IMPERIAL

PRINCETON UNIVERSITY

CENTER FOR STATISTICS AND MACHINE LEARNING

Recent Advances in the OSQP Solver

Ian McInerney – Imperial College London
Bartolomeo Stellato – Princeton University, ORFE
Vineet Bansal – Princeton University, CSML
Amit Solomon – Princeton University

2025 INFORMS Annual Meeting
28/10/2025

Main contributors

Ian McInerney

Vineet Bansal

Bartolomeo Stellato

Paul Goulart

Goran Banjac

Michel Schubiger

Agenda

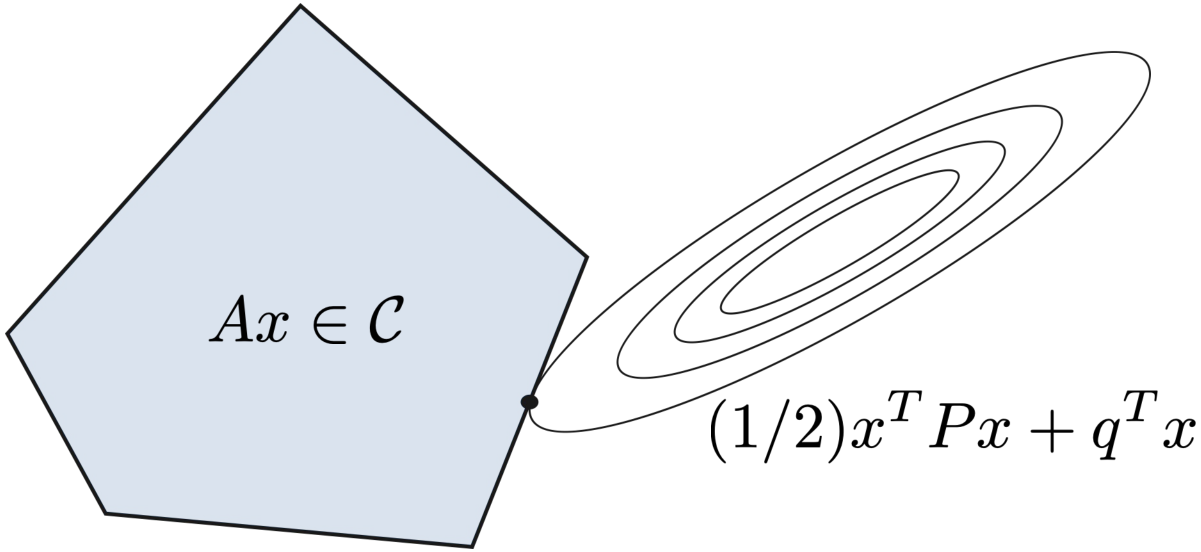
- The OSQP Solver
- New features:
  - Duality gap termination criteria
  - cuDSS integration
- The future

The OSQP Solver

Quadratic Programming Problem

$$\begin{array}{ll} \text{minimize} & (1/2)x^T P x + q^T x \\ \text{subject to} & Ax \in \mathcal{C} \end{array}$$

Quadratic program: $\mathcal{C} = [l, u]$



The OSQP Algorithm

Problem

$$\begin{array}{ll} \text{minimize} & (1/2)x^T P x + q^T x \\ \text{subject to} & l \leq Ax \leq u \end{array}$$

Algorithm

Linear system solve

$$(x^{k+1}, \nu^{k+1}) \leftarrow \text{solve} \begin{bmatrix} P + \sigma I & A^T \\ A & -\frac{1}{\rho} I \end{bmatrix} \begin{bmatrix} x^{k+1} \\ \nu^{k+1} \end{bmatrix} = \begin{bmatrix} \sigma x^k - q \\ z^k - \frac{1}{\rho} y^k \end{bmatrix}$$

Easy operations

$$\tilde{z}^{k+1} \leftarrow z^k + (\nu^{k+1} - y^k)/\rho$$

$$z^{k+1} \leftarrow \Pi\left(\tilde{z}^{k+1} + y^k/\rho\right)$$

$$y^{k+1} \leftarrow y^k + \rho\left(\tilde{z}^{k+1} - z^{k+1}\right)$$

OSQP Solver Features

- Infeasibility detection
- Solution Polishing
- Matrix & vector updates
- Modular linear algebra (CUDA & MKL)
- Embedded (library-free) mode
- Solver instance code generation
- High-level interfaces (JuMP, CVXPY, MATLAB)
- Open source – Apache 2.0 license

