Noname manuscript No.
(will be inserted by the editor)

QOCO-GPU: A Quadratic Objective Conic Optimizer with GPU Acceleration

Govind M. Chari · Behçet Açıkmeşe

Received: date / Accepted: date

Abstract We present a GPU-accelerated backend for QOCO, a C-based solver for quadratic objective second-order cone programs (SOCPs) based on a primal-dual interior point method. Our backend uses NVIDIA's cuDSS library to perform a direct sparse $LDL^\top$ factorization of the KKT system at each iteration. We also develop custom CUDA kernels for cone operations and show that parallelizing these operations is essential for achieving peak performance. Additionally, we refactor QOCO to introduce a modular backend abstraction that decouples solver logic from the underlying linear algebra implementations, allowing the existing CPU and new GPU backend to share a unified codebase. This GPU backend is accessible through a direct Python interface and through CVXPY, allowing for easy use. Numerical experiments on a range of large-scale quadratic programs and SOCPs with tens to hundreds of millions of nonzero elements in the KKT matrix, demonstrate speedups of up to 50-70 times over the CPU implementation.

1 Introduction

We consider the following quadratic objective second-order cone program (SOCP)

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \frac{1}{2}x^\top Px + c^\top x \\ \text{subject to} & Gx \preceq_{\mathcal{K}} h \\ & Ax = b, \end{array} \tag{1}$$

with optimization variable $x \in \mathbb{R}^n$. The objective is defined by $P \succeq 0$ and $c \in \mathbb{R}^n$. The equality and conic constraints are defined by $A \in \mathbb{R}^{p\times n}$, $G \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^p$, and $h \in \mathbb{R}^m$. The conic inequality $Gx \preceq_{\mathcal{K}} h$ denotes $h - Gx \in \mathcal{K}$, where $\mathcal{K}$ is the Cartesian product

$$\mathcal{K} := \mathcal{C}_1 \times \mathcal{C}_2 \times \cdots \times \mathcal{C}_K,$$

and $\mathcal{C}_k$ is either the non-negative orthant or a second-order cone. We assume that Problem (1) is **feasible** and has a **bounded** optimal objective.

G. Chari
E-mail: gchari@uw.edu

B. Açıkmeşe
E-mail: behcet@uw.edu

In this work, we focus on solving *large-scale* instances of Problem (1) with tens to hundreds of millions of nonzero elements in the KKT matrix using the QOCO solver [4], a C-based solver for quadratic objective SOCPs that implements a primal-dual interior point method [20].

1.1 Related work

While GPUs have dramatically accelerated machine learning workloads, leveraging them in optimization algorithms has historically been challenging. The main difficulty is that sparse linear algebra, especially sparse matrix factorizations, is significantly more challenging to accelerate than dense linear algebra [15]. As a result, most GPU accelerated optimization methods have relied on algorithms that avoid direct sparse factorizations [12, 16, 18].

Two main approaches have been considered: first-order optimization methods such as PDLP [9], which require only matrix-vector multiplications that are straightforward to accelerate on a GPU, and using indirect methods such as the preconditioned conjugate gradient method to solve linear systems rather than direct sparse factorizations [12, 16, 18]. However, the first-order methods are highly sensitive to problem conditioning and can struggle to converge to high accuracy solutions.

Interior-point methods (IPMs), however, are more robust and can achieve high accuracy, but have required using indirect methods to solve the linear system for GPU implementation [18]. These indirect methods require only matrix-vector multiplications, which allow them to be accelerated on a GPU, but the iteration count of these methods depends on the condition number of the linear system, which can be extremely large, leading to long per-iteration runtime [11].

In late 2023, NVIDIA released cuDSS, a high-performance direct sparse solver for GPUs, which could be integrated into optimization algorithms. Since its release, cuDSS has been integrated as a linear system solver in the conic solvers CUCLARABEL [5] and MOREAU [1], as well as in the nonlinear programming solver MADNLP [17].

