for phase retrieval

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

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

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

$$\min_X \quad \left\{ \operatorname{tr}(X) \quad : \quad \langle a_i a_i^H, X\rangle = b_i, \quad X \succeq 0, \quad X \in \mathbb{S}^n(\mathbb{C}) \right\},$$

where $\mathbb{S}^n(\mathbb{C})$ denots 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 [77].

In Table 7 we observe significant performance improvements with cuHALLaR compared to the CPU HALLaR as problem sizes increase. For medium-sized instances with $n = 31,623$, cuHALLaR achieves consistent speedups of 3.86-5.23x, completing calculations in approximately 29-31 seconds versus 119-153 seconds on CPU. For larger instances with $n = 100,000$, the speedup increases dramatically to 29.24-46.78x, with cuHALLaR solving problems in just 8.6-10.4 seconds compared to 304-434 seconds on CPU. The largest reported instances of $n = 316,228$ demonstrate impressive speedups of 15.05-22.58x, with GPU solution times around 29-34 seconds compared to 508-699 seconds on CPU. Most notably, cuHALLaR successfully solves the massive instance with $n = 3,162,278$ in under 10 minutes, while the CPU implementation times out after 12 hours. It's important to highlight that cuLoRADS was unable to handle these phase retrieval problems at all due to its reliance on the SDPA format, which becomes prohibitively expensive in both memory and computational requirements for problems of this scale. This further demonstrates the superior memory efficiency and scalability of our GPU implementation compared to both CPU-based methods and other GPU-accelerated SDP solvers.

Problem Instance	HALLaR	cuHALLaR	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	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	Timed Out	583.782/14	N/A

Table 7: Runtimes (in seconds) for the Phase Retrieval problem. A relative tolerance of $\epsilon = 10^{-5}$ is set and a time limit of 43200 seconds (12 hours) is given.

5 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 [47]. 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 more than 300 million constraints in under 3 minutes, 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 delivers speed improvements exceeding 74x, solving massive instances with over 3 million dimensions in under 10 minutes—problems that remain intractable for CPU-based approaches.

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. These developments would further advance the approaches introduced in [47] 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.

A ADAP-AIPP

This section presents the ADAP-AIPP method first developed in [47, 60]. HALLaR uses ADAP-AIPP in its step 4 to find an approximate stationary point (according to the criteria in (6)) of a nonconvex problem of the form

$$\min_{U} \quad \left\{ g(U) = \mathcal{L}_\beta(UU^\top; p) \quad : \quad U \in B_1^s \right\} \tag{18}$$

where $\mathcal{L}_\beta(X;p)$ is as in (5) and B_1^s is as in (2).

Given an initial point $\underline{W} \in B_1^s$ and a tolerance $\rho > 0$, the goal of ADAP-AIPP is to find a a triple $(\overline{W}; \overline{R}, \rho)$ that satisfies

$$\overline{R} \in \nabla g(\overline{W}) + N_{B_1^s}(\overline{W}), \quad \|\overline{R}\|_F \le \rho, \quad g(\overline{W}) \le g(\underline{W}). \tag{19}$$

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 (18), an initial point $\underline{W} \in B_1^s$, an initial prox stepsize $\lambda_0 > 0$, and a tolerance $\rho > 0$.

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

$$\lambda = \lambda_0, \quad \bar{M}_0 = 1; \tag{20}$$

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

$$x_0 = W_{j-1}, \quad (\mu, L_0) = (1/2, \underline{M}_j), \tag{21}$$

$$\psi_s = \lambda g + \frac{1}{2}\| \cdot - W_{j-1}\|_F^2, \quad \psi_n = \lambda \delta_{B_1^s}; \tag{22}$$

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

$$\lambda g(W_{j-1}) - \left[\lambda g(W) + \frac{1}{2}\|W - W_{j-1}\|_F^2 \right] \ge V \bullet (W_{j-1} - W), \tag{23}$$

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

$$R_j := \frac{V_j + W_{j-1} - W_j}{\lambda_j} \tag{24}$$

and go to step 3;

3. if $\|R_j\|_F \le \rho$, 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 [47] that ADAP-AIPP is able to find a triple that satisfies (19) 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 $\min_{U \in B_1^s} \lambda g(U) + 0.5\| \cdot - W_{j-1}\|_F^2$, 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 [47].