Using MKL & CUDA from Python

One-line import change

```python
# Import OSQP from a specific algebra backend module
from osqp.mkl import OSQP as OSQP_mkl
from osqp.cuda import OSQP as OSQP_cuda

prob_mkl = OSQP_mkl()
prob_cuda = OSQP_cuda()

# Setup workspace and change alpha parameter
prob_mkl.setup(P, q, A, l, u, alpha=1.0)

# Solve problem
res = prob_mkl.solve()
```

Setting in object constructor

```python
# Create an OSQP object with a specific algebra backend
if osqp.algebra_available('cuda'):
    # 'builtin' (default), 'mkl', or 'cuda'
    prob = osqp.OSQP(algebra='cuda')
else:
    prob = osqp.OSQP()

# Setup workspace and change alpha parameter
prob.setup(P, q, A, l, u, alpha=1.0)

# Solve problem
res = prob.solve()

...

# Solve with OSQP cuda on CVXPY
import cvxpy as cp

problm = cp.Problem(...)
problem.solve(solver=OSQP, algebra="cuda")
```

It works with CVXPY

New features

Duality Gap Termination

Three important residuals

Primal Residual

Measure of primal feasibility

$$r_{\text{prim}} = Ax - z$$

Dual Residual

Measure of dual feasibility

$$r_{\text{dual}} = Px + q + A^{\mathrm{T}} y$$

Duality Gap

Distance between primal and dual objectives

$$r_{\text{gap}} = x^{\mathrm{T}} P x + q^{\mathrm{T}} x + u^{\mathrm{T}} y_{+} + l^{\mathrm{T}} y_{-}$$

Duality gap termination - Accuracy

Residual-only termination failed to minimize duality gap in 27.8% of benchmark problems

Equality-constrained QP	3.0%
Huber	73.5%
Lasso	0.0%
Portfolio Optimization	100%
Random QP	9%
MPC Control Prob.	5.5%
SVM	0.0%

Percent of problems failing to reduce duality gap to below 1e-3 tolerance

Duality gap termination

- Implemented a duality gap termination criteria
- Iterate until duality gap is less than the tolerance:

$$|r_{\text{gap}}| \leq \epsilon_{\text{gap}} = \epsilon_{\text{abs}} + \epsilon_{\text{rel}} \max\{|x^{\mathrm{T}} P x|, ||q^{\mathrm{T}} x||, |S_{\mathcal{C}}(y)|\}.$$

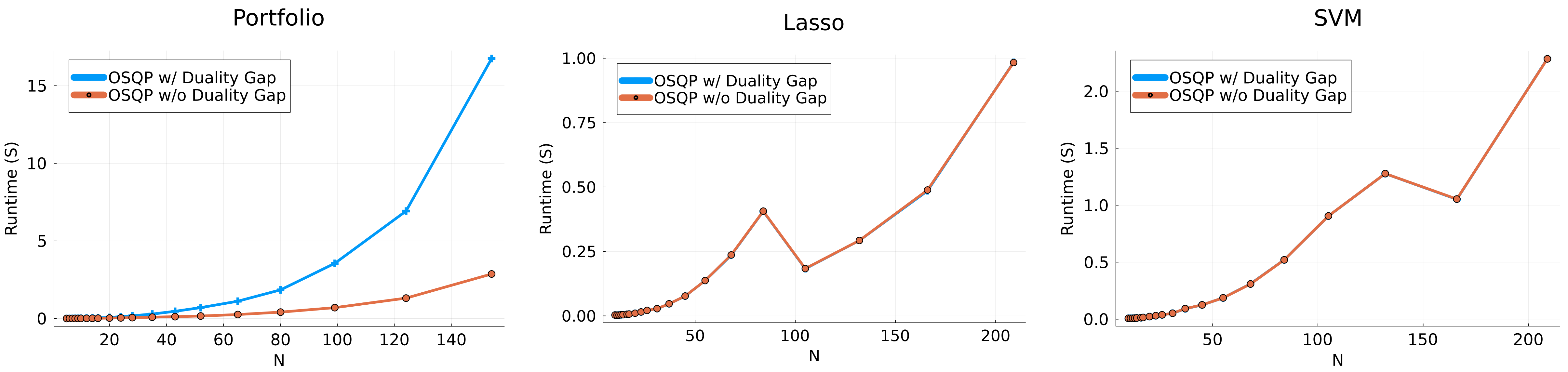
Duality gap termination - Accuracy

With all 3 residuals checked, all benchmark problems converged to below the duality gap tolerance

