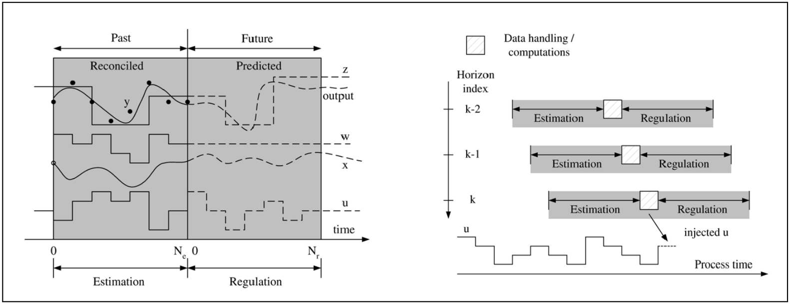
Algorithms and Methods for Fast Model Predictive Control

Gianluca Frison

Technical University of Denmark
Department of Applied Mathematics and Computer Science

13 April 2016

Model Predictive Control (and Moving Horizon Estimation)



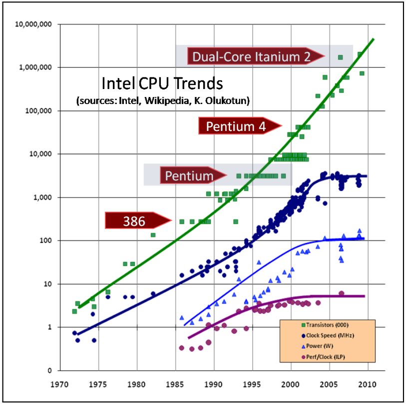
Background: Model Predictive Control

Model Predictive Control

+ optimal control signal
+ easy incorporation of forecasts
+ predictive adaptation to setpoint changes
+ natural handling of constraints and MIMO
+ generalization to non-linear systems
- need for a model
- an optimization problem at each sampling instant

Background: "The Free Lunch is Over" (2005)

- For decades, CPU frequency increase boosted CPU performance
- Around 2002 CPU frequency stalled, transistor count kept doubling every 2 years (Moore's law)
- Use additional transistors to increase CPU performance-per-clock: vectorization (SIMD), parallelization (multi-core)



Background: "The Free Lunch is Over" (2005)

Consequences:

- "If your program is too slow, just wait for the next computer generation" is not true any more
- Vectorization and parallelization require extra programming effort (compilers can't do proper auto-vec. and auto-par.)
- In real-time critical applications, more performance requires more (hardware-exploiting) software optimization

Thesis approach

Algorithms and Methods for Fast Model Predictive Control

- Methods: dense linear algebra implementation methods for embedded optimization (Part I)
- Algorithms: structure-exploiting algorithms for MPC (Parts II and III)
- Both algorithms and their implementation are equally important in the development of fast solvers
- Bottom-up approach: speed-up performance-critical routines to speed-up the overall solvers

Part I

Dense Linear Algebra Routines for Embedded Optimization

Framework: embedded optimization

Assumptions about embedded optimization:

- Computational speed is a key factor: solve optimization problems in real-time on resources-constrained hardware.
- Data matrices are reused several times (e.g. at each optimization algorithm iteration): look for a good data structure.
- Structure-exploiting algorithms can exploit the high-level sparsity pattern: data matrices assumed dense.
- Size of matrices is relatively small (tens or few hundreds): generally fitting in cache.

BLAS

- Basic Linear Algebra Subprograms
- The de-facto standard interface for linear algebra
- Implementations optimized for many computer architectures
  - but optimized for large-scale matrices
  - often poor small-scale performance (large overhead)
- Divided into 3 levels:
  - level 1: vector-vector operations: $\mathcal{O}(n)$ storage, $\mathcal{O}(n)$ flops
  - level 2: matrix-vector operations: $\mathcal{O}(n^2)$ storage, $\mathcal{O}(n^2)$ flops
  - level 3: matrix-matrix operations: $\mathcal{O}(n^2)$ storage, $\mathcal{O}(n^3)$ flops
- an access to memory (memop) is much slower than a flop
  - in level 3 BLAS there is a lot of space for optimization