A.1 ADAP-FISTA

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

$$\min_{U\in B_1^s} \psi_s(U) := \lambda g(U) + 0.5\| \cdot - W\|_F^2 \tag{25}$$

where g is as in (18). 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 (25), an initial point $x_0 \in B_1^s$ 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

$$a_i = \frac{\tau_i + \sqrt{\tau_i^2 + 4\,\tau_i A_i\,(L_{i+1}-\mu)}}{2\,(L_{i+1}-\mu)}, \quad \tilde{x}_i = \frac{A_i\,y_i + a_i\,x_i}{A_i + a_i}, \tag{26}$$

$$y_{i+1} \;=\; \arg\min_{u\in B_1^s}\left\{\ell_{\psi_s}(u;\tilde{x}_i) + \tfrac{L_{i+1}}{2}\|u-\tilde{x}_i\|^2\right\}; \tag{27}$$

if

$$\ell_{\psi_s}(y_{i+1};\tilde{x}_i) + \tfrac{(1-\chi)\,L_{i+1}}{4}\,\|y_{i+1}-\tilde{x}_i\|^2 \;\geq\; \psi_s(y_{i+1}), \tag{28}$$

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

3. update

$$A_{i+1} = A_i + a_i, \qquad \tau_{i+1} = \tau_i + a_i\,\mu, \tag{29}$$

$$s_{i+1} = (L_{i+1}-\mu)\,(\tilde{x}_i - y_{i+1}), \tag{30}$$

$$x_{i+1} = \frac{1}{\tau_{i+1}}\left[\mu\,a_i\,y_{i+1} + \tau_i\,x_i - a_i\,s_{i+1}\right]; \tag{31}$$

4. if

$$\|y_{i+1}-x_0\|^2 \;\geq\; \chi\,A_{i+1}\,L_{i+1}\,\|y_{i+1}-\tilde{x}_i\|^2, \tag{32}$$

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

5. Compute

$$v_{i+1} = \nabla\psi_s(y_{i+1}) - \nabla\psi_s(\tilde{x}_i) + L_{i+1}(\tilde{x}_i - y_{i+1}). \tag{33}$$

If the inequality

$$\|v_{i+1}\| \leq \sigma\|y_{i+1}-x_0\| \tag{34}$$

holds then stop with **success** and output $(y, v, L) := (y_{i+1}, v_{i+1}, L_{i+1})$; otherwise, $i \leftarrow i+1$ and go to step 1.

Several remarks about ADAP-FISTA are now given. ADAP-FISTA is either able to successfully find a suitable solution of (25) 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 (25). If the objective in (25) is strongly convex, then ADAP-FISTA always succeeds in solving (25). For more technical details on the type of solution that ADAP-FISTA aims to find, see Appendix B in [47].

References

[1] Abdo Alfakih, L. Woosuk Jung, M. Walaa Moursi, and Henry 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.

[2] Farid Alizadeh, Jean-Pierre A. Haeberly, and Michael L. Overton. Primal-dual interior-point methods for semidefinite programming: Convergence rates, stability and numerical results. *SIAM Journal on Optimization*, 8(3):746–768, 1998.

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[4] Srinadh Bhojanapalli, Nicolas Boumal, Prateek Jain, and Praneeth Netrapalli. Smoothed analysis for low-rank solutions to semidefinite programs in quadratic penalty form. In *Conference On Learning Theory*, pages 3243–3270. PMLR, 2018.

[5] Brian Borchers. Csdp, ac library for semidefinite programming. *Optimization methods and Software*, 11(1-4):613–623, 1999.

[6] Jon Borwein and Henry Wolkowicz. Regularizing the abstract convex program. *Journal of Mathematical Analysis and Applications*, 83(2):495–530, 1981.

[7] Nicolas Boumal, Vlad Voroninski, and Afonso Bandeira. The non-convex burer-monteiro approach works on smooth semidefinite programs. *Advances in Neural Information Processing Systems*, 29, 2016.