	OSQP <1.0	**OSQP 1.0**
Equality-constrained QP	3.0%	0.0%
Huber	73.5%	0.0%
Lasso	0.0%	0.0%
Portfolio Optimization	100%	0.0%
Random QP	9%	0.0%
MPC Control Prob.	5.5%	0.0%
SVM	0.0%	0.0%

Percent of problems failing to reduce duality gap to below 1e-3 tolerance

Duality gap termination - Performance



Solved to low accuracy: 1e-3

CPU: Intel Xeon W-2255 @ 3.7GHz

New features

cuDSS Integration

Solving the linear system

Direct method

Quasi-definite matrix

$$\begin{bmatrix} P+\sigma I & A^T \\ A & -\frac{1}{\rho} I \end{bmatrix} \begin{bmatrix} x \\ \nu \end{bmatrix} = \begin{bmatrix} \sigma x^k - q \\ z^k - \frac{1}{\rho} y^k \end{bmatrix}$$

Solve using LDL factorization

Indirect method

Positive-definite matrix

$$\left(P + \sigma I + \rho A^T A\right) x = \sigma x^k - q + A^T\left(\rho z^k - y^k\right)$$

Solve using conjugate gradient

CUDA Direct Sparse Solver (cuDSS) Integration

LDL solver from cuDSS

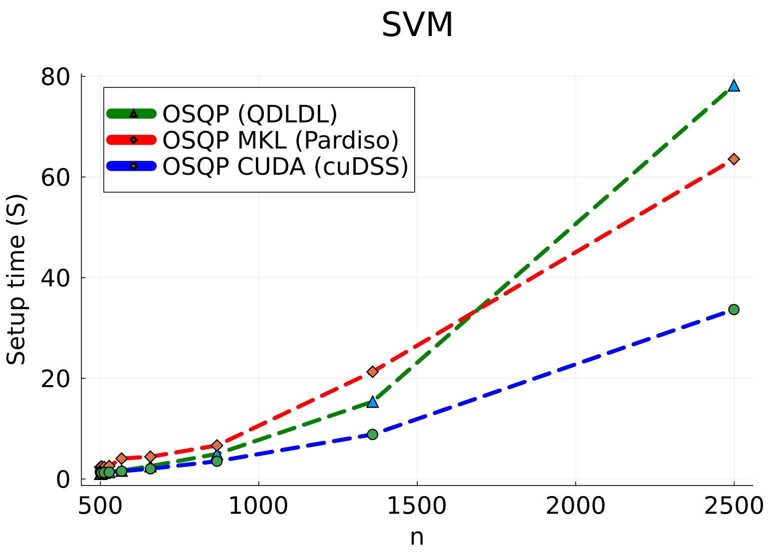
Solution of 2x2 KKT system:

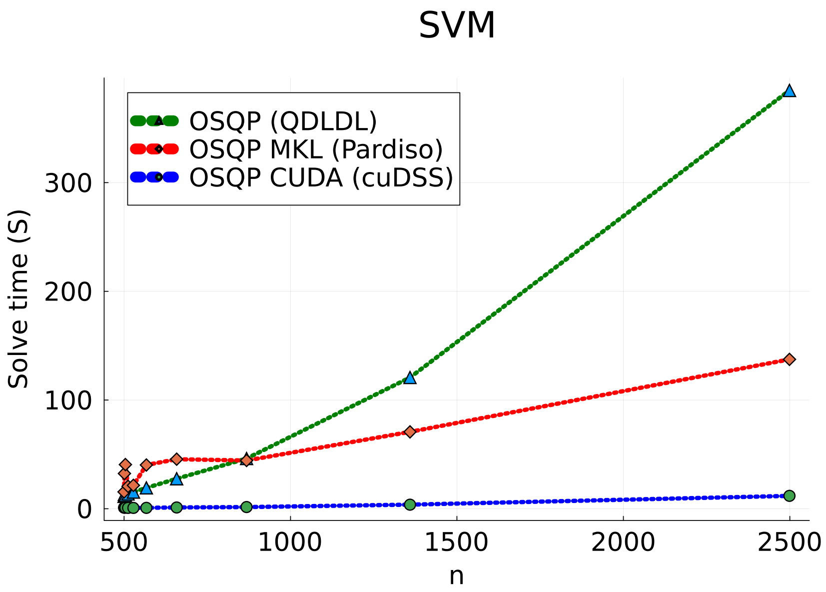
$$\begin{bmatrix} P + \sigma I & A^T \\ A & -\frac{1}{\rho} I \end{bmatrix} \begin{bmatrix} x \\ \nu \end{bmatrix} = \begin{bmatrix} \sigma x^k - q \\ z^k - \frac{1}{\rho} y^k \end{bmatrix}$$

