CuClarabel: GPU Acceleration for a Conic Optimization Solver

Yuwen Chen[1], Danny Tse[2], Parth Nobel[3], Paul Goulart[1], and Stephen Boyd[3]

[1]Department of Engineering Science, University of Oxford, Oxford, UK
[2]Department of Computer Science, Stanford University, Stanford, CA
[3]Department of Electrical Engineering, Stanford University, Stanford, CA

Abstract

We present the GPU implementation of the general-purpose interior-point solver Clarabel for convex optimization problems with conic constraints. We introduce a mixed parallel computing strategy that processes linear constraints first, then handles other conic constraints in parallel. This mixed parallel computing strategy currently supports linear, second-order cone, exponential cone, and power cone constraints. We demonstrate that integrating a mixed parallel computing strategy with GPU-based direct linear system solvers enhances the performance of GPU-based conic solvers, surpassing their CPU-based counterparts across a wide range of conic optimization problems. We also show that employing mixed-precision linear system solvers can potentially achieve additional acceleration without compromising solution accuracy.

1 Introduction

We consider the following convex optimization problem [1] with a quadratic objective and conic constraints:

$$
\begin{array}{ll}
\text{minimize} & \frac{1}{2}x^T P x + q^T x \\
\text{subject to} & Ax + s = b, \\
& s \in \mathcal{K},
\end{array} \tag{$\mathcal{P}$}
$$

with respect to x, s and with parameters $A \in \mathbf{R}^{m \times n}$, $b \in \mathbf{R}^m$, $q \in \mathbf{R}^n$ and $P \in SS_+^n$ and variables $x \in \mathbf{R}^n, s \in \mathbf{R}^m$. The cone $\mathcal{K}$ is a closed convex cone. The formulation $(\mathcal{P})$ is very general and can model most conic convex optimization problems in practice. Examples include the optimal power flow problem in power systems [2], model predictive control in control [3, 4], limit analysis of engineering structures in mechanics [5], support vector machines [6] and lasso problems [7] in machine learning, statistics, and signal processing, and portfolio optimization in finance [8, 9]. Linear equality (zero cones) and inequality (nonnegative cones), second-order cone [10], and semidefinite cone [11] constraints have been long supported in standard conic optimization solvers, and support for exponential and power cone constraints was recently included in several state-of-the-art conic optimization solvers [12, 13, 14, 15]. The combination of these cones can represent many more elaborate convex constraints through the lens of disciplined convex programming [16, 17, 18].

The dual problem of $(\mathcal{P})$ is

$$
\begin{aligned}
\text{maximize} \quad & -\tfrac{1}{2}x^T P x - b^T z \\
\text{subject to} \quad & Px + A^T z = -q, \\
& z \in \mathcal{K}^*,
\end{aligned} \tag{$\mathcal{D}$}
$$

with respect to x, z and where $\mathcal{K}^*$ is the dual cone of $\mathcal{K}$. Solving $\mathcal{P}$ and $\mathcal{D}$ is equivalent to solving the Karush-Kuhn-Tucker (KKT) conditions when strong duality holds. On the other hand, the set of strongly primal infeasibility certificates for $(\mathcal{P})$ is

$$
\mathbb{P} = \left\{ z \mid A^T z = 0, \ z \in \mathcal{K}^*, \ \langle b, z \rangle < 0 \right\}, \tag{1}
$$

and the set of strongly dual infeasibility certificates is

$$
\mathbb{D} = \left\{ x \mid Px = 0, \ -Ax \in \mathcal{K}, \ \langle q, x \rangle < 0 \right\}. \tag{2}
$$

Finding an optimal solution with strong duality or detecting a strongly infeasible certificate can be unified into solving a linear complementarity problem with two additional slack variables $\tau, \kappa \geq 0$ [19], which can be reformulated to the following problem [15]:

$$
\begin{aligned}
\text{minimize} \quad & s^T z + \tau \kappa \\
\text{subject to} \quad & \tfrac{1}{\tau} x^T P x + q^T x + b^T z = -\kappa, \\
& Px + A^T z + q\tau = 0, \\
& Ax + s - b\tau = 0, \\
& (s, z, \tau, \kappa) \in \mathcal{K} \times \mathcal{K}^* \times \mathbf{R}_+ \times \mathbf{R}_+,
\end{aligned} \tag{$\mathcal{H}$}
$$

with respect to x, s, z, τ, κ. In [15] it is also shown that $\mathcal{H}$ is always (asymptotically) feasible, and we can recover either an optimal solution or a strong infeasibility certificate of $(\mathcal{P})$ and $(\mathcal{D})$, depending on the value of the optimal solution $(x^\star, z^\star, s^\star, \tau^\star, \kappa^\star)$ to $(\mathcal{H})$:

i) If $\tau^\star > 0$ then $(x^\star/\tau^\star, s^\star/\tau^\star)$ is an optimal solution to $(\mathcal{P})$ and $(x^\star/\tau^\star, z^\star/\tau^\star)$ is an optimal solution to $(\mathcal{D})$.

ii) If $\kappa^\star > 0$ then at least one of the following holds:

- $(\mathcal{P})$ is strongly infeasible and $z^\star \in \mathbb{P}$.
- $(\mathcal{D})$ is strongly infeasible and $x^\star \in \mathbb{D}$.

The optimal solution $\tau^\star, \kappa^\star$ satisfy the complementarity slackness condition, i.e., at most one of $\tau^\star, \kappa^\star$ is nonzero. The pathological case $\tau^\star = \kappa^\star = 0$ has been discussed in [19].

The *interior-point method* [20] is a popular choice for solving $(\mathcal{H})$. However, it usually requires to factorize linear systems that are increasingly ill-conditioned. Since the complexity of matrix factorizations scales with respect to the number of nonzero entries within a linear system, it is time-consuming to solve large-scale conic optimization problems with interior-point methods.

1.1 Our contribution

We describe a Julia GPU implementation of the general-purpose interior-point solver Clarabel [15]. In our implementation, we support cases where $\mathcal{K}$ is an intersection of *atomic* cones: zero cones, nonnegative cones, second order cones, exponential cones, and power cones. We propose a *mixed parallel computing strategy* that parallelizes computing for each type of cone, integrates the CUDSS [21] library for linear system solving, and supports *mixed-precision* linear system solves for moderate speed improvements. Furthermore, we evaluate our solver against others across a variety of conic optimization problems. Our implementation of CuClarabel is available at `https://github.com/cvxgrp/CuClarabel`.

1.2 Related work

Interior-point methods and solver development. The interior-point method [20] was first discovered by Dikin [22], and became more mainstream after the conception of Karmarkar's method [23], a polynomial-time algorithm for linear programming, and Renegar's [24] path following method. Interior-point methods are known for their ability to solve conic optimization problems to high precision, and is chosen as the default algorithm for many conic optimization solvers [12, 25, 15, 26, 27]. Common variations of the interior-point method include potential reduction methods and path-following methods.

Interior-point methods employ a Newton-like strategy to compute a search direction at every iteration. Unfortunately, the matrix factorization required to compute this direction which scales with the dimension of an optimization problem, rendering very large problems difficult to solve. Developing efficient interior-point methods on exotic cones directly is a promising research direction [26, 27] to alleviate this computational burden. Using exotic cones, we can represent equivalent problems with significantly fewer variables and exploit sparse structure within these exotic cones for efficient implementation of interior-point methods [25, 28, 29]. However, operations within an exotic cone can hardly be parallelized, while parallelism across heterogeneous cones of different dimensionalities will introduce significant synchronization delay.

GPU acceleration in optimization algorithms. GPUs are playing an increasingly significant role in scientific computing. In optimization, GPUs are used to run solvers based on first-order methods. For example, CuPDLP [30] is based on the popular PDHG algorithm [31], which requires only matrix multiplication and addition without the use of direct methods (i.e. it is *factorization-free*). Solvers that require the solution to linear systems, like SCS [13] and CuOSQP [32], have relied on indirect iterative methods—such as the conjugate gradient (CG), the minimal residual (MINRES) [33], and the generalized minimum residual (GMRES) [34] methods—to solve these systems on GPUs. However, the linear systems solved in first-order methods are generally much better conditioned than those encountered in interior-point methods, where the linear systems become increasingly ill-conditioned as the iterations progress. As an interior-point method approaches higher precision, the number of iterations for each inner indirect linear solves increases significantly, which will eventually offset benefits of GPU parallelism and make GPU-based interior-point methods less preferable compared to CPU-based solvers with direct methods in overall computational time [35]. Recently, NVIDIA released the CUDSS package [36], which provides fast direct methods on GPUs for sparse linear systems. Previous work has integrated CUDSS in a nonlinear optimization solver [37], resulting in significant speed-up on large-scale problems compared to its CPU-based counterpart.

Mixed-precision methods. Mixed-precision, or multiprecision, methods [38, 39] have become progressively more popular due to their synergies with modern GPU architectures. For example, in lower-precision configuration one enjoys significant speedup in algorithms, as modern architectures have 32-bit implementations that are around twice as fast as their 64-bit counterparts [38, 40]. Mixed precision methods aim to capitalize on the computational benefits of performing expensive operations in lower precision, while maintaining (or reducing the negative impact to) the superior numerical accuracy from higher precision. Classically, mixed-precision methods for direct (linear system solving) methods involve factorizing a matrix in lower precision, and then applying iterative refinement. In this approach, only the more expensive steps, such as the matrix factorizations and backsolves, are done in lower precision. Mixed-precision methods are used throughout scientific computing, with applications including BLAS operations [41], different linear system solvers such as Krylov methods (CG, GMRES) [39, 42, 43], solving partial differential equations [44], and training deep neural networks [44, 45, 46].

1.3   Paper outline

In §2, we review and discuss the interior point method used by Clarabel [15] for solving conic optimization problems. In §3, we outline how to implement parallel computation for cone operations and solve linear systems within our GPU solver. §4 details our numerical experiments. We detail the supported cones in §A, how to calculate the barrier functions in §B and scaling matrices for each cone in §C.