[8] Nicolas Boumal, Vladislav Voroninski, and Afonso S Bandeira. Deterministic guarantees for burer-monteiro factorizations of smooth semidefinite programs. *Communications on Pure and Applied Mathematics*, 73(3):581–608, 2020.

[9] Samuel Burer and Renato DC Monteiro. A nonlinear programming algorithm for solving semidefinite programs via low-rank factorization. *Mathematical programming*, 95(2):329–357, 2003.

[10] Samuel Burer and Renato DC Monteiro. Local minima and convergence in low-rank semidefinite programming. *Mathematical programming*, 103(3):427–444, 2005.

[11] Y. Carmon, J. C. Duchi, O. Hinder, and A. Sidford. Accelerated methods for nonconvex optimization. *SIAM J. Optim.*, 28(2):1751–1772, 2018.

[12] Yuwen Chen, Danny Tse, Parth Nobel, Paul Goulart, and Stephen Boyd. Cuclarabel: Gpu acceleration for a conic optimization solver. *arXiv preprint arXiv:2412.19027*, 2024.

[13] Jack Choquette. Nvidia hopper h100 gpu: Scaling performance. *IEEE Micro*, 43(3):9–17, 2023.

[14] Diego Cifuentes. On the Burer–Monteiro method for general semidefinite programs. *Optimization Letters*, 15(6):2299–2309, 2021.

[15] Diego Cifuentes and Ankur Moitra. Polynomial time guarantees for the Burer-Monteiro method. *Advances in Neural Information Processing Systems*, 35:23923–23935, 2022.

[16] Qi Deng, Qing Feng, Wenzhi Gao, Dongdong Ge, Bo Jiang, Yuntian Jiang, Jingsong Liu, Tianhao Liu, Chenyu Xue, Yinyu Ye, et al. An enhanced admm-based interior point method for linear and conic optimization. *arXiv preprint arXiv:2209.01793*, 2022.

[17] Lijun Ding and Benjamin Grimmer. Revisiting spectral bundle methods: Primal-dual (sub)linear convergence rates. *SIAM Journal on Optimization*, 33(2):1305–1332, 2023.

[18] Lijun Ding and Stephen J Wright. On squared-variable formulations for nonlinear semidefinite programming. *arXiv preprint arXiv:2502.02099*, 2025.

[19] Lijun Ding, Alp Yurtsever, Volkan Cevher, Joel A Tropp, and Madeleine Udell. An optimal-storage approach to semidefinite programming using approximate complementarity. *SIAM Journal on Optimization*, 31(4):2695–2725, 2021.

[20] Ahmed Douik and Babak Hassibi. Low-rank riemannian optimization on positive semidefinite stochastic matrices with applications to graph clustering. In *International Conference on Machine Learning*, pages 1299–1308. PMLR, 2018.

[21] Dmitriy Drusvyatskiy, Henry Wolkowicz, et al. The many faces of degeneracy in conic optimization. *Foundations and Trends® in Optimization*, 3(2):77–170, 2017.

[22] Faniriana Rakoto Endor and Irène Waldspurger. Benign landscape for burer-monteiro factorizations of maxcut-type semidefinite programs. *arXiv preprint arXiv:2411.03103*, 2024.

[23] Murat A Erdogdu, Asuman Ozdaglar, Pablo A Parrilo, and Nuri Denizcan Vanli. Convergence rate of block-coordinate maximization burer–monteiro method for solving large sdps. *Mathematical Programming*, 195(1):243–281, 2022.

[24] Mituhiro Fukuda, Masakazu Kojima, Kazuo Murota, and Kazuhide Nakata. Exploiting sparsity in semidefinite programming via matrix completion i: General framework. *SIAM Journal on Optimization*, 11(3):647–674, 2001.

[25] Michael Garstka, Mark Cannon, and Paul Goulart. Cosmo: A conic operator splitting method for convex conic problems. *Journal of Optimization Theory and Applications*, 190(3):779–810, 2021.