LAPACK

- Linear Algebra PACKage
- Standard software library for numerical linear algebra
- E.g. Cholesky factorization, matrix inversion
- Built on top of BLAS
  - unblocked routines using level 1 & 2 BLAS (small matrices)
  - blocked routines using level 3 BLAS (large matrices)
- Bad multi-thread scalability (not explicit parallelism)
  - PLASMA project
- Bad small-scale performance (level 1 & 2 BLAS)
  - examples later in the talk

(D)GEMM

- (DP) general matrix-matrix multiplication
- Key sub-operation in all level 3 BLAS & LAPACK
- Often used to benchmark BLAS implementations
- In optimized BLAS, high-performance by employing:
  - blocking for registers
  - machine-specific instructions (e.g. SIMD)
  - special internal matrix format
  - blocking for cache

Computational Performance

- measured in Gflops = (# of flops) / ($10^9\cdot$ solution time in s)
- e.g. `dsyrk + dpotrf` costs $n^3 + \frac{1}{3}n^3 = \frac{4}{3}n^3$ flops
- compared with theoretical peak performance
- measure of CPU utilization
- useful to identify performance bottlenecks
- room for improvement?

Implementation of dsyrk + dpotrf on Intel Ivy-Bridge

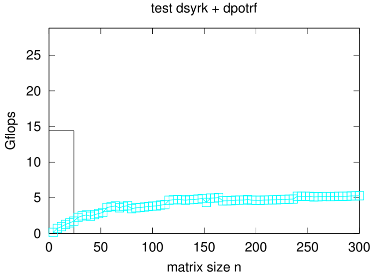
Test operation:

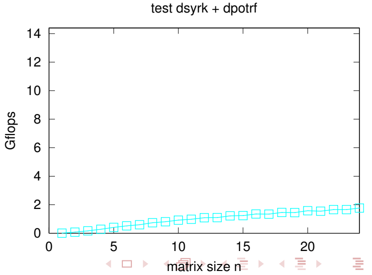
$$\mathcal{L} = \left(\mathcal{Q} + \mathcal{A} \cdot \mathcal{A}^T\right)^{1/2}$$

NetlibBLAS

- Reference BLAS & LAPACK
- triple-loop linear algebra
- machine independent code

[all code is single-threaded]
[all code compiled with gcc]





Triple-loop implementation

- less memops if inner loop over k: each element is computed as

$$c_{ij} = c_{ij} + \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, \quad i = 0, \dots, n-1, \quad j = 0, \dots, n-1$$

- issue #1: dependent operations, can not hide latency (since FP instructions are pipelined, latency > throughput)

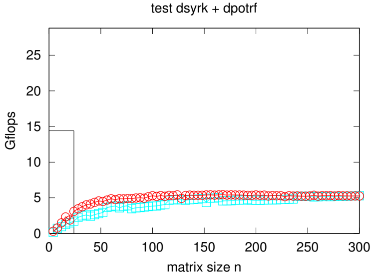
$$\begin{aligned} c_{ij} &= c_{ij} + a_{i0} \cdot b_{0j} \\ c_{ij} &= c_{ij} + a_{i1} \cdot b_{1j} \\ c_{ij} &= c_{ij} + a_{i2} \cdot b_{2j} \\ c_{ij} &= c_{ij} + a_{i3} \cdot b_{3j} \end{aligned}$$

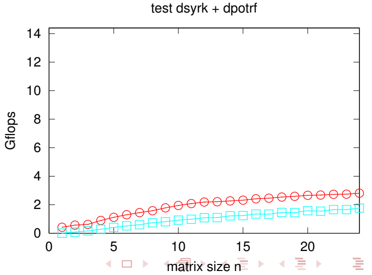
- issue #2: ratio flops/memops=2/2=1

Implementation of `dsyrk` + `dpotrf` on Intel Ivy-Bridge

Code Generation

- e.g. fix the size of the loops: compiler can unroll loops and avoid branches
- need to generate the code for each problem size