2   Overview of Clarabel

We first sketch briefly the main operations used in the Clarabel solver [15] for computing a solution to $(\mathcal{P})$. Solving the equivalent problem $(\mathcal{H})$ amounts to finding a root of the following nonlinear equations

$$
G(x, z, s, \tau, \kappa) = \begin{bmatrix} 0 \\ s \\ \kappa \end{bmatrix} - \begin{bmatrix} P & A^T & q \\ -A & 0 & b \\ -q^T & -b^T & 0 \end{bmatrix} \begin{bmatrix} x \\ z \\ \tau \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \frac{1}{\tau} x^T P x \end{bmatrix} = 0, \tag{3}
$$

$$
(x, z, s, \tau, \kappa) \in \mathcal{F} = \mathbf{R}^n \times \mathcal{K} \times \mathcal{K}^* \times \mathbf{R}_+ \times \mathbf{R}_+.
$$

Besides the zero cone that is a linear constraint, other supported conic constraints are smoothed by nonlinear equations in pairs within an interior-point method,

$$
s = -\mu \nabla f(z), \quad \tau \kappa = \mu, \tag{4}
$$

where μ is the smoothing parameter and $f(\cdot)$ is the logarithmically homogeneous self-concordant barrier (LHSCB) function for cone $\mathcal{K}^*$. The barrier functions for different cones are detailed in §B. The trajectory (also called the *central path*)

$$
G(v) = \mu G(v^0),
$$
$$
s = -\mu \nabla f(z), \quad \tau \kappa = \mu, \tag{5}
$$

where $v = (x, z, s, \tau, \kappa)$, characterizes the solution of (3) in the right limit $\mu \to 0$, given an initial point v^0. After starting from v^0, the Clarabel solver iterates the following steps for each iteration k.

Update residuals and check the termination condition.   We update several key metrics at the start of each iteration. Defining the normalized variables $\bar{x} = x/\tau, \bar{s} = s/\tau, \bar{z} = z/\tau$, the primal and dual residuals are then

$$
r_p = A\bar{x} - \bar{s} + b,
$$
$$
r_d = P\bar{x} + A^\top \bar{z} + q,
$$

with corresponding primal and dual objectives

$$
g_p = \frac{1}{2} \bar{x}^\top P \bar{x} + q^T x,
$$
$$
g_d = -\frac{1}{2} \bar{x}^\top P \bar{x} - b^T \bar{z},
$$

and complementarity slackness $\mu = \frac{s^\top z + \kappa \tau}{\nu + 1}$. The solver returns an approximate optimal point if

$$
\|r_p\|_\infty < \epsilon_{\text{feas}} \max\{1, \|b\|_\infty + \|\bar{x}\|_\infty + \|\bar{s}\|_\infty\}
$$
$$
\|r_d\|_\infty < \epsilon_{\text{feas}} \max\{1, \|q\|_\infty + \|\bar{x}\|_\infty + \|\bar{z}\|_\infty\}
$$
$$
|g_p - g_d| < \epsilon_{\text{feas}} \max\{1, \min\{|g_p|, |g_d|\}\}.
$$

Otherwise, it returns a certificate of primal infeasibility if

$$\|A^T z\|_\infty < -\epsilon_{\text{inf}} \max(1, \|x\|_\infty + \|z\|_\infty)(b^T z)$$
$$b^T z < -\epsilon_{\text{inf}},$$

and a certificate of dual infeasibility if

$$\|Px\|_\infty < -\epsilon_{\text{inf}} \max(1, \|x\|_\infty)(b^T z)$$
$$\|Ax + s\|_\infty < -\epsilon_{\text{inf}} \max(1, \|x\|_\infty + \|s\|_\infty)(q^T x)$$
$$q^T x < -\epsilon_{\text{inf}}.$$

Note that $\epsilon_{\text{feas}}, \epsilon_{\text{inf}}$ are predefined parameters within the Clarabel solver.

Find search directions. We then compute Newton-like search directions using a linearization of the central path (5). In other words, we solve the following linear system given some right-hand side residual $d = (d_x, d_z, d_\tau, d_s, d_\kappa)$,

$$\begin{bmatrix} 0 \\ \Delta s \\ \Delta \kappa \end{bmatrix} - \begin{bmatrix} P & A^T & q \\ -A & 0 & b \\ -(q + 2P\xi)^T & -b^T & \xi^T P \xi \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta z \\ \Delta \tau \end{bmatrix} = - \begin{bmatrix} d_x \\ d_z \\ d_\tau \end{bmatrix} \tag{6a}$$

$$H\Delta z + \Delta s = -d_s, \quad \kappa \Delta \tau + \tau \Delta \kappa = -d_\kappa, \tag{6b}$$

where H is the same scaling matrix as in the CPU version of Clarabel [15] and described in §C. We have shown in [15] that solving (6) reduces to solve the next linear system with two different right-hand sides,

$$\underbrace{\begin{bmatrix} P & A^\top \\ A & -H \end{bmatrix}}_{K} \left[\begin{array}{c|c} \Delta x_1 & \Delta x_2 \\ \Delta z_1 & \Delta z_2 \end{array} \right] = \left[\begin{array}{c|c} d_x & -q \\ -(d_z - d_s) & b \end{array} \right]. \tag{7}$$

After solving (7), we recover the search direction $\Delta = (\Delta x, \Delta z, \Delta \tau, \Delta s, \Delta \kappa)$ using

$$\Delta \tau = \frac{d_\tau - d_\kappa/\tau + (2P\xi + q)^\top \Delta x_1 + b^\top \Delta z_1}{\kappa/\tau + \xi^\top P \xi - (2P\xi + q)^\top \Delta x_2 - b^\top \Delta z_2}$$

$$= \frac{d_\tau - d_\kappa/\tau + q^\top \Delta x_1 + b^\top \Delta z_1 + 2\xi^\top P \Delta x_1}{\|\Delta x_2 - \xi\|_P^2 - \|\Delta x_2\|_P^2 - q^\top \Delta x_2 - b^\top \Delta z_2}, \tag{8a}$$

and

$$\Delta x = \Delta x_1 + \Delta \tau \Delta x_2, \quad \Delta z = \Delta z_1 + \Delta \tau \Delta z_2, \tag{8b}$$

$$\Delta s = -d_s - H\Delta z, \quad \Delta \kappa = -(d_\kappa + \kappa \Delta \tau)/\tau. \tag{8c}$$

In an interior-point method with a predictor-corrector scheme, we need to solve (6) with two different values for d. The first is for the affine step (predictor) with

$$d = (G(x, z, s, \tau, \kappa), \kappa\tau, s).$$

The other assigns d as

$$(d_x, d_z, d_\tau) = (1 - \sigma)G(x, z, s, \tau, \kappa), \quad d_\kappa = \kappa\tau + \Delta\kappa\Delta\tau - \sigma\mu,$$
$$d_s = \begin{cases} W^\top (\lambda \backslash (\lambda \circ \lambda + \eta - \sigma\mu \mathbf{e})) & \text{(symmetric)} \\ s + \sigma\mu \nabla f(z) + \eta & \text{(nonsymmetric)}, \end{cases} \tag{9}$$

in the combined step (predictor+corrector), where $\lambda = W^{-\top}s = Wz$ and $\mathbf{e}$ is the idempotent for a symmetric cone with the product operator '$\circ$' and its inverse operator '$\backslash$' [47]. Here η denotes a higher-order correction term, which is a heuristic technique that can significantly accelerate the convergence of interior-point methods [48]. We set it to the Mehrotra correction [49]

$$\eta = (W^{-1}\Delta s) \circ (W\Delta z), \tag{10a}$$

for symmetric cones and the 3rd-order correction [50]

$$\eta = -\frac{1}{2}\nabla^3 f(z)[\Delta z^a, \nabla^2 f(z)^{-1}\Delta s^a], \tag{10b}$$

for nonsymmetric cones. The centering parameter σ controls the decreasing speed of both the residual $G(x, z, s, \tau, \kappa)$ and the complementarity slackness μ. which is determined heuristically by value of the affine step size α_a, which is the maximal value ensuring $v + \alpha_a\Delta_a \in \mathcal{F}$.

Update iterates. At the end of each iteration k, we move the current iterate v along the combined direction Δ_c and obtain the new iterate $v + \alpha_c\Delta_c$. The combined step size α_c should satisfy $v + \alpha_c\Delta_c \in \mathcal{F}$, and furthermore we have to ensure that the new iterate $v + \alpha_c\Delta_c$ stays in the neighborhood of the central path (5) [50].

3 GPU formulation of Clarabel

3.1 A primer on GPU programming

Originally developed for computationally demanding gaming applications, graphics processing units (GPUs) are *massively parallel*, *multithreaded*, *manycore processors*, with massive computational power and memory bandwidth. As a result, GPUs are used throughout scientific computing. In this section, we outline some core properties of GPU computing devices to highlight why a GPU implementation is a natural extension to Clarabel. We then discuss our specific implementation details.

Due to their natural parallelism, GPUs differ dramatically from CPUs in the way their transistors are configured. GPUs have a smaller number of caches (blocks where memory access is fast) and instruction processing blocks, and far more, albeit simpler, computational blocks (i.e. arithmetic logic and floating point units). This suggests that CPUs utilize their larger caches to minimize instruction and memory latency within each thread, while GPUs switch between their significantly larger number of threads to hide said latency.

GPUs use a *single instruction, many threads* (SIMT) approach. In practice, the GPU receives a stream of instructions – each instruction is sent to groups of GPU cores (also known as a *warp*) and acts on multiple data in parallel. Each group of GPU cores, as a result, has *single instruction, many data* (SIMD) structure. This contrasts the traditional CPU vector lane approach of *single instruction, single data*. Note that modern CPUs also support SIMD, but at a much smaller scale than GPUs, as implied by the transistor layout. Furthermore, the SIMD structure requires data parallelism for parallel execution on a GPU.

In our approach, we utilize the SIMD structure of the GPU to accelerate our interior-point method. For each iteration of our solving algorithm, we execute the following steps:

Interior-point method. Main computing steps.