[26] Naomi Graham, Hao Hu, Jiyoung Im, Xinxin Li, and Henry Wolkowicz. A restricted dual peaceman-rachford splitting method for a strengthened dnn relaxation for qap. *INFORMS Journal on Computing*, 34(4):2125–2143, 2022.

[27] Robert Grone, Charles R. Johnson, Eduardo M. Sá, and Henry Wolkowicz. Positive definite completions of partial hermitian matrices. *Linear Algebra and its Applications*, 58:109–124, 1984.

[28] Martin Grötschel, László Lovász, and Alexander Schrijver. Polynomial algorithms for perfect graphs. *In North-Holland mathematics studies*, 88:325–356, 1984.

[29] Qiushi Han, Chenxi Li, Zhenwei Lin, Caihua Chen, Qi Deng, Dongdong Ge, Huikang Liu, and Yinyu Ye. A low-rank admm splitting approach for semidefinite programming. *arXiv preprint arXiv:2403.09133*, 2024.

[30] Qiushi Han, Zhenwei Lin, Hanwen Liu, Caihua Chen, Qi Deng, Dongdong Ge, and Yinyu Ye. Accelerating low-rank factorization-based semidefinite programming algorithms on gpu. *arXiv preprint arXiv:2407.15049*, 2024.

[31] Christoph Helmberg, Franz Rendl, Robert J. Vanderbei, and Henry Wolkowicz. An interior-point method for semidefinite programming. *SIAM Journal on Optimization*, 6(2):342–361, 1996.

[32] Hao Hu, Renata Sotirov, and Henry Wolkowicz. Facial reduction for symmetry reduced semidefinite and doubly nonnegative programs. *Mathematical Programming*, 200(1):475–529, 2023.

[33] Wen Huang and Xiangxiong Zhang. Solving phaselift by low-rank riemannian optimization methods for complex semidefinite constraints. *SIAM Journal on Scientific Computing*, 39(5):B840–B859, 2017.

[34] Shucheng Kang, Xin Jiang, and Heng Yang. Local linear convergence of the alternating direction method of multipliers for semidefinite programming under strict complementarity. *arXiv preprint arXiv:2503.20142*, 2025.

[35] W. Kong, J.G. Melo, and R.D.C. Monteiro. Complexity of a quadratic penalty accelerated inexact proximal point method for solving linearly constrained nonconvex composite programs. *SIAM J. Optim.*, 29(4):2566–2593, 2019.

[36] W. Kong, J.G. Melo, and R.D.C. Monteiro. An efficient adaptive accelerated inexact proximal point method for solving linearly constrained nonconvex composite problems. *Comput. Optim. Appl.*, 76(2):305–346, 2019.

[37] Eitan Levin, Joe Kileel, and Nicolas Boumal. The effect of smooth parametrizations on nonconvex optimization landscapes. *Mathematical Programming*, 209(1):63–111, 2025.

[38] Xinxin Li, Ting Kei Pong, Hao Sun, and Henry Wolkowicz. A strictly contractive peaceman-rachford splitting method for the doubly nonnegative relaxation of the minimum cut problem. *Computational optimization and applications*, 78(3):853–891, 2021.

[39] Zhenwei Lin, Zikai Xiong, Dongdong Ge, and Yinyu Ye. Pdcs: A primal-dual large-scale conic programming solver with gpu enhancements. *arXiv preprint arXiv:2505.00311*, 2025.

[40] Shuyang Ling. Local geometry determines global landscape in low-rank factorization for synchronization. *Foundations of Computational Mathematics*, pages 1–33, 2025.

[41] Haihao Lu and Jinwen Yang. cupdlp. jl: A gpu implementation of restarted primal-dual hybrid gradient for linear programming in julia. *arXiv preprint arXiv:2311.12180*, 2023.

[42] Haihao Lu and Jinwen Yang. A practical and optimal first-order method for large-scale convex quadratic programming. *arXiv preprint arXiv:2311.07710*, 2023.

[43] Haihao Lu, Jinwen Yang, Haodong Hu, Qi Huangfu, Jinsong Liu, Tianhao Liu, Yinyu Ye, Chuwen Zhang, and Dongdong Ge. cupdlp-c: A strengthened implementation of cupdlp for linear programming by c language. *arXiv preprint arXiv:2312.14832*, 2023.