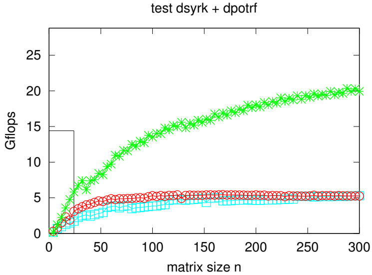


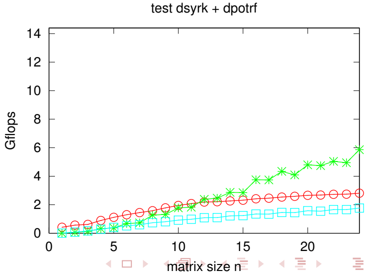


Implementation of dsyrk + dpotrf on Intel Ivy-Bridge

OpenBLAS

- high-performance for large matrices

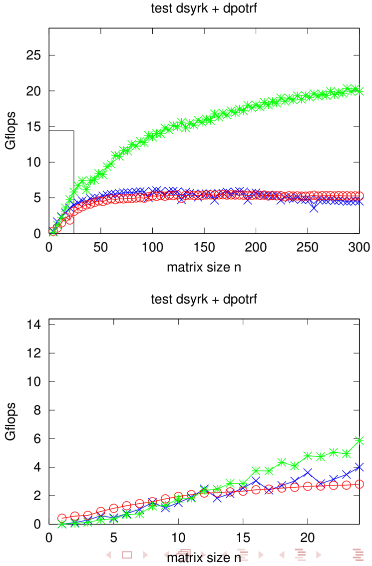




Implementation of dsyrk + dpotrf on Intel Ivy-Bridge

HPMPC - blocking for registers

- HPMPC: library for High-Performance implementation of solvers for Model Predictive Control
- hide latency of instructions
- reuse of matrix elements once in registers



Blocking for registers

- idea: use registers to hold a sub-matrix of C
- e.g. 2×2 sub-matrix in registers

	$b_{k,j+0}$	$b_{k,j+1}$
$a_{i+0,k}$	$c_{i+0,j+0} + a_{i+0,k} \cdot b_{k,j+0}$	$c_{i+0,j+1} + a_{i+0,k} \cdot b_{k,j+1}$
$a_{i+1,k}$	$c_{i+1,j+0} + a_{i+1,k} \cdot b_{k,j+0}$	$c_{i+1,j+1} + a_{i+1,k} \cdot b_{k,j+1}$

- solution #1: independent operations, can hide latency

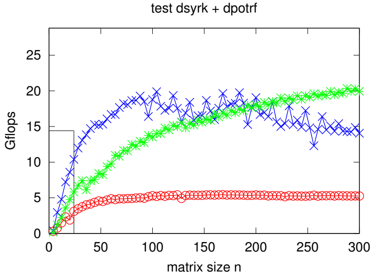
$$
\begin{aligned}
c_{i+0,j+0} &= c_{i+0,j+0} + a_{i+0,0} \cdot b_{0,j+0} \\
c_{i+1,j+0} &= c_{i+1,j+0} + a_{i+1,0} \cdot b_{0,j+0} \\
c_{i+0,j+1} &= c_{i+0,j+1} + a_{i+0,0} \cdot b_{0,j+1} \\
c_{i+1,j+1} &= c_{i+1,j+1} + a_{i+1,0} \cdot b_{0,j+1} \\
c_{i+0,j+0} &= c_{i+0,j+0} + a_{i+0,1} \cdot b_{1,j+0} \\
\ldots &= \ldots
\end{aligned}
$$

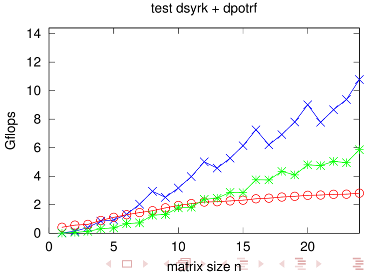
- solution #2: ratio flops/memops=8/4=2

Implementation of `dsyrk` + `dpotrf` on Intel Ivy-Bridge

HPMPC - SIMD instructions

- use SIMD (Single-Instruction Multiple-Data)
- AVX: 4 doubles per vector
- performance drop for *n* multiple of 32 - cache associativity