---

1. Update the variable v.

2. Update residuals and objective values for termination check.

3. Update the scaling matrix H.

4. Factorize the matrix K in (7).

5. Compute the right-hand sides of (7) for the constant, affine step, and combined steps.

6. Solve the linear systems (7) three times with different right-hand sides.

7. Recover the step direction Δ via (8).

8. Perform a line search and neighborhood check with respect to the central path.

The matrix factorization (step 4) and the back-solve (step 6) are the most time-consuming parts in each iteration of an interior point method [48] and can be computed using the CUDSS package. Steps 1 and 2 contain matrix addition and multiplication operations that have already been supported in CUDA, while steps 3, 5, 7, and 8 include cone operations that should be tailored for each cone. We will detail how to parallelize each of these steps in the next subsection.

3.2 Mixed parallel computing strategy

The cone operations related to steps 3, 5, 7, and 8 above require only information local to each constituent cone, and hence can be executed concurrently with respect to individual cones. We propose the *mixed parallel computing strategy* for a cone operation across different types of cones. Each family of cones is handled in parallel, and families of cones are handled sequentially.

As stated earlier, the cone operations related to zero cones and nonnegative cones are simply vector additions and multiplications, which have already been parallelized in CUDA. In our implementation we aggregate all zero cones into a single zero cone in our preprocessing step. The same holds for nonnegative cones. For the remaining cones, we parallelize within each *family* of cones (i.e., second-order cone, exponential cone, etc.). For each family of cones, we allocate a thread to each cone belonging to that family, and execute in parallel. We synchronize threads after executing for each type of cone. This adheres to the SIMD computing paradigm, as each type of cone has its own set of instructions for updating its scaling matrix.

Note that the SIMD GPU computing structure naturally favors *balanced workloads*, i.e., the workloads in each thread should be similar so that the synchronization will not take too much time. The exponential and power cones are 3-dimensional nonsymmetric cones that all cone operations have balanced workload among the same class of cones, thus we can parallelize the computation for each cone. On the other hand, the second-order cones may vary in dimensionality, which may not mesh well with the SIMD computing paradigm. In most use cases for large-scale second-order cone problems, e.g. optimal power flow problems [2] and finite-element problems [5], the second-order cones are of small (less than 5) dimensionality; thus the effect of this workload imbalance is negligible. Regardless, we support second-order cones of all sizes. In the next section, we outline a preprocessing procedure for our second-order cones we use to balance workloads.

We illustrate our mixed parallel computing strategy via step 3, the scaling matrix update. Recall that the conic constraint $\mathcal{K}$ can be decomposed as a Cartesian product of p constituent atomic cones $\mathcal{K}_1 \times \cdots \times \mathcal{K}_p$ that are ordered by cone types. We assume $\mathcal{K}_1$ is a zero cone, $\mathcal{K}_2$ is a nonnegative cone, $\mathcal{K}_3$ to $\mathcal{K}_i$ are second-order cones, $\mathcal{K}_{i+1}$ to $\mathcal{K}_j$ are exponential cones and $\mathcal{K}_{j+1}$ to $\mathcal{K}_p$ are power cones. Due to our composition into constituent atomic cones, H is a

block-diagonal matrix,

$$
H = \begin{bmatrix} H_1 & & \\ & \ddots & \\ & & H_p \end{bmatrix},
$$

where H_1 and H_2 are diagonal matrices and each block $H_t, t \geq 3$ is the scaling matrix corresponding to cone $\mathcal{K}_t$ of small dimensionality.

For the update of H at iteration $k + 1$, we first set diagonal terms of the scaling matrix H_1^{k+1} for the zero cone to all 0s, and then update the diagonals of the scaling matrix H_2^{k+1} for the nonnegative cone by element-wise vector division, as in the Nesterov-Todd (NT) scaling [51]. These computations are already parallelized by CUDA. For conic constraints other than the zero cone and the nonnegative cone, the corresponding parts in the scaling matrix H are no longer diagonal, but we observe H_t only requires local information within each constituent cone, and thus we can solve for each H_t concurrently, independent to the other cones. We implement kernel functions `_soc_update_H()`, `_exp_update_H()` and `_pow_update_H()` for second-order cones, exponential cones and power cones respectively. The kernel function will process one cone operation per thread, and the kernel functions belongs to the same class of cones are executed in parallel and finally synchronized by calling `@sync`. Details regarding the update functions for each type of cone and Algorithm 1 illustrating the update of scaling matrix H can also be found in §C.

Since the mixed parallel computing strategy is independent of choices of optimization algorithms, it is also applicable for GPU implementations for cone operations in first-order operator-splitting conic solvers, like SCS [13] and COSMO [14].

3.3   Second-order cone preprocessing for balanced workloads

We can decompose any large second-order cone into a set of small second-order cones of fixed dimensionality d. The workload is then balanced for each second-order cone, adhering to the SIMD computing paradigm. For an n-dimensional second-order cone $\mathcal{K}_{soc}^n\{x \in \mathbf{R}^n \mid x_1 \geq \sqrt{\sum_{i=2}^n x_i^2}\}$, we can decompose it into the equivalent set of $q = \lceil (n-2)/(d-2) \rceil$ second-order cones:

$$
\begin{aligned}
x_1 &\geq \sqrt{x_2^2 + \cdots + x_{d-1}^2 + z_1^2} \\
z_1 &\geq \sqrt{x_d^2 + \cdots + x_{2d-2}^2 + z_2^2} \\
&\vdots \\
z_q &\geq \sqrt{x_{n-d}^2 + \cdots + x_n^2}
\end{aligned},
$$

where $z \in \mathbf{R}^q$ is an a slack variable. Each of the first $q - 1$ second-order cones are d-dimensional, while the qth cone has dimension not larger than d. The workload is now balanced among these p cones.

3.4   Data structures

Since data parallelism is required for parallel execution on GPUs, we manage data for each cone as a structure of arrays (SoA) in our GPU implementation, in contrast to the existing CPU counterpart which uses an array of structures (AoS). In other words, instead of creating structs for $\mathcal{K}_i, i = 1, \ldots, p$ and storing pointers to each struct in an array, we concatenate the same type of local variable from different cones into a global variable and then store it in a global struct for $\mathcal{K}$, which is illustrated in Figure 1.
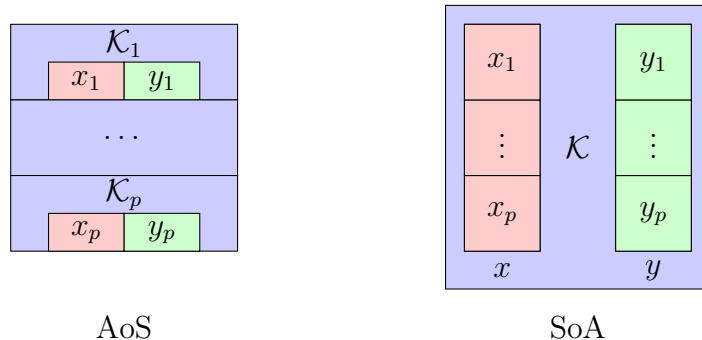
Figure 1: Illustration of AoS (CPU) and SoA (GPU) data structure

In addition, indexing cones in order in the setup phase can simplify the implementation of mixed parallel computing strategy. We reorder the input cones such that memory is coalescing for the same class of cones, which can accelerate computation on GPUs. Matrices are stored in the compressed sparse row (CSR) format. Instead of storing only the triangular part of a square matrix as in the CPU-based Clarabel, we store the full matrix for more efficient multiplication on the GPU.

3.5 Solving linear systems

Most of the computation time for our interior point method is spent in factorizing the matrix K and in the three backsolve operations in (7). Our GPU implementation also leverages the power of the newly released sparse linear system solver CUDSS [36] for the LDL^T factorization and backsolve operations. The iterative refinement [52] is implemented to increase the numerical stability of the backsolve operation.

Currently, a GPU has more computing cores for FLOAT32 than FLOAT64 and hence better performance for parallel algorithms. Also, FLOAT32 requires less memory and takes less time for the same computation than FLOAT64. However, lowering the precision will introduce numerical instability for solving linear systems. Thus, we employ a *mixed precision* solve in our matrix solves. Specifically, we use mixed precision for data within the iterative refinement step:

1. Solve for the residual at step i: $r_i = b - Kx_{i-1}$, where x_{i-1} is our guess for iteration $i-1$. (Full precision)

2. Solve the linear system under a regularization parameter $(K + \delta I)\Delta_i = r_i$. (Lower precision)

3. Take $x_i = x_{i-1} + \Delta_i$. (Full precision)

In all our matrix solves, given matrix K and right-hand side b, we factorize the lower precision copy of K, and then solve the linear system in this lower precision. A regularization parameter δ is added to K to increase the numerical stability of matrix factorization. It is the sum of a static regularization and a dynamic regularization. To offset the regularization effect and rounding errors from the lower precision in backsolves, we solve in full precision for the other steps, i.e., we compute the residual r_i and save the update $\{x_i\}$ in full precision. We loop until either (1) we reach a pre-specified number of max steps for iterative refinement, or (2) the ℓ_∞ norm of

$b - K x_i$ reaches a certain threshold, specifically $\|b - K x_i\|_\infty \leq t_{\text{abs}} + t_{\text{rel}} \|b\|_\infty$. For the mixed precision, we set the static regularization to the square root of machine precision, i.e., $\sqrt{\epsilon}$, and take a maximum of 10 iterative refinement steps. For the full-precision solve, t_{abs} (absolute tolerance) and t_{rel} (relative tolerance) are both 10^{-12} by default.

Although the mixed precision for the iterative refinement can improve the numerical stability for solving linear system in lower precision, it is to be expected that the mixed precision may take longer time to converge or fail in cases where the matrix K is extremely ill-conditioned. Caution is required when using mixed precision for numerically hard problems, e.g., conic programs with exponential cones.