[44] Andrew D McRae. Benign landscapes for synchronization on spheres via normalized laplacian matrices. *arXiv preprint arXiv:2503.18801*, 2025.

[45] Andrew D McRae and Nicolas Boumal. Benign landscapes of low-dimensional relaxations for orthogonal synchronization on general graphs. *SIAM Journal on Optimization*, 34(2):1427–1454, 2024.

[46] Renato D. C. Monteiro. Primal–dual path-following algorithms for semidefinite programming. *SIAM Journal on Optimization*, 7(3):663–678, 1997.

[47] Renato DC Monteiro, Arnesh Sujanani, and Diego 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.

[48] NVIDIA. Nvidia hopper architecture in-depth, 2022. Accessed: 2025-05-07.

[49] Liam O'Carroll, Vaidehi Srinivas, and Aravindan Vijayaraghavan. The burer-monteiro sdp method can fail even above the barvinok-pataki bound. *Advances in Neural Information Processing Systems*, 35:31254–31264, 2022.

[50] Wenqing Ouyang, Ting Kei Pong, and Man-Chung Yue. Burer-monteiro factorizability of nuclear norm regularized optimization. *arXiv preprint arXiv:2505.00349*, 2025.

[51] Brendan O'donoghue, Eric Chu, Neal Parikh, and Stephen Boyd. Conic optimization via operator splitting and homogeneous self-dual embedding. *Journal of Optimization Theory and Applications*, 169:1042–1068, 2016.

[52] 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*, pages 1–10, 2018.

[53] Frank Permenter and Pablo Parrilo. Partial facial reduction: simplified, equivalent sdps via approximations of the psd cone. *Mathematical Programming*, 171:1–54, 2018.

[54] Chi Bach Pham, Wynita Griggs, and James Saunderson. A scalable frank-wolfe-based algorithm for the max-cut sdp. In *International Conference on Machine Learning*, pages 27822–27839. PMLR, 2023.

[55] Thomas Pumir, Samy Jelassi, and Nicolas Boumal. Smoothed analysis of the low-rank approach for smooth semidefinite programs. *Advances in Neural Information Processing Systems*, 31, 2018.

[56] James Renegar. Accelerated first-order methods for hyperbolic programming. *Mathematical Programming*, 173(1):1–35, 2019.

[57] Michel Schubiger, Goran Banjac, and John Lygeros. Gpu acceleration of admm for large-scale quadratic programming. *Journal of Parallel and Distributed Computing*, 144:55–67, 2020.

[58] Nimita Shinde, Vishnu Narayanan, and James Saunderson. Memory-efficient structured convex optimization via extreme point sampling. *SIAM Journal on Mathematics of Data Science*, 3(3):787–814, 2021.

[59] Jos F Sturm. Using sedumi 1.02, a matlab toolbox for optimization over symmetric cones. *Optimization methods and software*, 11(1-4):625–653, 1999.

[60] A. Sujanani and R.D.C. Monteiro. An adaptive superfast inexact proximal augmented Lagrangian method for smooth nonconvex composite optimization problems. *J. Scientific Computing*, 97(2), 2023.

[61] Tianyun Tang and Kim-Chuan Toh. Exploring chordal sparsity in semidefinite programming with sparse plus low-rank data matrices. *arXiv preprint arXiv:2410.23849*, 2024.

[62] Tianyun Tang and Kim-Chuan Toh. A feasible method for general convex low-rank sdp problems. *SIAM Journal on Optimization*, 34(3):2169–2200, 2024.

[63] Tianyun Tang and Kim-Chuan Toh. A feasible method for solving an sdp relaxation of the quadratic knapsack problem. *Mathematics of Operations Research*, 49(1):19–39, 2024.

[64] Tianyun Tang and Kim-Chuan Toh. Solving graph equipartition sdps on an algebraic variety. *Mathematical programming*, 204(1):299–347, 2024.

[65] M. J. Todd, K. C. Toh, and R. H. Tütüncü. On the nesterov–todd direction in semidefinite programming. *SIAM Journal on Optimization*, 8(3):769–796, 1998.