1.2 Contribution

We present three main contributions. First, we develop a CUDA backend for the QOCO solver that uses cuDSS to perform a direct sparse $LDL^\top$ factorization of the KKT system [1]. This enables GPU acceleration while avoiding indirect solvers, whose performance is sensitive to conditioning. Second, we implement custom CUDA kernels for parallel cone operations required by the interior-point method and show that these parallelized operations are necessary to achieve good performance on the GPU. Third, we refactor QOCO to use a modular backend abstraction that decouples solver logic from the underlying linear algebra implementation, allowing both CPU and GPU backends to share a unified codebase.

We present benchmarks that demonstrate a $50 - 70\times$ speedup over the CPU version of QOCO on some large problem instances. This GPU-accelerated version of QOCO can be called directly through a Python interface or through CVXPY [6], making it easy to use.

Compared to CUCLARABEL [5], an existing GPU-accelerated IPM for conic optimization, our C-based implementation avoids the just-in-time compilation overhead of Julia, and provides a unified interface for CPU and GPU backends within Python and CVXPY scripts, making prototyping and comparing the backends easier.

[1] Our implementation can be found at https://github.com/qoco-org/qoco

```python
# Solve with CPU backend with Python interface.
solver_cpu = qoco.QOCO(algebra="builtin")
solver_cpu.setup(n, m, p, P, c, A, b, G, h, l, nsoc, q)
result_cpu = solver_cpu.solve()

# Solve with GPU backend with Python interface.
solver_gpu = qoco.QOCO(algebra="cuda")
solver_gpu.setup(n, m, p, P, c, A, b, G, h, l, nsoc, q)
result_gpu = solver_gpu.solve()

# Solve with CPU backend in CVXPY.
problem.solve(solver="QOCO", algebra="builtin")

# Solve with GPU backend in CVXPY.
problem.solve(solver="QOCO", algebra="cuda")
```

Listing 1: Calling CPU and GPU backends from Python interface and CVXPY

2 Implementation

Each iteration of QOCO's primal-dual IPM requires solving two linear systems with the same coefficient matrix (the KKT matrix) and performing various cone operations on $\mathcal{K}$. The numerical factorization of the KKT matrix can be accelerated with cuDSS and the cone operations are parallelizable because $\mathcal{K}$ is the Cartesian product of many cones, $\mathcal{C}_k$.

2.1 Modular backend

To support both CPU and GPU backends without duplicating code, we refactor the main IPM loop of QOCO to be backend-agnostic. All solver code shared between the two backends operates on abstract data types rather than backend-specific data structures.

Specifically, we introduce the types `QOCOMatrix` and `QOCOVector`, which represent matrices and vectors used in QOCO. Each backend provides its own implementation of these types. The CPU implementation stores data in CPU memory, whereas the GPU implementation stores pointers to both CPU and GPU memory.

The linear system solver is abstracted through a `Linsys` interface that defines `initialize`, `factor`, `solve`, and `update` functions. Each backend provides its own implementation of this interface. The CPU backend uses the QDLDL linear system solver [19], while the GPU backend uses cuDSS.

Cone operations such as Jordan products, projecting onto cone $\mathcal{K}$, and computing the Nesterov-Todd scalings are implemented separately for the CPU and GPU backends so that these operations can be parallelized on the GPU.

At compile time, a `CMake` flag selects the desired backend and includes the corresponding implementations of `QOCOMatrix`, `QOCOVector`, `Linsys`, and the cone operations.

The Python interface exposes both backends with `pybind11`, allowing users to select the CPU or GPU implementation at runtime. This is illustrated in Listing 1.