4 Numerical experiments

We have benchmarked our Julia GPU implementation of the Clarabel solver[1] against the state-of-the-art commercial interior-point solvers Mosek [12] and Gurobi [53]. We also include our Rust CPU implementation of Clarabel solver with the 3rd-party multithreaded supernodal LDL factorization method in the *faer-rs* package [54]. We have included benchmark results for several classes of problems including quadratic programming (QP), second-order cone programming (SOCP) and exponential cone programming. All benchmarks are performed using the default settings for each solver, with pre-solve disabled where applicable to ensure equivalent problem-solving conditions. No additional iteration limits are imposed beyond each solver's internal defaults. All experiments were carried out on a workstation with Intel(R) Xeon(R) w9-3475X CPU @ 4.8 GHz with 256 GB RAM and NVIDIA GeForce RTX 4090 24GB GPU. All benchmarks tests are scripted in Julia and access solver interfaces via JuMP [82]. We use Rust compiler version 1.76.0 and Julia version 1.10.2.

4.1 Benchmarking metrics

We choose the same benchmarking tests as used in the Clarabel solver [15], and compare our results using metrics that are commonly used when comparing solve time across different solvers. For a set of N test problems, we define the *shifted geometric mean g_s* as

$$g_s = \left[\prod_{p=1}^{N} (t_{p,s} + k) \right]^{\frac{1}{N}} - k,$$

where $t_{p,s}$ is the time in seconds for solver s to solve problem p, and $k = 1$ is the shift. The normalized shifted geometric mean is then defined as

$$r_s = \frac{g_s}{\min_{s'} g_{s'}}.$$

Note that the solver with the lowest shifted geometric mean solve time has a normalized score of 1. We assign a solve time $t_{p,s}$ equal to the maximum allowable solve time if solver s fails to solve the problem p.

The *relative performance ratio* for a solver s and a problem p is defined as

$$u_{p,s} = \frac{t_{p,s}}{\min_{s'} t_{p,s'}}.$$

---

[1]https://github.com/cvxgrp/CuClarabel

The *relative performance profile* denotes the fraction of problems solved by solver s within a factor τ of the solve time of the best solver, which is defined as $f_s^r : \mathbf{R}_+ \mapsto [0,1]$

$$f_s^r(\tau) = \frac{1}{N} \sum_p \mathcal{I}_{\leq \tau}(u_{p,s}),$$

where $\mathcal{I}_{\leq \tau}(u) = 1$ if $\tau \leq u$ and $\mathcal{I}_{\leq \tau}(u) = 0$ otherwise. We also compute the *absolute performance profile* $f_s^a : \mathbf{R}_+ \mapsto [0,1]$, which denotes the fraction of problems solved by solver s within τ seconds and is defined as

$$f_s^a(\tau) = \frac{1}{N} \sum_p \mathcal{I}_{\leq \tau}(t_{p,s}).$$

We showcase plots of performance profiles and tables of shifted geometric means for different classes of problems supported in the Clarabel solver. We included further detailed numerical results for all solvers in §D, including the iteration counts and the total time.

4.2 Quadratic programming

We first present benchmark results for QPs. Note that in this setting, the set $\mathcal{K}$ in $(\mathcal{P})$ is restricted to the composition of zero cones, i.e., linear equality constraints, and nonnegative cones, i.e., linear inequality constraints. We consider two classes of problems, the portfolio optimization problem and the Huber fitting problem.

Portfolio optimization, a problem arising in quantitative finance, aims to allocate assets in a manner that maximizes expected return while keeping risk under control. We can formulate it as

$$\begin{array}{ll} \text{maximize} & \mu^T x - \gamma x^T \Sigma x \\ \text{subject to} & \mathbf{1}^T x = 1, \\ & x \geq 0, \end{array}$$

where $x \in \mathbf{R}^n$ (the variable) represents the ratio of allocated assets, $\mu \in \mathbf{R}^n$ is the vector of expected returns, $\gamma > 0$ is the risk-aversion parameter, and $\Sigma \in \mathbf{S}_+^n$ the risk covariance matrix which is of the form $\Sigma = F F^\top + D$ with $F \in \mathbf{R}^{p \times n}$ and $D \in \mathbf{R}^{p \times p}$ diagonal. We set the rank p to the integer closest to $0.1n$, and vary n from 5000 to 25000.

Huber fitting is a version of robust least squares. For a given matrix $A \in \mathbf{R}^{m \times n}$ and vector $b \in \mathbf{R}^m$, we replace the least squares loss function with the Huber loss. The Huber loss makes the penalty incurred by larger points linear instead of quadratic, thus outliers have a smaller effect on the resulting estimator. Precisely, the problem is stated as:

$$\text{minimize} \sum_{i=1}^m \phi_h \left(a_i^T x - b_i \right)$$

where $a_i^\top$ is the i-th row of A and the Huber loss $\phi_h : \mathbf{R} \to \mathbf{R}$ is defined as

$$\phi_h(t) = \begin{cases} t^2 & |t| \leq T \\ T(2|t| - T) & \text{otherwise} \end{cases}.$$

We set m to the nearest integer of $1.5n$ and vary value of n from 5000 to 25000.

Results for large QP tests are shown in Figure 2. We benchmark ten different examples from two classes above and set the time limit to 1h. We compare our GPU implementation ClarabelGPU with ClarabelRs, the Rust version 0.9.0 version of Clarabel equipped with the *faer* multithreaded linear system solver; and two commercial solvers, Gurobi and Mosek. ClarabelGPU is the fastest

Figure 2: Performance profiles for the large QPs problem set

(a) Relative performance profile

(b) Absolute performance profile

		ClarabelGPU	ClarabelRs	Gurobi	Mosek
Shifted GM	Full Acc.	1.0	11.29	2.13	4.18
	Low Acc.	1.0	11.29	2.13	4.18
Failure Rate (%)	Full Acc.	0.0	20.0	0.0	0.0
	Low Acc.	0.0	10.0	0.0	0.0

(c) Benchmark timings as shifted geometric mean and failure rates

solver on these problems, and it has the lowest per-iteration time for almost all examples. Since most of time of an interior point solver is spent on factorizing and solving a linear system in QPs, we can say that ClarabelGPU benefits from the use of the CUDSS linear system solver and it is more than 2 times faster than Gurobi, about 4 times faster than Mosek and 10x times faster than the existing Rust implementation with the multithreaded *faer* linear system solver.

4.3   Second-order cone programming

We next consider the second-order cone relaxations of optimal power flow problems [55] from the IEEE PLS PGLib-OPF benchmark library [2], using the *PowerModels.jl* package [56] for modeling convenience. Note that only second-order cones of dimensionality 3 or 4 are used in these second-order cone relaxations, which satisfies our assumption in §3.2: that the dimensionality of each cone is very small.

We compare our GPU implementation with the CPU-based Clarabel solver and Mosek solver. We also include results of the Mosek solver with pre-solve for comparison, which is denoted as `Mosek*` in the plots. The maximum termination time is again set to 1h. We benchmark the second-order cone relaxations of 120 problems from the PGLib-OPF library, where the number of second-order cones exceed 2000.

Results for these problems are shown in Figure 3. Both ClarabelGPU and ClarabelRs are faster and more numerically stable than Mosek even with the presolve step. Moreover, the GPU implementation can solve 118 out of 120 examples within an hour, while the Rust version of Clarabel can solve 104 out of 120 examples within the same time limit. In contrast, Mosek with presolve fails on about 40% of the optimal flow problems, and the one without presolve fails on over 90% of the problems.

Overall, the GPU solver is several times faster than Rust-based CPU solver. However, note that ClarabelGPU fails on two examples that ClarabelRs successfully solves. This highlights the

Figure 3: Performance profiles for the large OPF SOCPs problem set

(a) Relative performance profile

(b) Absolute performance profile

		ClarabelGPU	**Mosek***	**ClarabelRs**	**Mosek**
Shifted GM	Full Acc.	1.0	6.63	3.62	34.33
	Low Acc.	1.0	7.02	2.99	36.36
Failure Rate (%)	Full Acc.	1.7	39.0	13.6	90.7
	Low Acc.	0.0	39.0	3.4	90.7

(c) Benchmark timings as shifted geometric mean and failure rates

different numerical performance of the linear system solvers between CUDSS and *faer*.

4.4 Exponential cone programming

We benchmark two classes of exponential cone programming problems with varying dimensionality, the logistic regression and the entropy maximization problems [13].

Logistic regression is a classical machine learning model for binary classification. Given a feature matrix $A \in \mathbf{R}^{m \times n}$ and a vector of labels $b \in \mathbf{R}^m$ such that $b_i \in \{0, 1\}$, we formulate the *sparse* logistic regression problem as:

$$\text{minimize} \quad \sum_{i=1}^m \left(\log \left(1 + \exp \left(x^T a_i \right) \right) - b_i x^T a_i \right) + \lambda \|x\|_1,$$

where x is our variable and a_i^T is the i-th row of A. Note that this is the sparse logistic regression problem due to the ℓ_1-norm penalty on x. Indeed, when $\lambda = 0$, we recover the classical logistic regression problem.

Each entry of A is generated via $A_{ij} \sim \mathcal{N}(0, 1)$. The label b is generated following the same method in [57, 13]. We set $m = 5n$ and vary value of n from 2000 to 10000.

The entropy maximization problem aims to maximize entropy over a probability distribution given a set of m linear inequality constraints, which can be interpreted as bounds on the expectations of arbitrary functions. The problem is formulated as

$$\begin{aligned} \text{maximize} \quad & -\sum_{i=1}^n x_i \log x_i \\ \text{subject to} \quad & 1^T x = 1, \\ & Ax \le b, \end{aligned}$$

13

Figure 4: Performance profiles for the exponential cone programming problem set

(a) Relative performance profile

(b) Absolute performance profile

		ClarabelGPU	**Mosek***	**ClarabelRs**	**Mosek**
Shifted GM	Full Acc.	1.0	1.47	8.42	1.58
	Low Acc.	1.0	1.47	8.42	1.58
Failure Rate (%)	Full Acc.	0.0	0.0	40.0	0.0
	Low Acc.	0.0	0.0	20.0	0.0

(c) Benchmark timings as shifted geometric mean and failure rates

Each element of A is generated from the distribution $A_{ij} \sim \mathcal{N}(0, n)$. Then, we set $b = Av/1^\top v$ where $v \in \mathbf{R}^n$ is generated randomly from $v_i \sim U[0, 1]$, which ensures the problem is always feasible. We set m to the nearest integer of $0.5n$ and vary value of n from 2000 to 10000.