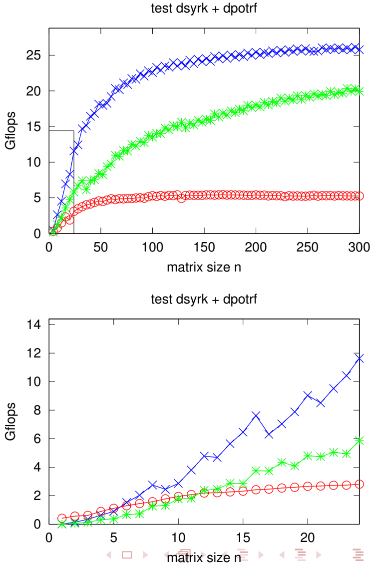
Use of SIMD

- idea: perform the same instructions on small vectors of data, element-wise
- e.g. 2-wide registers, $4x2$ sub-matrix

$$
\begin{array}{c|cc}
 & b_0 & b_1 \\
\hline
\begin{array}{c} a_0 \\ a_1 \end{array} & \begin{bmatrix} c_{00} \\ c_{10} \end{bmatrix} + \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_0 \end{bmatrix} & \begin{bmatrix} c_{01} \\ c_{11} \end{bmatrix} + \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_1 \end{bmatrix} \\
\begin{array}{c} a_2 \\ a_3 \end{array} & \begin{bmatrix} c_{20} \\ c_{30} \end{bmatrix} + \begin{bmatrix} a_2 \\ a_3 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_0 \end{bmatrix} & \begin{bmatrix} c_{21} \\ c_{31} \end{bmatrix} + \begin{bmatrix} a_2 \\ a_3 \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_1 \end{bmatrix}
\end{array}
$$

- 2-wide SIMD gives up to 2x speed-up

Implementation of dsyrk + dpotrf on Intel Ivy-Bridge



HPMPC - panel-major matrix format

- panel-major matrix format: arrange matrix elements in memory as accessed by the dgemm routine
- smooth performance

Access pattern in optimized BLAS

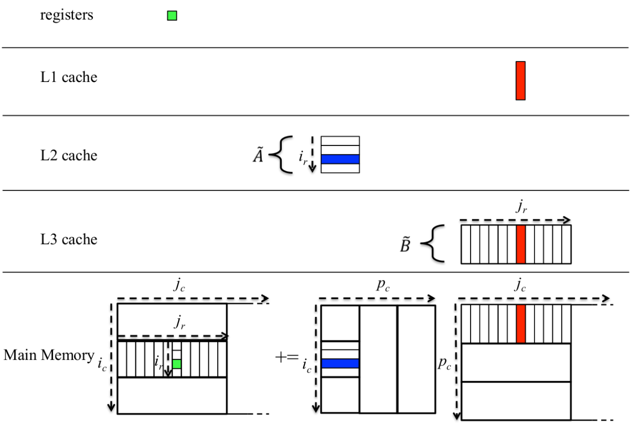
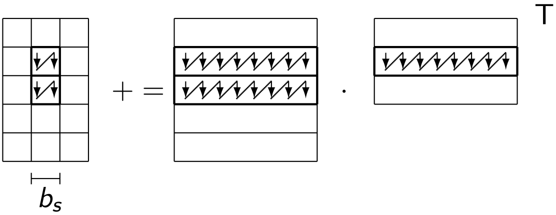


Figure : Access pattern of data in different cache levels for the `dgemm` routine in GotoBLAS/OpenBLAS/BLIS. Data is packed (on-line) into buffers following the access pattern.

Panel-major matrix format



- matrix elements are stored in the same order such as the `gemm` kernel accesses them
- optimal 'NT' variant (namely, A not-transposed, B transposed)
- panels width b_s is the same for the left and the right matrix operand, as well as for the result matrix

Optimized BLAS vs HPMPC software stack

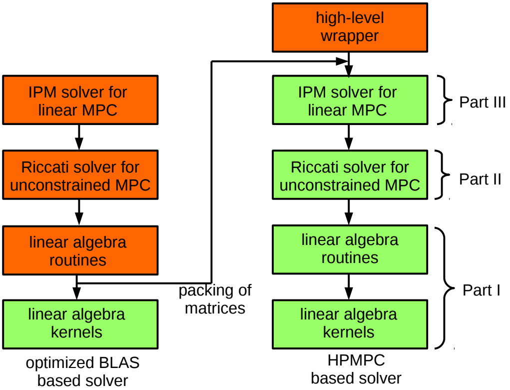
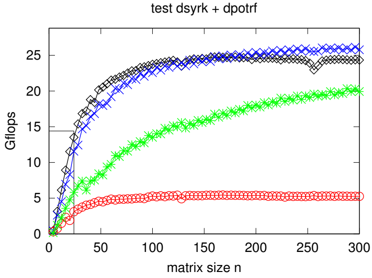