2.2 Linear system solver

The primary operation accelerated by the GPU backend is the $LDL^\top$ factorization of the KKT system in Equation (2), where W_k is the Nesterov-Todd scaling matrix. In the CPU version of QOCO, this

factorization dominates the runtime and its cost grows rapidly with problem size. Therefore, we accelerate this factorization on the GPU using cuDSS.

$$\begin{bmatrix} P & A^\top & G^\top \\ A & 0 & 0 \\ G & 0 & -W_k^\top W_k \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} \tag{2}$$

cuDSS has three phases: analysis, factor, and solve. The analysis phase is executed once and computes a fill-reducing reordering of the KKT matrix, applies this permutation to the KKT matrix, and computes a symbolic factorization that determines the sparsity pattern of the factors. This phase is the most expensive part of the solve, and for large optimization problems, it can take longer than the IPM iterations themselves. This is because cuDSS computes the reordering on the CPU, since the graph algorithms used for this step are difficult to accelerate on the GPU. The factor phase is executed once per IPM iteration and computes the $LDL^\top$ factor of the KKT matrix. The solve phase is executed twice per IPM iteration and performs triangular solves using the factored system.

2.3 GPU performance considerations

Simply using cuDSS as a linear system solver is not sufficient for developing a high-performance GPU backend due to certain implementation and hardware-level challenges that have to be addressed. Since data transfer between the CPU and GPU is significantly slower than memory access within the GPU, it is important to minimize CPU-GPU data transfers. To address this, during the setup phase of the solver, all internal QOCO data structures are allocated on the CPU and then copied to the GPU. During the solve phase, all computation is performed on the GPU. After convergence, the optimal solution is copied back to the CPU.

Another issue is that all cone operations on $\mathcal{K}$ must also be executed on the GPU, which can be extremely slow if parallelism is not exploited. As discussed earlier, operations on $\mathcal{K}$ decompose into independent operations on each cone $\mathcal{C}_k$. Therefore, these operations are trivially parallelizable.

We have observed in our implementation that this parallelism should be exploited. In earlier versions of our GPU backend, the cone operations were executed serially on a single GPU core. Although this produced a correct algorithm, it was up to an order of magnitude slower, especially on problems with many second-order cones, and the time spent on cone operations exceeded the runtime of the matrix factorization, which should be the main computational bottleneck of the solve phase. The reason is that GPUs are designed for massively parallel workloads, while CPUs outperform GPUs on sequential computations. Therefore, parallelizing cone operations is not merely a performance optimization, but a requirement for an efficient GPU implementation. To do this, we implemented custom CUDA kernels for the cone operations. Each kernel implements the necessary operations for a single cone $\mathcal{C}_k$, and we map cones to GPU threads by launching a grid of thread blocks whose total number of threads matches the number of cones.

3 Numerical results

Here, we benchmark QOCO-GPU against the CPU implementation of QOCO, as well as CUCLARABEL, MOSEK, and GUROBI [2]. For all solvers, we use their default settings but set the tolerances $\epsilon_{\text{abs}} = \epsilon_{\text{rel}} = 10^{-7}$. All results were generated on a computer with an Intel i9-14900k processor, 96 GB of RAM, and an NVIDIA GeForce RTX 5090 with 32 GB of VRAM.

[2] Our benchmarks are publicly available at https://github.com/qoco-org/qoco-gpu-benchmarks

In this section, problem size is defined as the total number of nonzero elements in A, G, and the upper half of P. We evaluate the solvers on a set of benchmark problems which include three quadratic programs (QPs): Huber regression, single-period portfolio optimization [10], and multi-period portfolio optimization [2], and two second-order cone programs (SOCPs): group lasso regression [22] and total variation denoising [3]. The runtime of each solver, which includes setup and solve time, is limited to an hour. To compare the performance of solvers, we use performance profiles [7] and the shifted geometric mean. Details on how these metrics are computed can be found in [4,5,8,19].

3.1 Benchmark problems

We consider the following QPs: single period portfolio optimization, multiperiod portfolio optimization, and Huber regression.

The single period portfolio optimization problem is

$$\begin{array}{ll} \underset{x,y}{\text{minimize}} & x^\top D x + y^\top y - \gamma^{-1}\mu^\top x \\ \text{subject to} & y = F^\top x \\ & x \in \Delta_n \end{array}$$

where $F \in \mathbb{R}^{k\times 100k}$ for k: $\{2, 5, 10, 15, 25, 50, 75, 125\}$ and Δ_n is the n-dimensional unit simplex.