The benchmark results for these two problems with varying dimensionality are shown in Figure 4. From §D, even though the GPU acceleration of Clarabel is offset by nearly doubled number of iterations compared to Mosek, we can still achieve more than 2 times of acceleration for the overall time. That is to say we can possibly achieve more acceleration in ClarabelGPU if we can improve the numerical stability to the same level of Mosek on exponential cone programs. Compared to ClarabelRs, we find ClarabelGPU can benefit from GPU computation up to 20x times faster on logistic regression problems.

4.5 Mixed precision

In the implementation of the mixed precision setting, as described we only set the data type of the CUDSS linear system solver to FLOAT32. Since the use of mixed precision accelerates the numerical factorization rather than the symbolic factorization within a factorization method, we only record the computational time for an interior point method without the setup time.

We test the mixed precision setting on QPs, including portfolio optimization and Huber fitting problems from §4.2. We compare it with the standard GPU implementation of FLOAT64 data type. The results in Table 1 show that the use of mixed precision strategy can reduce the solve time up to a factor of 2. This will be beneficial to problems like the parametric programming, where the setup is needed only once and the symbolic factorization structure can be reused repeatedly later on [58]. However, we also notice that the mixed precision may not be sufficient for conic optimization problems where the Hessian block H in (7) is ill-conditioned, especially for the exponential cone that the exponential component causes huge variation in the magnitude of different entries.

14

Table 1: Solve times and iteration counts for the mixed precision QP test

	iterations		total time (s)	
Problem	Full	Mixed	Full	Mixed
PORTFOLIO_OPTIMIZATION_N_10000	19	19	0.729	0.631
PORTFOLIO_OPTIMIZATION_N_25000	19	21	7.08	4
PORTFOLIO_OPTIMIZATION_N_20000	19	23	3.91	2.54
HUBER_FITTING_N_5000	9	9	2.61	1.32
HUBER_FITTING_N_25000	10	10	143	58.3
HUBER_FITTING_N_10000	10	10	12.6	5.76
PORTFOLIO_OPTIMIZATION_N_5000	17	18	0.196	0.259
HUBER_FITTING_N_15000	10	10	34.9	15.1
PORTFOLIO_OPTIMIZATION_N_15000	19	19	1.88	1.18
HUBER_FITTING_N_20000	9	9	69.8	29.1

5 Conclusion

We have developed a GPU interior point solver for conic optimization.[2] We propose a mixed parallel computing strategy to process linear constraints with second-order cone, exponential cone and power cone constraints. The GPU solver shows several times of acceleration compared to state-of-the-art CPU conic solvers on many problems to high precision, including QPs, SOCPs and exponential programmings.

We also note that some initialization steps, such as population of the KKT matrix and matrix equilibration, take non-negligible time for large problems. We will also move the setup steps onto a GPU in our future development. The proposed mixed parallel computing strategy for GPU implementation is also applicable for conic solvers based on first-order operator-splitting methods.

Acknowledgments

References

[1] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004.

[2] Sogol Babaeinejadsarookolaee, Adam Birchfield, Richard Christie, Carleton Coffrin, Christopher DeMarco, Ruisheng Diao, Michael Ferris, Stephane Fliscounakis, Scott Greene, Renke Huang, et al. The power grid library for benchmarking AC optimal power flow algorithms. *arXiv:1908.02788*, 2019.

[3] Francesco Borrelli, Alberto Bemporad, and Manfred Morari. *Predictive control for linear and hybrid systems.* Cambridge University Press, 2017.

---

[2]`https://github.com/cvxgrp/CuClarabel`

[4] Stephen Boyd, Laurent El Ghaoui, Eric Feron, and Venkataramanan Balakrishnan. *Linear Matrix Inequalities in System and Control Theory*. Society for Industrial and Applied Mathematics, 1994.

[5] Athanasios Makrodimopoulos and Chris Martin. Upper bound limit analysis using simplex strain elements and second-order cone programming. *International Journal for Numerical and Analytical Methods in Geomechanics*, 31(6):835–865, 2007. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nag.567.

[6] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[7] Robert Tibshirani. Regression shrinkage and selection via the Lasso. *Journal of the Royal Statistical Society: Series B*, 58(1):267–288, 1996.

[8] Pavlo Krokhmal. Higher moment coherent risk measures. *Quantitative Finance*, 7(4):373–387, 2007.

[9] Alexander Vinel and Pavlo Krokhmal. Certainty equivalent measures of risk. *Annals of Operations Research*, 249(1-2):75–95, 2017.

[10] Miguel Sousa Lobo, Lieven Vandenberghe, Stephen Boyd, and Hervé Lebret. Applications of second-order cone programming. *Linear Algebra and its Applications*, 284(1):193–228, 1998. International Linear Algebra Society (ILAS) Symposium on Fast Algorithms for Control, Signals and Image Processing.

[11] Lieven Vandenberghe and Stephen Boyd. Semidefinite programming. *SIAM Review*, 38(1):49–95, 1996.

[12] MOSEK ApS. *MOSEK Modeling Cookbook*, 2023.

[13] 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(3):1042–1068, 2016.

[14] 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.

[15] Paul Goulart and Yuwen Chen. Clarabel: An interior-point solver for conic programs with quadratic objectives, May 2024. arXiv:2405.12762 [math].

[16] Michael Grant. *Disciplined Convex Programming*. PhD thesis, Stanford University, 2004.

[17] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. `https://cvxr.com/cvx`, March 2014.

[18] Anqi Fu, Balasubramanian Narasimhan, and Stephen Boyd. CVXR: An R package for disciplined convex optimization. *Journal of Statistical Software*, 94(14):1–34, 2020.

[19] Brendan O'Donoghue. Operator Splitting for a Homogeneous Embedding of the Linear Complementarity Problem. *SIAM Journal on Optimization*, 31(3):1999–2023, January 2021.

[20] Yurii Nesterov and Arkadii Nemirovskii. *Interior-Point Polynomial Algorithms in Convex Programming*. Society for Industrial and Applied Mathematics, 1994.

[21] Cudss.jl: Julia interface for nvidia cudss, 2024. https://github.com/exanauts/CUDSS.jl.

[22] IIliya Dikin. Iterative solution of problems of linear and quadratic programming. *Sov. Math., Dokl.*, 8:674–675, 1967.

[23] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, page 302–311, New York, NY, USA, 1984. Association for Computing Machinery.

[24] James Renegar. A polynomial-time algorithm, based on newton's method, for linear programming. *Math. Program.*, 40(1–3):59–93, jan 1988.

[25] Alexander Domahidi, Eric Chu, and Stephen Boyd. ECOS: An SOCP solver for embedded systems. In *2013 European Control Conference (ECC)*, pages 3071–3076, 2013.

[26] Chris Coey, Lea Kapelevich, and Juan Pablo Vielma. Solving natural conic formulations with Hypatia.jl. *INFORMS Journal on Computing*, 34(5):2686–2699, 2022.

[27] Mehdi Karimi and Levent Tunçel. Domain-driven solver (DDS) version 2.1: a matlab-based software package for convex optimization problems in domain-driven form. *Mathematical Programming Computation*, 10 2023.

[28] Martin Andersen, Joachim Dahl, and Lieven Vandenberghe. Implementation of nonsymmetric interior-point methods for linear optimization over sparse matrix cones. *Mathematical Programming Computation*, 2(3):167–201, 2010.

[29] Yuwen Chen and Paul Goulart. An efficient IPM implementation for a class of nonsymmetric cones, 2023.

[30] Haihao Lu and Jinwen Yang. cuPDLP.jl: A GPU Implementation of Restarted Primal-Dual Hybrid Gradient for Linear Programming in Julia, December 2023.

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

[32] 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, October 2020. arXiv:1912.04263 [math].

[33] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, January 2003.

[34] Youcef Saad and Martin Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[35] Edmund Smith, Jacek Gondzio, and Julian Hall. GPU Acceleration of the Matrix-Free Interior Point Method. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics*, pages 681–689, Berlin, Heidelberg, 2012. Springer.

[36] Nvidia Inc. CUDSS, 2023. https://developer.nvidia.com/cudss.

[37] François Pacaud, Sungho Shin, Alexis Montoison, Michel Schanen, and Mihai Anitescu. Condensed-space methods for nonlinear programming on GPUs, May 2024. arXiv:2405.14236 [math].

[38] Ahmad Abdelfattah, Hartwig Anzt, Erik Boman, Erin Carson, Terry Cojean, Jack Dongarra, Mark Gates, Thomas Grützmacher, Nicholas Higham, Xiaoye Sherry Li, Neil

Lindquist, Yang Liu, Jennifer A. Loe, Piotr Luszczek, Pratik Nayak, Srikara Pranesh, Sivasankaran Rajamanickam, Tobias Ribizel, Barry Smith, Kasia Swirydowicz, Stephen Thomas, Stanimire Tomov, Yaohung Tsai, Ichitaro Yamazaki, and Ulrike Meier Yang. A survey of numerical methods utilizing mixed precision arithmetic. *CoRR*, abs/2007.06674, 2020.

[39] Nicholas Higham and Theo Mary. Mixed precision algorithms in numerical linear algebra. *Acta Numerica*, 31:347–414, 2022.

[40] Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. Accelerating scientific computations with mixed precision algorithms. *CoRR*, abs/0808.2794, 2008.

[41] Ahmad Abdelfattah, Timothy Costa, Jack Dongarra, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas Higham, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Mawussi Zounon. A set of batched basic linear algebra subprograms and lapack routines. *ACM Trans. Math. Softw.*, 47(3), June 2021.

[42] Erin Carson and Tomáš Gergelits. Mixed precision s-step lanczos and conjugate gradient algorithms, 2021.