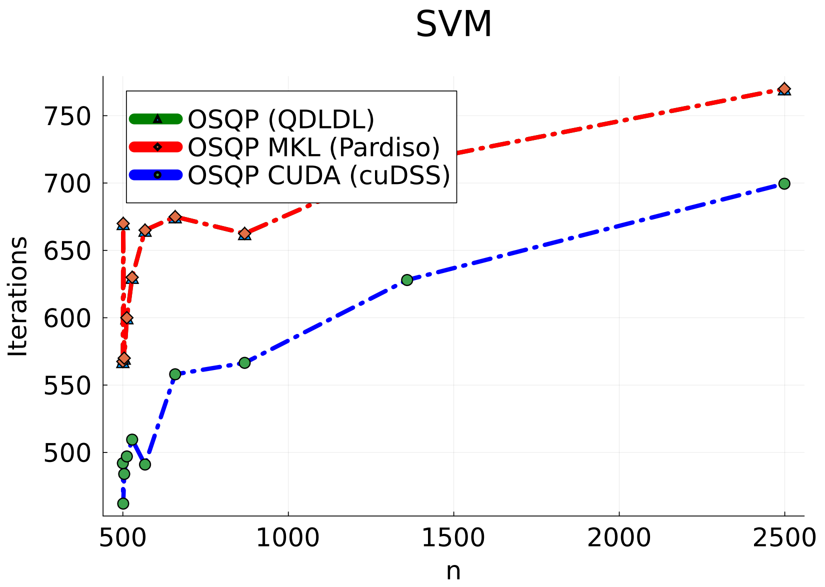
GPU-native solver

- Form 2x2 KKT matrix structure on CPU in setup, then transfer to GPU
- All solve computations are on GPU

cuDSS – Performance vs. other direct solvers







# of variables in problem = 101*n
# of constraints in problem = 200*n
Size of KKT matrix = 301*n X 301*n

Solved to low accuracy: 1e-3

Cuda: NVidia H100 80GB
CPU: Intel Xeon Gold 6548Y+ (64 cores)

Linear System Solvers Available

	Built-in	**MKL**	**CUDA**
Direct	✓ (QDLDL)	✓ (Pardiso)	✓ (cuDSS)
Indirect	✗	✓ (PCG)	✓ (PCG)

The future

Future improvements

Software Engineering

- CUDA Streams
- CUDA Graphs
- Batched mode

Algorithmic Improvements

- Restarting
- Mixed/low precision
- ADMM-tailored solution derivatives
- Improved step-size selection strategies