The multiperiod portfolio optimization problem is

$$\begin{array}{ll} \underset{w_t, y_t}{\min} & \displaystyle\sum_{t=1}^{T}\left(w_t^\top D w_t + \|y_t\|_2^2 - \frac{1}{\gamma}\mu_t^\top w_t + \|w_t - w_{t-1}\|_2^2\right) \\ \text{s.t.} & w_0 = \bar{w}_0 \\ & \mathbf{1}^\top w_t = 1, \qquad t = 1, \ldots, T \\ & y_t = F^\top w_t, \qquad t = 1, \ldots, T \\ & 0 \le y_t \le 0.01, \qquad t = 1, \ldots, T \\ & \|w_t\|_1 \le L_{\max}, \qquad t = 1, \ldots, T, \end{array}$$

where $F \in \mathbb{R}^{5000\times 50}$ is the factor loading matrix, and we solve for time horizons T: $\{2, 4, 6, 8, 10, 15, 20, 25, 30, 35\}$.

The Huber regression problem is given by

$$\underset{x}{\text{minimize}} \quad \sum_{i=1}^{m} \phi(a_i^\top x - b_i),$$

where $a_i^\top$ denotes the i^{th} row of A and $\phi : \mathbb{R} \to \mathbb{R}$ is the Huber loss. We take $A \in \mathbb{R}^{10N\times N}$ for $N \in \{50, 200, 500, 1000, 2000, 4000, 6000, 10000\}$.

We consider the following SOCPs: group lasso regression and total variation denoising.

The group lasso regression problem is

$$\underset{x}{\text{minimize}} \quad \|Ax - b\|_2^2 + \lambda \sum_{i=1}^{N} \|x^{(i)}\|_2,$$

where $x = [x^{(1)}, x^{(2)}, \ldots, x^{(N)}]$ represents a partitioning of the regression variables into groups with each $x^{(i)}$ corresponding to one group. We choose $A \in \mathbb{R}^{250N\times 10N}$ and solve for N: $\{5, 20, 50, 100, 150, 300, 450, 750\}$.

The total variation denoising problem is

$$\min_{U} \quad \mathrm{TV}(U) + \frac{\lambda}{2}\|U - Y\|_F^2,$$

where $\|\cdot\|_F$ is the Frobenius norm, TV is the total-variation operator and Y is the corrupted image. We test with the following images from the `sk-image` collection [21]: `brick`, `camera`, `grass`, `chelsea`, `coffee`, `astronaut`, `immunohistochemistry`, and `logo`.

3.2 Benchmark results

Figure 1 shows the performance profiles and shifted geometric mean across benchmark problems and Table 1 reports the total runtime results, including setup and solve times for each solver, where the cell corresponding to the fastest solver for each problem instance is highlighted. Table 1 also includes percentage of the total runtime spent in setup for QOCO-GPU when the solver is called directly through its Python interface, as this information is not available through CVXPY. Missing entries correspond to runs that exceed the one hour time limit or did not converge. For the total variation denoising problems, GUROBI failed to converge to the desired accuracy.

For smaller problems, the CPU version of QOCO outperforms QOCO-GPU, but once the KKT matrix contains around 10^5 nonzeros, the GPU version becomes faster. For the largest problems we observe speedups of up to 70× over QOCO. As problems get larger, around 10^5 to 10^6 nonzeros in the KKT matrix, we also observe that both GPU solvers (QOCO-GPU and CUCLARABEL) outperform all CPU solvers (QOCO, MOSEK, and GUROBI). QOCO-GPU and CUCLARABEL exhibit similar performance on most problems, but for a few problems, such as the largest Huber regression problem and largest group lasso regression problem, QOCO-GPU is approximately $2-3\times$ faster than CUCLARABEL.

Overall, QOCO-GPU has the lowest shifted geometric mean, followed by CUCLARABEL, MOSEK, QOCO, and GUROBI. It is expected that QOCO is amongst the slowest solvers on these large problems, since it uses a single-threaded matrix factorization, whereas MOSEK and GUROBI use multithreaded factorizations.