[43] Serge Gratton, Ehouarn Simon, David Titley-Peloquin, and Philippe Toint. Exploiting variable precision in gmres, 2020.

[44] Joel Hayford, Jacob Goldman-Wetzler, Eric Wang, and Lu Lu. Speeding up and reducing memory usage for scientific machine learning via mixed precision, 2024.

[45] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2018.

[46] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications, 2015.

[47] Lieven Vandenberghe. *The CVXOPT linear and quadratic cone program solvers*, 2010.

[48] Stephen Wright. *Primal-Dual Interior-Point Methods*. Society for Industrial and Applied Mathematics, 1997.

[49] Sanjay Mehrotra. On the Implementation of a Primal-Dual Interior Point Method. *SIAM Journal on Optimization*, 2(4):575–601, November 1992. Publisher: Society for Industrial and Applied Mathematics.

[50] Joachim Dahl and Erling Andersen. A primal-dual interior-point algorithm for nonsymmetric exponential-cone optimization. *Mathematical Programming*, 2021.

[51] Yurii Nesterov and Michael Todd. Primal-dual interior-point methods for self-scaled cones. *SIAM Journal on Optimization*, 8(2):324–364, 1998.

[52] Jorge Nocedal and Stephen Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, New York, NY, 2006.

[53] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[54] A linear algebra foundation for the Rust programming language. `https://github.com/sarah-ek/faer-rs`. Accessed: 2024-15-05.

[55] Rabih Jabr. Radial distribution load flow using conic programming. *IEEE Transactions on Power Systems*, 21(3):1458–1459, Aug 2006.

[56] Carleton Coffrin, Russell Bent, Kaarthik Sundar, Yeesian Ng, and Miles Lubin. Powermodels.jl: An open-source framework for exploring power flow formulations. In *2018 Power Systems Computation Conference (PSCC)*, pages 1–8, June 2018.

[57] Christopher Fougner and Stephen Boyd. Parameter Selection and Preconditioning for a Graph Form Solver. In Roberto Tempo, Stephen Yurkovich, and Pradeep Misra, editors, *Emerging Applications of Control and Systems Theory: A Festschrift in Honor of Mathukumalli Vidyasagar*, pages 41–61. Springer International Publishing, Cham, 2018.

[58] Martin Herceg, Michal Kvasnica, Colin Jones, and Manfred Morari. Multi-Parametric Toolbox 3.0. In *Proc. of the European Control Conference*, pages 502–510, 2013.

[59] Levent Tunçel. Generalization of primal-dual interior-point methods to convex optimization problems in conic form. *Foundations of Computational Mathematics*, 1(3):229–254, 2001.

A   Supported cones in Clarabel and CuClarabel

We support the following atomic cones in our GPU solver:

- The *zero cone*, defined as

$$\{0\}^n = \{x \in \mathbf{R}^n \mid x_i = 0, \ i = 1, \ldots, n\}.$$

  The dual cone of the zero cone is $(\{0\}^n)^* = \mathbf{R}^n$.

- The *nonnegative cone*, defined as

$$\mathbf{R}_+^n = \{x \in \mathbf{R}^n \mid x_i \geq 0, \ i = 1, \ldots, n\}.$$

  The nonnegative cone is a self-dual convex cone, i.e., $(\mathbf{R}_+^n)^* = \mathbf{R}_+^n$.

- The *second-order cone* $\mathcal{K}_{\mathrm{soc}}^n$ (also called the *quadratic* or *Lorentz cone*), defined as

$$\mathcal{K}_{\mathrm{soc}}^n = \left\{ (t, x) \mid x \in \mathbf{R}^{n-1}, \ t \in \mathbf{R}_+, \ \|x\|_2 \leq t \right\}.$$

  The second-order cone is self-dual, i.e., $\mathcal{K}_{\mathrm{soc}} = \mathcal{K}_{\mathrm{soc}}^*$.

- The *exponential cone*, a 3-dimensional cone defined as

$$\mathcal{K}_{\exp} = \left\{ (x, y, z) \mid y > 0, \ y \exp\left(\frac{x}{y}\right) \leq z \right\} \cup \{(x, 0, z) \mid x \leq 0, \ z \geq 0\},$$

  with its dual cone given by

$$\mathcal{K}_{\exp}^* = \left\{ (u, v, w) \mid u < 0, \ -u \exp\left(\frac{v}{u}\right) \leq \exp(1)w \right\} \cup \{(0, v, w) \mid v \geq 0, \ w \geq 0\}.$$

- The 3-dimensional *power cone* with exponent $\alpha \in (0, 1)$, defined as

$$\mathcal{K}_{\mathrm{pow}, \alpha} = \left\{ (x, y, z) \mid x^\alpha y^{1-\alpha} \geq |z|, \ x \geq 0, \ y \geq 0 \right\},$$

  with its dual cone given by

$$\mathcal{K}_{\mathrm{pow}, \alpha}^* = \left\{ (u, v, w) \,\middle|\, \left(\frac{u}{\alpha}\right)^\alpha \left(\frac{v}{1-\alpha}\right)^{1-\alpha} \geq |w|, \ u \geq 0, \ v \geq 0 \right\}.$$

B   Barrier functions for different cones

For cones introduced in §A, the corresponding logarithmically-homogeneous self-concordant barrier functions with degree ν are given by:

1. *Nonnegative cone* $\mathbf{R}_+^n$ of degree n:

$$f(z) = -\sum_{i \in [\![n]\!]} \log(z_i), \ z \in \mathbf{R}_+^n.$$

2. *Second order cone* $\mathcal{K}_q^n$ of degree 1:

$$f(z) = -\frac{1}{2} \log\left(z_1^2 - \sum_{i=2}^n z_i^2 \right), \ z \in \mathcal{K}_q^n.$$

3. *Dual exponential cone* $\mathcal{K}^*_{\exp}$ *of degree 3:*

$$f(z) = -\log\left(z_2 - z_1 - z_1 \log\left(\frac{z_3}{-z_1}\right)\right) - \log(-z_1) - \log(z_3), \ z \in \mathcal{K}^*_{\exp}.$$

4. *Dual power cone* $\mathcal{K}^*_{\text{pow}}$ *of degree 3:*

$$f(z) = -\log\left(\left(\frac{z_1}{\alpha}\right)^{2\alpha}\left(\frac{z_2}{1-\alpha}\right)^{2(1-\alpha)} - z_3^2\right) - (1-\alpha)\log(z_1) - \alpha\log(z_2), \ z \in \mathcal{K}^*_{\text{pow}}.$$

C   Scaling matrices

The scaling matrix H in (6) is the Jacobian from the linearization of the central path (5). We set $H = 0$ for the zero cone and choose the NT scaling [51] for nonnegative and second-order cones, which are both symmetric cones. The NT scaling method exploits the self-scaled property of symmetric cone $\mathcal{K}$ where exists a unique scaling point $w \in \mathcal{K}$ satisfying

$$H(w)s = z.$$

The matrix $H(w)$ can be factorized as $H^{-1}(w) = W^\top W$, and we set $H = H^{-1}(w)$ in (6). The factors w, W are then computed following [47].

For exponential and power cones that are not symmetric, the central path is defined by the set of point satisfying

$$Hz = s, \quad H\nabla f^*(s) = \nabla f(z),$$

where f^* is the conjugate function of f, and the symmetric scaling from [59, 50] is implemented. We define *shadow iterates* as

$$\tilde{z} = -\nabla f(s), \quad \tilde{s} = -\nabla f^*(z),$$

with $\tilde{\mu} = \langle \tilde{s}, \tilde{z} \rangle / \nu$. A scaling matrix H is chosen to be the rank-4 Broyden-Fletcher-Goldfarb-Shanno (BFGS) update as in a quasi-Newton method,

$$H = H_{\text{BFGS}} = Z(Z^\top S)^{-1} Z^\top + H_a - H_a S(S^\top H_a S)^{-1} S^\top H_a,$$

where $Z = [z, \tilde{z}], S = [s, \tilde{s}]$ and $H_a = \mu \nabla^2 f(z)$ is an estimate of the Hessian as in [50].

Algorithm 1 shows how to update the scaling matrix H at iteration $k+1$:

Algorithm 1 Update the scaling matrix H

1: **Input:** the scaling matrix H^k at iteration k
2:
3: Update H_1^{k+1} // Zero cone
4: Update H_2^{k+1} // Nonnegative cone
5:
6: `@sync for` $t = 3$ to i // Second-order cones
7: $H_t^{k+1} = \text{\_soc\_update\_H}(H_t^k)$
8: **end**
9:
10: `@sync for` $t = i + 1$ to j // Exponential cones
11: $H_t^{k+1} = \text{\_exp\_update\_H}(H_t^k)$
12: **end**
13:
14: `@sync for` $t = j + 1$ to p // Power cones
15: $H_t^{k+1} = \text{\_pow\_update\_H}(H_t^k)$
16: **end**
17:
18: **return** H^{k+1} // Output the new scaling matrix H^{k+1}

D   Detailed benchmark results

Table 2: Solve times and iteration counts for the large LPs problem set

Problem	vars.	cons.	nnz(A)	nnz(P)	iterations		total time (s)	
					ClarabelGPU	Mosek	ClarabelGPU	Mosek
QAP15	28605	22275	117225	0	22	16	4.72	3.6
BRAZIL3	62488	23968	181026	0	20	15	0.66	1.53
EX10	104968	17680	1197360	0	13	16	5.75	14.7
NEOS_5052403_CYGNET	104004	32868	4964040	0	23	12	7.66	4.86
SUPPORTCASE10	195224	14770	584622	0	35	20	9.6	5.65
SQUARE41	164591	62234	13690857	0	28	12	8.39	28.5
CHROMATICINDEX1024_7	215039	73728	417780	0	9	10	1.8	6.4
NEOS_3025225	231263	69846	9497642	0	28	19	50.4	19.9
RMINE15	443271	42438	964608	0	55	29	41.2	14
PHYSICIANSCHED3_3	417922	79555	1214174	0	452	284	251	174
NEOS_5251015	486996	136971	1955853	0	-	9	-	22.6
GRAPH40_40	566100	102600	1466100	0	11	-	3.36	-
DATT256_LP	535365	262144	2028020	0	10	5	7.52	223
S250R10	557246	273142	1864891	0	103	32	29.3	6.56
S100	743567	364417	2506751	0	92	33	96.1	9.42
SAVSCHED1	877295	328575	2351813	0	28	16	10.9	85.8
WOODLANDS09	958865	382147	3410269	0	-	8	-	344
SCPM1	1005000	500000	7250000	0	27	24	24.4	13.2
BHARAT	2030687	590519	4296196	0	67	-	67.3	-
TPL_TUB_WS1617	2649817	747601	6215769	0	141	63	110	133
FHNW_BINSCHEDULE1	3056177	1141653	10894631	0	49	-	232	-
SUPPORTCASE19	2868909	1429098	7145290	0	64	17	56	12.5
S82	3469140	1690631	10403870	0	-	71	-	235