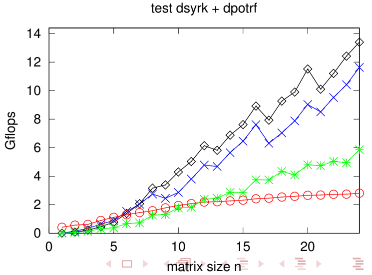
Figure : Structure of a Riccati-based IPM for linear MPC problems when implemented using linear algebra in either optimized BLAS or HPMPC. Routines in the orange boxes use matrices in column-major format, routines in the green boxes use matrices in panel-major format.

Implementation of dsyrk + dpotrf on Intel Ivy-Bridge

HPMPC - merging of linear algebra routines

- specialized kernels for complex operations
- improves small-scale performance
- worse large-scale performance





Merging of linear algebra routines - `dsyrk + dpotrf`

$$\mathcal{L} = \left(\mathcal{Q} + \mathcal{A} \cdot \mathcal{A}^T\right)^{1/2} =$$

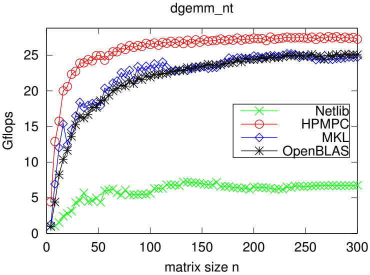
$$\begin{bmatrix} \mathcal{L}_{00} & * & * \\ \mathcal{L}_{10} & \mathcal{L}_{11} & * \\ \mathcal{L}_{20} & \mathcal{L}_{21} & \mathcal{L}_{22} \end{bmatrix} = \left(\begin{bmatrix} \mathcal{Q}_{00} & * & * \\ \mathcal{Q}_{10} & \mathcal{Q}_{11} & * \\ \mathcal{Q}_{20} & \mathcal{Q}_{21} & \mathcal{Q}_{22} \end{bmatrix} + \begin{bmatrix} \mathcal{A}_0 \\ \mathcal{A}_1 \\ \mathcal{A}_2 \end{bmatrix} \cdot \begin{bmatrix} \mathcal{A}_0^T & \mathcal{A}_1^T & \mathcal{A}_2^T \end{bmatrix} \right)^{1/2} =$$

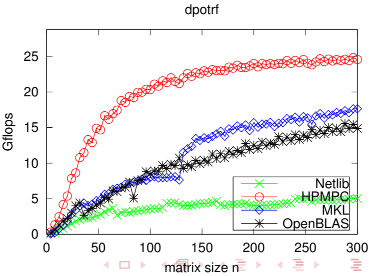
$$\begin{bmatrix} (\mathcal{Q}_{00} + \mathcal{A}_0 \cdot \mathcal{A}_0^T)^{1/2} & * & * \\ (\mathcal{Q}_{10} + \mathcal{A}_1 \cdot \mathcal{A}_0^T)\mathcal{L}_{00}^{-T} & (\mathcal{Q}_{11} + \mathcal{A}_1 \cdot \mathcal{A}_1^T - \mathcal{L}_{10} \cdot \mathcal{L}_{10}^T)^{1/2} & * \\ (\mathcal{Q}_{20} + \mathcal{A}_2 \cdot \mathcal{A}_0^T)\mathcal{L}_{00}^{-T} & (\mathcal{Q}_{21} + \mathcal{A}_2 \cdot \mathcal{A}_1^T - \mathcal{L}_{20} \cdot \mathcal{L}_{10}^T)\mathcal{L}_{11}^{-T} & (\mathcal{Q}_{22} + \mathcal{A}_2 \cdot \mathcal{A}_2^T - \mathcal{L}_{20} \cdot \mathcal{L}_{20}^T - \mathcal{L}_{21} \cdot \mathcal{L}_{21}^T)^{1/2} \end{bmatrix}$$