Finally, for larger problems, up to 75% of QOCO-GPU's runtime is spent in the setup phase, where the dominant cost is the reordering step in cuDSS's analysis phase. This bottleneck has also been observed in [13] and [14]. However, the analysis phase only needs to be performed once. If the problem data changes, the reordering can be reused as long as the sparsity pattern of the problem remains fixed. This makes QOCO-GPU well suited for large parameteric optimization problems, where the initial cost of the analysis phase can be amortized over multiple solves.

Table 1: **Runtime in seconds for benchmark problems (QOCO-GPU shows setup time percentage in parentheses)**

Problem	Size	QOCO-GPU	QOCO	CuClarabel	Mosek	Gurobi
huber_50	5500	0.057 (17%)	0.003	0.036	0.011	0.027
huber_200	52000	0.084 (40%)	0.031	0.069	0.097	0.069
huber_500	280000	0.165 (56%)	0.280	0.152	0.605	0.663
huber_1000	1060000	0.416 (69%)	1.868	0.431	4.153	5.278
huber_2000	4120000	1.534 (78%)	18.510	1.650	36.219	20.051
huber_4000	16240000	5.885 (79%)	159.576	6.435	331.281	89.658
huber_6000	36360000	13.167 (78%)	478.210	25.557	1390.174	252.042
huber_10000	100600000	39.816 (76%)	1459.514	107.326	-	439.726
portfolio_10	8020	0.078 (11%)	0.003	0.046	0.008	0.003
portfolio_50	140100	0.100 (36%)	0.063	0.115	0.064	0.050
portfolio_100	530200	0.159 (51%)	0.350	0.181	0.237	0.191

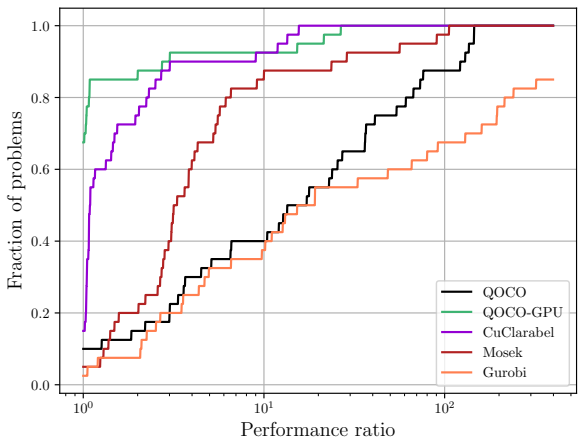
Continued on next page

Problem	Size	QOCO-GPU	QOCO	CuClarabel	Mosek	Gurobi
portfolio_200	2060400	0.393 (64%)	5.282	0.427	1.015	0.813
portfolio_500	12651000	2.206 (76%)	79.760	2.560	6.763	5.564
portfolio_900	40771800	7.501 (71%)	406.031	7.738	20.330	26.248
portfolio_1300	84892600	17.128 (67%)	1149.773	18.409	47.382	61.083
portfolio_1800	162543600	34.458 (65%)	2618.632	37.001	108.825	170.539
multiperiod_portfolio_2	385600	0.247	0.891	0.244	0.302	0.511
multiperiod_portfolio_5	956500	0.568	3.742	0.589	1.148	1.516
multiperiod_portfolio_10	1908000	1.209	15.418	1.266	3.215	5.662
multiperiod_portfolio_15	2859500	1.955	33.728	2.097	7.102	12.838
multiperiod_portfolio_25	4762500	3.601	85.747	3.776	14.854	35.032
multiperiod_portfolio_50	9520000	8.427	346.344	9.208	76.905	277.693
multiperiod_portfolio_75	14277500	14.317	871.908	15.257	411.356	693.854
multiperiod_portfolio_125	23792500	43.295	2952.094	40.338	229.780	3221.442
group_lasso_5	8805	0.055 (19%)	0.004	0.043	0.006	0.069
group_lasso_20	110220	0.097 (41%)	0.107	0.081	0.032	2.944
group_lasso_50	650550	0.234 (69%)	1.104	0.226	0.216	34.604
group_lasso_100	2551100	0.716 (80%)	4.666	6.445	0.986	229.384
group_lasso_150	5701650	1.543 (83%)	16.067	3.134	2.171	877.229
group_lasso_300	22653300	6.551 (82%)	112.120	6.309	13.908	-
group_lasso_450	50854950	15.627 (79%)	342.902	14.973	49.441	-
group_lasso_750	141008250	54.863 (74%)	1400.521	165.127	336.563	-
tv_denoising_camera	2092037	6.110	18.302	6.229	23.394	-
tv_denoising_grass	2092037	6.064	21.103	5.880	38.575	-
tv_denoising_brick	2092037	6.141	18.471	6.381	18.190	-
tv_denoising_chelsea	2966850	8.706	1058.975	13.401	26.812	-
tv_denoising_coffee	5267013	16.804	2287.905	24.972	64.232	-
tv_denoising_astronaut	5753869	18.558	2702.704	27.056	99.934	-
tv_denoising_immunohistochemistry	5753869	18.457	2681.049	26.344	96.628	-
tv_denoising_logo	7233017	27.720	-	36.955	153.639	-