Table 3: Solve times and iteration counts for the QPs problem set

Problem	vars.	cons.	nnz(A)	nnz(P)	iterations				total time (s)			
					ClarabelGPU	ClarabelRs	Gurobi	Mosek	ClarabelGPU	ClarabelRs	Gurobi	Mosek
PORTFOLIO_OPTIMIZATION_N_5000	5501	5500	1259971	5500	15	15	17	9	0.442	7.9	0.9	2.65
PORTFOLIO_OPTIMIZATION_N_10000	11001	11000	5022139	11000	18	18	19	7	1.67	57.1	4.53	10.7
PORTFOLIO_OPTIMIZATION_N_15000	16501	16500	11282198	16500	18	18	18	8	4.16	165	10.4	28.7
PORTFOLIO_OPTIMIZATION_N_20000	22001	22000	20040608	22000	17	17	18	8	8.06	372	22.3	63.1
HUBER_FITTING_N_5000	22500	27500	4725756	7500	8	8	13	15	4.24	68.7	6.53	10.3
PORTFOLIO_OPTIMIZATION_N_25000	27501	27500	31300056	27500	18	18	20	9	14.7	600	44.2	124
HUBER_FITTING_N_10000	45000	55000	18826645	15000	8	8	11	17	18.7	636	39.7	77
HUBER_FITTING_N_15000	67500	82500	42304462	22500	8	8	11	18	48.6	2.18e+03	110	383
HUBER_FITTING_N_20000	90000	110000	75138584	30000	8	-	14	20	99.2	-	269	1.21e+03
HUBER_FITTING_N_25000	112500	137500	117388752	37500	8	-	11	19	180	-	400	2.31e+03

Table 4: Solve times and iteration counts for the large OPF LP problem set

Problem	vars.	cons.	nnz(A)	nnz(P)	iterations			total time (s)		
					ClarabelGPU	ClarabelRs	Mosek	ClarabelGPU	ClarabelRs	Mosek
CASE10192_EPIGRIDS	92051	27914	180020	697	17	17	35	0.742	0.533	0.764
CASE10192_EPIGRIDS__API	92051	27914	180020	697	17	17	28	0.706	0.526	0.627
CASE10480_GOC	99926	29816	196673	276	20	20	27	0.82	0.712	0.701
CASE10480_GOC__API	99926	29816	196673	276	18	18	31	0.806	0.666	0.75
CASE13659_PEGASE	120495	38218	230046	0	122	107	81	2.21	3.29	2.69
CASE13659_PEGASE__API	120495	38218	230046	0	15	15	59	0.876	0.648	1.82
CASE20758_EPIGRIDS	182928	56275	355508	1881	18	18	32	1.43	1.43	1.28
CASE20758_EPIGRIDS__API	182928	56275	355508	1881	18	18	31	1.4	1.41	1.25
CASE19402_GOC	184959	55077	364846	249	24	24	41	1.65	1.86	1.65
CASE19402_GOC__API	184959	55077	364846	249	25	25	38	1.72	2.03	1.59
CASE19402_GOC__SAD	184959	55077	364846	249	22	-	55	1.61	-	2.09
CASE24464_GOC	210481	63871	408258	348	22	-	26	1.82	-	1.52
CASE24464_GOC__API	210481	63871	408258	348	21	21	25	1.76	2.09	1.41
CASE24464_GOC__SAD	210481	63871	408258	348	24	-	65	1.82	-	2.74
CASE30000_GOC	213698	68919	399262	372	38	38	47	1.99	2.83	1.45
CASE30000_GOC__API	213698	68919	399262	372	30	30	39	1.91	2.54	1.34
CASE30000_GOC__SAD	213698	68919	399262	372	25	25	83	1.65	1.89	2.24
CASE78484_EPIGRIDS	685984	211266	1334253	0	29	29	117	8.31	16.8	68.9
CASE78484_EPIGRIDS__API	685984	211266	1334253	0	26	26	106	7.89	15.2	93.8
CASE78484_EPIGRIDS__SAD	685984	211266	1334253	0	30	30	120	8.37	17.6	68.7

Table 5: Solve times and iteration counts for the large OPF SOCP problem set

Problem	vars.	cons.	nnz(A)	nnz(P)	iterations				total time (s)			
					ClarabelGPU	Mosek*	ClarabelRs	Mosek	ClarabelGPU	Mosek*	ClarabelRs	Mosek
CASE2383WP_K	97600	20393	155950	0	109	-	107	-	1.81	-	3.63	-
CASE2383WP_K__API	97600	20393	155950	0	21	38	23	33	0.928	1.21	1.26	1.89
CASE2383WP_K__SAD	97600	20393	155950	0	99	-	99	-	1.78	-	3.48	-
CASE2312_GOC	98556	20518	159516	0	47	151	48	-	1.35	3.98	2.05	-
CASE2312_GOC__API	98556	20518	159516	0	65	250	65	-	1.41	6.23	2.19	-
CASE2312_GOC__SAD	98556	20518	159516	0	59	191	59	-	1.32	4.98	1.98	-
CASE2737SOP_K	109822	22777	176602	0	70	-	75	-	1.6	-	3.07	-
CASE2737SOP_K__API	109822	22777	176602	0	71	-	73	-	1.63	-	2.99	-
CASE2737SOP_K__SAD	109822	22777	176602	0	72	-	73	-	1.62	-	2.92	-
CASE2736SP_K	110022	22878	176901	0	71	-	69	-	1.63	-	2.88	-
CASE2736SP_K__API	110022	22878	176901	0	57	83	57	-	1.41	2.63	2.65	-
CASE2736SP_K__SAD	110022	22878	176901	0	71	-	72	-	1.53	-	3.67	-
CASE2746WP_K	111106	23320	178556	0	74	-	73	-	1.78	-	3.28	-
CASE2746WP_K__API	111106	23320	178556	0	31	33	32	35	1.18	1.22	1.78	2.36
CASE2746WP_K__SAD	111106	23320	178556	0	74	133	74	-	1.65	4.28	3.19	-
CASE2746WOP_K	111822	23434	179829	0	64	-	64	-	1.57	-	2.8	-
CASE2746WOP_K__API	111822	23434	179829	0	26	33	-	36	1.11	1.18	-	2.21
CASE2746WOP_K__SAD	111822	23434	179829	0	64	-	63	-	1.43	-	2.6	-
CASE3012WP_K	120676	25202	193907	0	94	-	91	-	1.96	-	3.94	-
CASE3012WP_K__API	120676	25202	193907	0	29	53	-	48	1.2	1.98	-	3.1
CASE3012WP_K__SAD	120676	25202	193907	0	89	-	91	-	1.94	-	3.92	-
CASE2848_RTE	122708	25858	197228	0	80	177	80	-	2.15	5.3	4.75	-
CASE2848_RTE__API	122708	25858	197228	0	75	-	75	-	2.28	-	5.25	-
CASE2848_RTE__SAD	122708	25858	197228	0	70	-	70	-	2.39	-	4.36	-
CASE2868_RTE	123912	26164	198869	0	84	225	84	-	2.35	6.49	5.8	-
CASE2868_RTE__API	123912	26164	198869	0	73	258	75	-	2.11	7.53	4.96	-
CASE2868_RTE__SAD	123912	26164	198869	0	79	288	79	-	2.24	8.31	5.24	-
CASE3120SP_K	124354	25856	199827	0	79	-	75	-	1.77	-	3.4	-
CASE3120SP_K__API	124354	25856	199827	0	87	-	92	-	2.02	-	4.14	-
CASE3120SP_K__SAD	124354	25856	199827	0	99	-	100	-	2.03	-	4.4	-
CASE2853_SDET	128886	27445	206896	0	102	269	103	-	2.1	8.51	4.34	-
CASE2853_SDET__API	128886	27445	206896	0	122	309	121	-	2.43	9.57	5.12	-
CASE2853_SDET__SAD	128886	27445	206896	0	101	276	101	-	2.1	8.64	4.33	-
CASE3022_GOC	134780	28060	217434	0	41	174	-	139	1.52	6.16	-	9.67
CASE3022_GOC__API	134780	28060	217434	0	58	202	-	177	1.84	6.88	-	12.9
CASE3022_GOC__SAD	134780	28060	217434	0	42	177	-	142	1.51	5.82	-	9.91
CASE3375WP_K	139126	29112	223999	0	101	-	89	-	2.14	-	4.04	-
CASE3375WP_K__API	139126	29112	223999	0	146	-	151	-	2.81	-	6.84	-
CASE3375WP_K__SAD	139126	29112	223999	0	103	-	92	-	2.17	-	4.25	-
CASE2869_PEGASE	143608	30153	236217	0	53	235	53	-	2.03	8.56	4.44	-
CASE2869_PEGASE__API	143608	30153	236217	0	50	264	50	-	1.88	9.44	4.4	-
CASE2869_PEGASE__SAD	143608	30153	236217	0	48	278	48	-	1.86	9.99	4.08	-
CASE2742_GOC	144110	29856	236467	0	110	119	109	-	3.89	4.9	9.42	-
CASE2742_GOC__API	144110	29856	236467	0	104	127	102	129	3.46	5.25	8.19	10.4
CASE2742_GOC__SAD	144110	29856	236467	0	107	119	106	-	3.83	4.94	9.09	-
CASE4661_SDET	198498	41599	320728	0	140	-	137	-	4.23	-	11.2	-
CASE4661_SDET__API	198498	41599	320728	0	125	-	125	-	3.94	-	10.2	-
CASE4661_SDET__SAD	198498	41599	320728	0	139	-	134	-	4.37	-	10.9	-
CASE3970_GOC	205819	42789	337328	0	140	154	143	-	4.94	9.11	15.6	-
CASE3970_GOC__API	205819	42789	337328	0	109	136	115	-	4.85	8.04	15.1	-
CASE3970_GOC__SAD	205819	42789	337328	0	142	169	144	-	5.07	10.2	16.2	-
CASE4020_GOC	216455	44877	355706	0	181	194	186	-	6.86	12.1	25.1	-
CASE4020_GOC__API	216455	44877	355706	0	101	144	101	-	4.77	9.28	15.4	-
CASE4020_GOC__SAD	216455	44877	355706	0	183	178	185	-	6.71	13.2	24.4	-
CASE4917_GOC	218213	45522	352833	0	52	164	-	147	2.99	8.63	-	16.7
CASE4917_GOC__API	218213	45522	352833	0	62	260	60	-	3.27	13.4	7.7	-
CASE4917_GOC__SAD	218213	45522	352833	0	47	184	-	166	2.91	10.1	-	18.8
CASE4601_GOC	225588	46885	368445	0	179	192	180	-	5.98	11.5	21.8	-
CASE4601_GOC__API	225588	46885	368445	0	103	147	109	162	4.66	8.99	15.6	20.5
CASE4601_GOC__SAD	225588	46885	368445	0	180	191	177	-	5.94	12.2	20	-
CASE4837_GOC	240160	49855	392884	0	139	152	137	-	7.18	10.6	24.3	-
CASE4837_GOC__API	240160	49855	392884	0	122	177	124	-	5.95	12.1	19	-
CASE4837_GOC__SAD	240160	49855	392884	0	136	161	130	-	7.05	11	23	-
CASE4619_GOC	254753	52616	419210	0	145	153	144	-	7.98	11	26.8	-