- each sub-matrix computed using a single specialized routine
  - reduce number of function calls
  - reduce number of load and store of the same data

High-performance LAPACK for small matrices

- Implemented as level 3 BLAS routines
- Blocking at registers level
- Specialized kernels merging `gemm` kernel with unblocked LAPACK routines





Part II

Algorithms for Unconstrained MPC and MHE Problems

Linear Time-Variant Optimal Control Problem

$$\min_{u,x} \quad \sum_{n=0}^{N-1} \frac{1}{2} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}^T \begin{bmatrix} R_n & S_n & r_n \\ S_n^T & Q_n & q_n \\ r_n^T & q_n^T & \rho_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_N \\ 1 \end{bmatrix}^T \begin{bmatrix} Q_N & q_N \\ q_N^T & \rho_N \end{bmatrix} \begin{bmatrix} x_N \\ 1 \end{bmatrix}$$

$$s.t. \quad x_{n+1} = A_n x_n + B_n u_n + b_n, \qquad n = 0, \dots, N-1$$

$$x_0 = \hat{x}_0$$

$$0 = D_N x_N + d_N$$

- MPC vs MHE
- equality constraints at last stage

Karush-Kuhn-Tucker optimality conditions

KKT system (for $N = 2$)

$$
\left[\begin{array}{ccc|ccc|cc}
Q_0 & S_0^T & A_0^T & & & & & \\
S_0 & R_0 & B_0^T & & & & & \\
A_0 & B_0 & & -I & & & & \\
\hline
 & & -I & Q_1 & S_1^T & A_1^T & & \\
 & & & S_1 & R_1 & B_1 & & \\
 & & & A_1 & B_1 & & -I & \\
\hline
 & & & & & -I & Q_2 & D_2^T \\
 & & & & & & D_2 &
\end{array}\right]
\left[\begin{array}{c}
x_0 \\ u_0 \\ \lambda_0 \\ \hline x_1 \\ u_1 \\ \lambda_1 \\ \hline x_2 \\ \lambda_2
\end{array}\right]
=
\left[\begin{array}{c}
-q_0 \\ -r_0 \\ -b_0 \\ \hline -q_1 \\ -r_1 \\ -b_1 \\ \hline -q_2 \\ -d_2
\end{array}\right]
$$

- Large, structured system of linear equations
- Sub-matrices are assumed dense or diagonal

Backward Riccati recursion

$$P_n = Q_n + A_n^T P_{n+1} A_n - \\ - (S_n^T + A_n^T P_{n+1} B_n)(R_n + B_n^T P_{n+1} B_n)^{-1}(S + B_n^T P_{n+1} A_n)$$

- structure-exploiting factorization of the KKT matrix
- begins factorization at the last stage
- does not require invertible Hessian
- can not handle additional equality constraints at the last stage
- naturally handles MPC problems
- $\mathcal{O}(N(n_x + n_u)^3)$ flops

Backward Riccati recursion

Main loop

```
...
for n ← N − 1, ..., 0 do
    A_n^T L_{n+1} ← [B_n^T; A_n^T] · L_{n+1,22}                 ▷ trmm
    M_n ← Q_n + (A_n^T L_{n+1}) · (A_n^T L_{n+1})^T              ▷ syrk
    [L_{n,11} ; L_{n,21} L_{n,22}] ← M_n^{1/2}                   ▷ potrf
end for
...
```

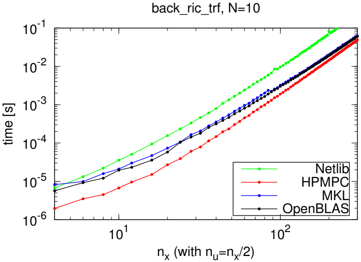
$$\mathcal{A}_n^T \mathcal{L}_{n+1} \leftarrow \begin{bmatrix} B_n^T \\ A_n^T \end{bmatrix} \cdot L_{n+1,22} \qquad \triangleright \texttt{trmm}$$