[66] Lieven Vandenberghe, Martin S Andersen, et al. Chordal graphs and semidefinite optimization. *Foundations and Trends® in Optimization*, 1(4):241–433, 2015.

[67] Irene Waldspurger and Alden Waters. Rank optimality for the burer–monteiro factorization. *SIAM journal on Optimization*, 30(3):2577–2602, 2020.

[68] Alex L Wang and Fatma Kılınç-Karzan. Accelerated first-order methods for a class of semidefinite programs. *Mathematical Programming*, 209(1):503–556, 2025.

[69] Jie Wang and Liangbing Hu. Solving low-rank semidefinite programs via manifold optimization. *arXiv preprint arXiv:2303.01722*, 2023.

[70] Yifei Wang, Kangkang Deng, Haoyang Liu, and Zaiwen Wen. A decomposition augmented lagrangian method for low-rank semidefinite programming. *SIAM Journal on Optimization*, 33(3):1361–1390, 2023.

[71] Zaiwen Wen, Donald Goldfarb, and Wotao Yin. Alternating direction augmented lagrangian methods for semidefinite programming. *Mathematical Programming Computation*, 2(3):203–230, 2010.

[72] Makoto Yamashita, Katsuki Fujisawa, Mituhiro Fukuda, Kazuhiro Kobayashi, Kazuhide Nakata, and Maho Nakata. *Latest Developments in the SDPA Family for Solving Large-Scale SDPs*, pages 687–713. Springer US, New York, NY, 2012.

[73] Makoto Yamashita, Katsuki Fujisawa, Kazuhide Nakata, Maho Nakata, Mituhiro Fukuda, Kazuhiro Kobayashi, and Kazushige Goto. A high-performance software package for semidefinite programs: Sdpa 7. 02 2010.

[74] Liuqin Yang, Defeng Sun, and Kim-Chuan Toh. Sdpnal+: a majorized semismooth newton-cg augmented lagrangian method for semidefinite programming with nonnegative constraints. *Mathematical Programming Computation*, 7(3):331–366, 2015.

[75] Yinyu Ye. Gset dataset of random graphs. `https://www.cise.ufl.edu/research/sparse/matrices/Gset`, 2003.

[76] Alp Yurtsever, Olivier Fercoq, and Volkan Cevher. A conditional-gradient-based augmented lagrangian framework. In *International Conference on Machine Learning*, pages 7272–7281. PMLR, 2019.

[77] Alp Yurtsever, Ya-Ping Hsieh, and Volkan Cevher. Scalable convex methods for phase retrieval. In *2015 IEEE 6th International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)*, pages 381–384. IEEE, 2015.

[78] Alp Yurtsever, Joel A Tropp, Olivier Fercoq, Madeleine Udell, and Volkan Cevher. Scalable semidefinite programming. *SIAM Journal on Mathematics of Data Science*, 3(1):171–200, 2021.

[79] Richard Y Zhang. Complexity of chordal conversion for sparse semidefinite programs with small treewidth. *Mathematical Programming*, pages 1–37, 2024.

[80] Richard Y Zhang and Javad Lavaei. Sparse semidefinite programs with guaranteed near-linear time complexity via dualized clique tree conversion. *Mathematical programming*, 188:351–393, 2021.

[81] Xin-Yuan Zhao, Defeng Sun, and Kim-Chuan Toh. A newton-cg augmented lagrangian method for semidefinite programming. *SIAM Journal on Optimization*, 20(4):1737–1765, 2010.

[82] Yang Zheng, Giovanni Fantuzzi, Antonis Papachristodoulou, Paul Goulart, and Andrew Wynn. Fast admm for semidefinite programs with chordal sparsity. In *2017 American Control Conference (ACC)*, pages 3335–3340. IEEE, 2017.

[83] Yang Zheng, Giovanni Fantuzzi, Antonis Papachristodoulou, Paul Goulart, and Andrew Wynn. Chordal decomposition in operator-splitting methods for sparse semidefinite programs. *Mathematical Programming*, 180(1):489–532, 2020.