Table 5: Solve times and iteration counts for the large OPF SOCP problem set

Problem	vars.	cons.	nnz(A)	nnz(P)	iterations				total time (s)			
					ClarabelGPU	Mosek*	ClarabelRs	Mosek	ClarabelGPU	Mosek*	ClarabelRs	Mosek
CASE4619_GOC__API	254753	52616	419210	0	130	161	130	-	6.85	11.9	21.4	-
CASE4619_GOC__SAD	254753	52616	419210	0	151	150	151	-	8.17	11.2	27	-
CASE5658_EPIGRIDS	282180	58620	461724	0	150	181	143	-	9.01	13.8	28.4	-
CASE5658_EPIGRIDS__API	282180	58620	461724	0	164	202	159	-	10.8	15	34.9	-
CASE5658_EPIGRIDS__SAD	282180	58620	461724	0	156	221	158	-	8.82	16.6	29.1	-
CASE6468_RTE	286248	59396	463599	0	104	250	101	-	5.5	16.1	17.7	-
CASE6468_RTE__API	286248	59396	463599	0	88	292	88	-	5.43	18.4	17.2	-
CASE6468_RTE__SAD	286248	59396	463599	0	93	254	99	-	5.2	16.5	17.5	-
CASE6470_RTE	287806	60144	465757	0	122	319	109	-	6.14	20.3	21.9	-
CASE6470_RTE__API	287806	60144	465757	0	86	330	85	-	5.22	21.4	17.2	-
CASE6470_RTE__SAD	287806	60144	465757	0	102	350	99	-	5.49	22.2	17.5	-
CASE6495_RTE	288050	60099	465939	0	112	329	115	-	6.07	21.3	20.1	-
CASE6495_RTE__API	288050	60099	465939	0	102	349	95	-	5.61	22.3	16.6	-
CASE6495_RTE__SAD	288050	60099	465939	0	112	344	108	-	5.64	22.1	18.7	-
CASE6515_RTE	288710	60239	466922	0	97	299	95	-	6.3	19.1	17.7	-
CASE6515_RTE__API	288710	60239	466922	0	81	332	81	-	5.29	21.7	15.8	-
CASE6515_RTE__SAD	288710	60239	466922	0	86	327	84	-	5.24	21.4	15.6	-
CASE7336_EPIGRIDS	358450	74618	585766	0	156	196	154	-	11.6	18	47.8	-
CASE7336_EPIGRIDS__API	358450	74618	585766	0	150	214	152	-	10.5	19.9	43.6	-
CASE7336_EPIGRIDS__SAD	358450	74618	585766	0	155	186	156	-	11.5	17.3	50.2	-
CASE10000_GOC	440997	92799	712177	0	-	86	-	-	-	10.3	-	-
CASE10000_GOC__API	440997	92799	712177	0	62	206	-	-	6.91	21.1	-	-
CASE10000_GOC__SAD	440997	92799	712177	0	90	152	-	-	7.44	16.4	-	-
CASE8387_PEGASE	459046	96351	750588	0	139	-	138	-	9.19	-	30.3	-
CASE8387_PEGASE__API	459046	96351	750588	0	140	-	140	-	10	-	34.3	-
CASE8387_PEGASE__SAD	459046	96351	750588	0	134	-	134	-	9.01	-	29.8	-
CASE9591_GOC	495073	102120	811889	0	226	-	229	-	21	-	85.7	-
CASE9591_GOC__API	495073	102120	811889	0	149	159	149	-	15.1	25.5	58.6	-
CASE9591_GOC__SAD	495073	102120	811889	0	228	200	228	-	21.3	31.5	86.8	-
CASE9241_PEGASE	502110	104741	827419	0	70	-	66	-	6.83	-	20.3	-
CASE9241_PEGASE__API	502110	104741	827419	0	67	-	70	-	6.61	-	20.6	-
CASE9241_PEGASE__SAD	502110	104741	827419	0	70	389	68	-	6.73	58.4	20.5	-
CASE10192_EPIGRIDS	529695	109758	866501	0	96	113	96	-	10.6	17	36	-
CASE10192_EPIGRIDS__API	529695	109758	866501	0	91	113	91	-	10.4	17.4	35.3	-
CASE10192_EPIGRIDS__SAD	529695	109758	866501	0	109	-	107	-	11.9	-	41.5	-
CASE10480_GOC	573754	118760	943278	0	164	-	162	-	18.2	-	76.2	-
CASE10480_GOC__API	573754	118760	943278	0	136	217	-	-	16.2	40.5	-	-
CASE10480_GOC__SAD	573754	118760	943278	0	164	242	164	-	18.3	43.6	77.4	-
CASE13659_PEGASE	662910	140961	1081042	0	83	-	82	-	11.8	-	42.9	-
CASE13659_PEGASE__API	662910	140961	1081042	0	78	-	78	-	11.4	-	39.9	-
CASE13659_PEGASE__SAD	662910	140961	1081042	0	78	329	-	-	11	60	-	-
CASE20758_EPIGRIDS	1048059	218151	1709288	0	122	-	123	-	27.5	-	99.1	-
CASE20758_EPIGRIDS__API	1048059	218151	1709288	0	154	-	155	-	39.9	-	150	-
CASE20758_EPIGRIDS__SAD	1048059	218151	1709288	0	137	-	135	-	30.3	-	111	-
CASE19402_GOC	1064421	219911	1753874	0	256	-	251	-	45.9	-	221	-
CASE19402_GOC__API	1064421	219911	1753874	0	212	-	216	-	39.3	-	193	-
CASE19402_GOC__SAD	1064421	219911	1753874	0	257	-	258	-	46	-	232	-
CASE30000_GOC__API	1195834	249462	1921390	0	150	-	-	-	24.6	-	-	-
CASE24464_GOC	1202964	248644	1983519	0	230	-	-	-	39.4	-	-	-
CASE24464_GOC__API	1202964	248644	1983519	0	213	-	214	-	33.4	-	149	-
CASE24464_GOC__SAD	1202964	248644	1983519	0	-	-	224	-	-	-	169	-
CASE78484_EPIGRIDS	3904758	811998	6389202	0	401	-	-	-	260	-	-	-
CASE78484_EPIGRIDS__API	3904758	811998	6389202	0	400	-	401	-	320	-	1.65e+03	-
CASE78484_EPIGRIDS__SAD	3904758	811998	6389202	0	408	-	-	-	263	-	-	-

Table 6: Solve times and iteration counts for the exponential cone programming problem set

Problem	vars.	cons.	nnz(A)	nnz(P)	iterations				total time (s)			
					ClarabelGPU	Mosek*	ClarabelRs	Mosek	ClarabelGPU	Mosek*	ClarabelRs	Mosek
ENTROPY_N_2000	11001	4000	258917	0	23	15	21	10	0.548	0.506	3.05	0.565
ENTROPY_N_4000	22001	8000	1019263	0	24	16	20	9	1.87	2.29	14.2	2.22
ENTROPY_N_6000	33001	12000	2278947	0	26	16	22	9	3.56	6.09	45.1	5.85
ENTROPY_N_8000	44001	16000	4037959	0	27	16	22	10	6.2	12.3	107	12.6
ENTROPY_N_10000	55001	20000	6297376	0	27	16	22	10	9.96	22.6	235	40.1
LOG_REG_N_2000	94002	34001	1320255	0	57	38	121	37	4.13	3.96	107	5.43
LOG_REG_N_4000	188002	68001	5205338	0	68	47	-	47	17.4	16.4	-	21.6
LOG_REG_N_6000	282002	102001	11529263	0	72	49	-	51	40.9	67.7	-	59.8
LOG_REG_N_8000	376002	136001	20589037	0	73	48	-	50	77.5	135	-	105
LOG_REG_N_10000	470002	170001	31904473	0	84	54	-	58	150	372	-	336