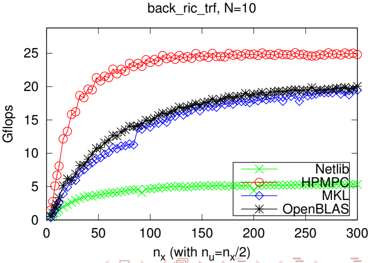
$$\mathcal{M}_n \leftarrow \mathcal{Q}_n + (\mathcal{A}_n^T \mathcal{L}_{n+1}) \cdot (\mathcal{A}_n^T \mathcal{L}_{n+1})^T \qquad \triangleright \texttt{syrk}$$

$$\begin{bmatrix} L_{n,11} & \\ L_{n,21} & L_{n,22} \end{bmatrix} \leftarrow \mathcal{M}_n^{1/2} \qquad \triangleright \texttt{potrf}$$

Backward Riccati recursion

- HPMPC much better for small problems
- performance plot similar to linear algebra ones





Forward Schur-complement recursion

$$\Sigma_{n+1} = Q_n + \left(\begin{bmatrix} A_n & B_n \end{bmatrix} \begin{bmatrix} \Sigma_n & S_n^T \\ S_n & R_n \end{bmatrix}^{-1} \begin{bmatrix} A_n \\ B_n \end{bmatrix} \right)^{-1}$$

- structure-exploiting factorization of the KKT matrix
- begins factorization at the first stage
- requires invertible Hessian (or regularization)
- handles additional equality constraints at the last stage
- naturally handles MHE problems
- $\mathcal{O}(N(n_x + n_u)^3)$ flops

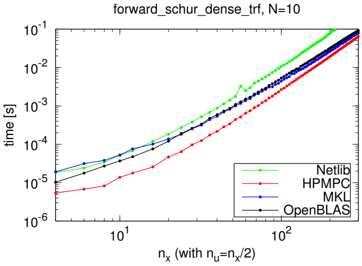
Forward Schur-complement recursion

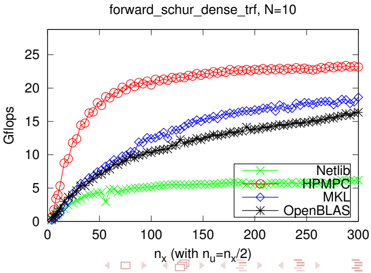
Main loop

$\ldots$
for $n \leftarrow 1, \ldots, N-1$ **do**
  $\Sigma \leftarrow Q_n + U_n \cdot U_n^T$ ▷ `lauum`
  $\mathcal{Q} \leftarrow \begin{bmatrix} \Sigma & 0 \\ S_n & R_n \end{bmatrix}$
  $\mathcal{A} \leftarrow \begin{bmatrix} A_n & B_n \end{bmatrix}$
  $\mathcal{L}_n \leftarrow \mathcal{Q}^{1/2}$ ▷ `potrf`
  $\mathcal{AL}_n \leftarrow \mathcal{A} \cdot \mathcal{L}^{-T}$ ▷ `trsm`
  $P_{inv} \leftarrow \mathcal{AL}_n \cdot \mathcal{AL}_n^T$ ▷ `syrk`
  $L \leftarrow P_{inv}^{1/2}$ ▷ `potrf`
  $U_{n+1} \leftarrow L^{-T}$ ▷ `trtri`
`end for`
$\ldots$

Forward Schur-complement recursion

- similar considerations to backward Riccati recursion
- but slightly worse performance due to more LAPACK routines





Hessian condensing - idea

- Idea: use state-space equation to eliminate states variables from the optimization problem
- Smaller but dense Hessian

$$\begin{bmatrix} B_0^T Q_1 B_0 + B_0^T A_1^T Q_2 A_1 B_0 + B_0^T A_1^T A_2^T Q_3 A_2 A_1 B_0 & * & * \\ B_1^T Q_2 A_1 B_0 + B_1^T A_2^T Q_3 A_2 A_1 B_0 & B_1^T Q_2 B_1 + B_1^T A_2^T Q_3 A_2 B_1 & * \\ B_2^T Q_3 A_2 A_1 B_0 & B_2^T Q_3 A_2 B_1 & B_2^T Q_3 B_2 \end{bmatrix}$$