Learn more & get OSQP

C Library

github.com/osqp/osqp

Python

pip install osqp*[mkl,cu12]*

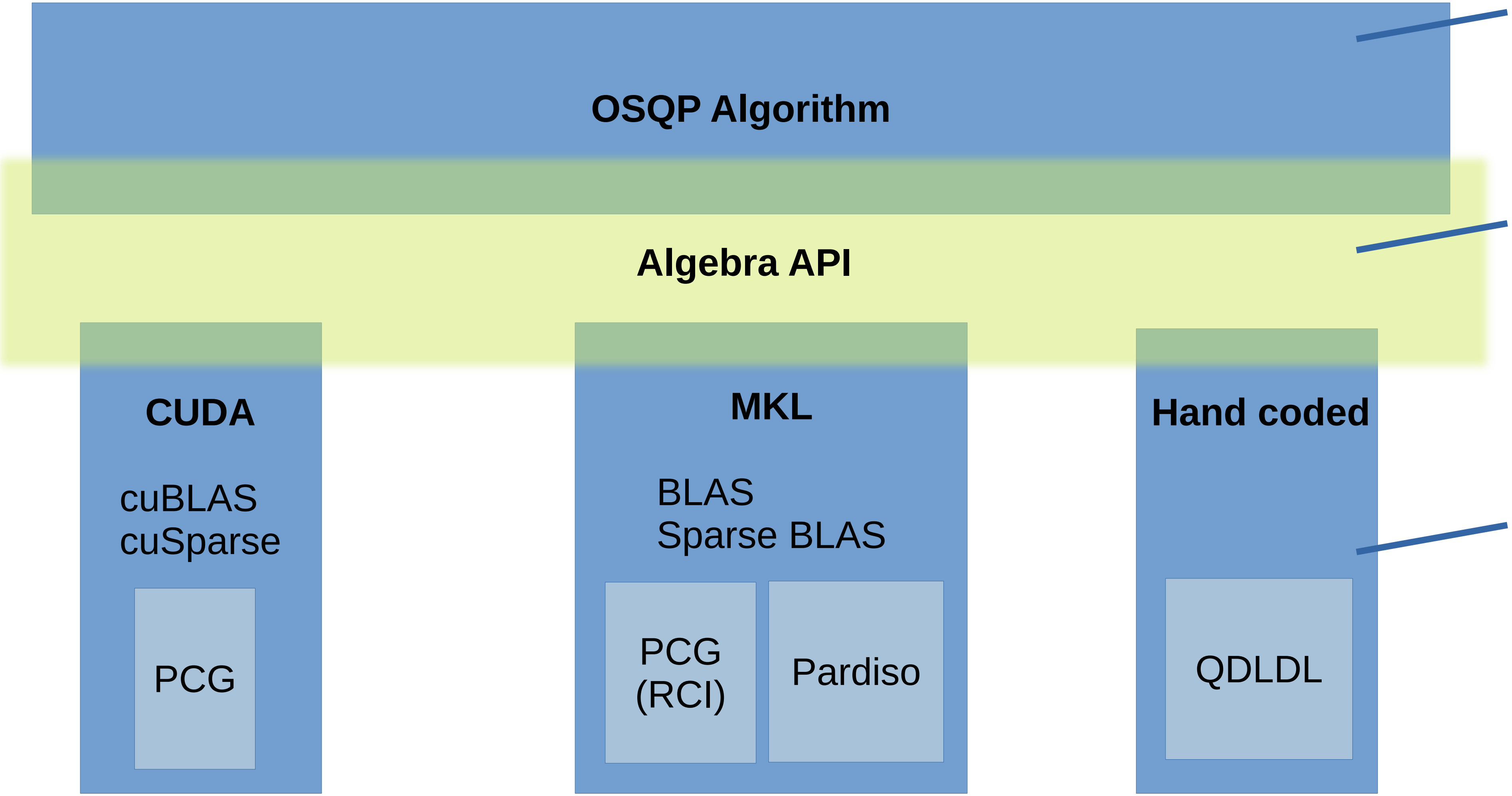
Julia

Pkg.add([“OSQP”, *“OSQPMKL”, “OSQPCUDA”]*)

[**osqp.org**](https://osqp.org) [**osqp.org/docs/**](https://osqp.org/docs/)

Backup slides

Modular Linear Algebra



Operates on opaque linear algebra types:
OSQPVector
OSQPMatrix

Defines operations on opaque types, e.g.,
OSQPVectorf_dot_prod
OSQPMatrix_Axpy

- Implements operations on opaque types
- Compile-time selected

Modular Linear Algebra – Key Points

Builtin

- Hand coded
- CSC-based sparse matrices
- Library free
- Linear system solvers
  - QDLDL

MKL

- CSC-based sparse matrices
- BLAS vector operations
- Linear system solvers
  - Pardiso
  - RCI Preconditioned conjugate gradient

CUDA

- cuSparse & cuBLAS libraries
- CSC & CSR matrices
- All data fully GPU resident
  - *osqp_setup* – Data copied to OSQP GPU workspace
  - *osqp_solve* – CPU-managed control flow, only transfer status values
- Linear system solvers
  - Preconditioned conjugate gradient

Same OSQP API for all backends

Modular Linear Algebra from Julia

One-line import change

```julia
using JuMP
using OSQP
using OSQPMKL

model = Model( () -> OSQP.Optimizer(OSQPMKLAlgebra()) )

@variable(model, x >= 0)
@variable(model, 0 <= y <= 3)
@objective(model, Min, 12x + 20y)
@constraint(model, c1, 6x + 8y >= 100)
@constraint(model, c2, 7x + 12y >= 120)
print(model)
optimize!(model)
```

It works with JuMP