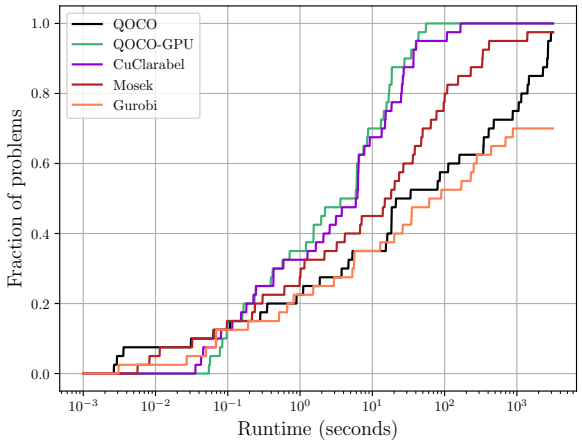
4 Limitations

Although the GPU backend provides substantial speedups on large problems, there are a few limitations. The sparse direct factorization can lead to significant fill-in, which increases memory usage and can lead to out-of-memory errors on the GPU for extremely large problems. This limitation is shared with any solvers, such as CUCLARABEL, that rely on sparse direct methods. Additionally, the GPU backend only offers a speedup for sufficiently large problems. For smaller problems, the overhead of the GPU kernel launches and the CPU-GPU memory transfers can outweigh the speedups offered by the GPU. Finally, the runtime bottleneck of this backend on large problems is the runtime of cuDSS's analysis phase.

Acknowledgements This research was supported by ONR grants N000142512231 and N00014-25-1-2319. We would like to thank Danylo Malyuta and Abhinav Kamath for their review of this paper.



(a) Relative performance profile



(b) Absolute performance profile

	QOCO-GPU	QOCO	CuClarabel	Mosek	Gurobi
Shifted GM	**1.00**	11.86	1.24	3.79	19.22
Failure Rate (%)	**0.0**	2.5	**0.0**	2.5	27.5

(c) Shifted geometric means and failure rates

Fig. 1: Performance profiles for benchmark problems

References

1. Barratt, S., Nobel, P., Diamond, S.: Moreau: GPU-native differentiable optimization (2026). URL https://moreau.so
2. Boyd, S., Diamond, S., Koh, K., Nystrup, P., Busseti, E., Kahn, R.N., Speth, J.: Multi-period trading via convex optimization. Foundations and Trends in Optimization **3**(1), 1–76 (2017)
3. Chambolle, A., Caselles, V., Cremers, D., Novaga, M., Pock, T., et al.: An introduction to total variation for image analysis. Theoretical foundations and numerical methods for sparse recovery **9**(263-340), 227 (2010)
4. Chari, G.M., Açıkmeşe, B.: Qoco: a quadratic objective conic optimizer with custom solver generation. Mathematical Programming Computation (2026). DOI 10.1007/s12532-026-00311-8. URL http://dx.doi.org/10.1007/s12532-026-00311-8
5. Chen, Y., Tse, D., Nobel, P., Goulart, P., Boyd, S.: CuClarabel: GPU acceleration for a conic optimization solver. arXiv preprint arXiv:2412.19027 (2024)
6. Diamond, S., Boyd, S.: CVXPY: A Python-embedded modeling language for convex optimization. Journal of Machine Learning Research **17**(83), 1–5 (2016)
7. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Mathematical Programming **91**(2), 201–213 (2002). DOI 10.1007/s101070100263. URL http://dx.doi.org/10.1007/s101070100263
8. Goulart, P.J., Chen, Y.: Clarabel: An interior-point solver for conic programs with quadratic objectives (2024). URL https://arxiv.org/abs/2405.12762
9. Lu, H., Yang, J.: cuPDLP.jl: A GPU implementation of restarted primal-dual hybrid gradient for linear programming in Julia. Operations Research **73**(6), 3440–3452 (2025)
10. Markowitz, H.: Portfolio selection. J. Finance **7**(1), 77 (1952)
11. Nocedal, J., Wright, S.J.: Numerical optimization. Springer (2006)
12. O'Donoghue, B.: Operator splitting for a homogeneous embedding of the linear complementarity problem. SIAM Journal on Optimization **31**, 1999–2023 (2021)
13. Pacaud, F., Shin, S.: GPU-accelerated dynamic nonlinear optimization with ExaModels and MadNLP. In: Conference on Decision and Control (2024)
14. Pacaud, F., Shin, S., Montoison, A., Schanen, M., Anitescu, M.: Condensed-space methods for nonlinear programming on GPUs. arXiv preprint arXiv:2405.14236 (2024)

15. Rennich, S.C., Stosic, D., Davis, T.A.: Accelerating sparse cholesky factorization on GPUs. Parallel Computing **59**, 140–150 (2016). DOI https://doi.org/10.1016/j.parco.2016.06.004. URL https://www.sciencedirect.com/science/article/pii/S016781911630059X. Theory and Practice of Irregular Applications
16. Schubiger, M., Banjac, G., Lygeros, J.: GPU acceleration of ADMM for large-scale quadratic programming. Journal of Parallel and Distributed Computing **144**, 55–67 (2020)
17. Shin, S., Anitescu, M., Pacaud, F.: Accelerating optimal power flow with GPUs: SIMD abstraction of nonlinear programs and condensed-space interior-point methods. Electric Power Systems Research **236**, 110651 (2024)
18. Smith, E., Gondzio, J., Hall, J.: GPU acceleration of the matrix-free interior point method. In: International Conference on Parallel Processing and Applied Mathematics, pp. 681–689. Springer (2011)
19. Stellato, B., Banjac, G., Goulart, P., Bemporad, A., Boyd, S.: OSQP: an operator splitting solver for quadratic programs. Mathematical Programming Computation **12**(4), 637–672 (2020). DOI 10.1007/s12532-020-00179-2. URL https://doi.org/10.1007/s12532-020-00179-2
20. Vandenberghe, L.: The CVXOPT linear and quadratic cone program solvers. Online: http://cvxopt. org/documentation/coneprog. pdf (2010)
21. van der Walt, S., Schönberger, J.L., Nunez-Iglesias, J., Boulogne, F., Warner, J.D., Yager, N., Gouillart, E., Yu, T., the scikit-image contributors: scikit-image: image processing in Python. PeerJ **2**, e453 (2014). DOI 10.7717/peerj.453. URL https://doi.org/10.7717/peerj.453
22. Yuan, M., Lin, Y.: Model selection and estimation in regression with grouped variables. Journal of the Royal Statistical Society Series B **68**, 49–67 (2006). DOI 10.1111/j.1467-9868.2005.00532.x