Hessian condensing - MPC case

Initial state and state space equations

$$x_0 = \hat{x}_0, \quad x_{n+1} = A_n x_n + B_n u_n + b_n$$

rewritten as

$$\bar{A}\bar{x} = \bar{B}\bar{u} + \bar{b} \quad \Rightarrow \quad \bar{x} = \bar{A}^{-1}\bar{B}\bar{u} + \bar{A}^{-1}\bar{b} \doteq \Gamma_u \bar{u} + \Gamma_{x,b}$$

where $(N = 3)$

$$\bar{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \bar{u} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix}, \quad \bar{b} = \begin{bmatrix} \hat{x}_0 \\ b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

$$\bar{A} = \begin{bmatrix} I & & & \\ -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix}, \quad \bar{B} = \begin{bmatrix} B_0 & & \\ & B_1 & \\ & & B_2 \end{bmatrix}$$

Hessian condensing - MPC case

Key idea to have $\mathcal{O}(N^2)$ Hessian condensing algorithms

$$\bar{A}^{-1} = \begin{bmatrix} I & & & \\ -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix}^{-1} = \begin{bmatrix} I & & & \\ A_0 & I & & \\ A_1 A_0 & A_1 & I & \\ A_2 A_1 A_0 & A_2 A_1 & A_2 & I \end{bmatrix}$$

- $\bar{A}$ is sparse ($\mathcal{O}(N)$ n.z.) but $\bar{A}^{-1}$ is dense ($\mathcal{O}(N^2)$ n.z.)

$$\Gamma_u = \begin{bmatrix} I & & & \\ -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix}^{-1} \begin{bmatrix} & & \\ B_0 & & \\ & B_1 & \\ & & B_2 \end{bmatrix} = \begin{bmatrix} & & \\ B_0 & & \\ A_1 B_0 & B_1 & \\ A_2 A_1 B_0 & A_2 B_1 & B_2 \end{bmatrix}$$

- backsolve vs matrix multiplication: n_x vs $N n_u$ trade-off

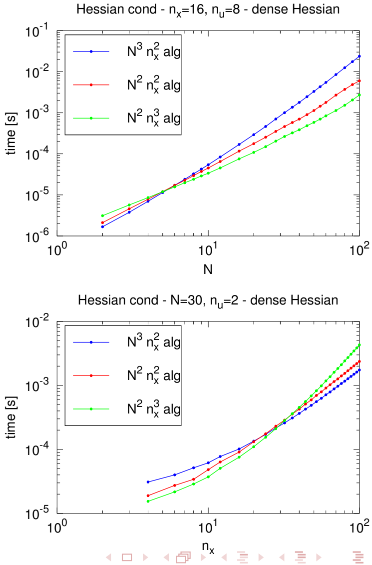
Hessian condensing - MPC case

If $S_n = 0$, condensed Hessian

$$H = \bar{R} + \Gamma_u^T \bar{Q} \Gamma_u$$
$$= \bar{R} + \bar{B}^T \bar{A}^{-T} \bar{Q} \bar{A}^{-1} \bar{B}$$

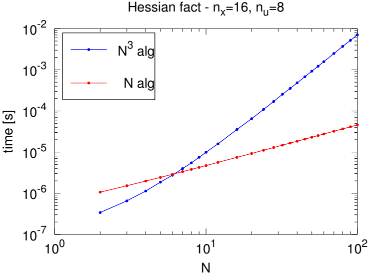
Three algorithms depending on the order of operations

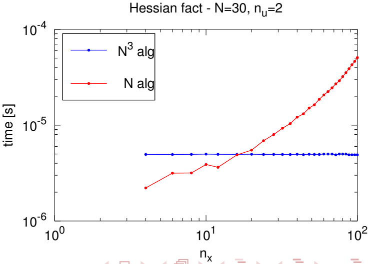
- $\mathcal{O}(N^3)$ and $\mathcal{O}(n_x^2)$
- $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$
- $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$



Hessian factorization - MPC case

- $\mathcal{O}(N^3)$ classical Cholesky factorization of condensed Hessian
- $\mathcal{O}(N)$ structure-exploiting Cholesky factorization of permuted condensed Hessian
  - starts form last stage
  - directly builds the factorized Hessian
  - combined with $(\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2))$ or $(\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3))$ Hessian condensing algorithms

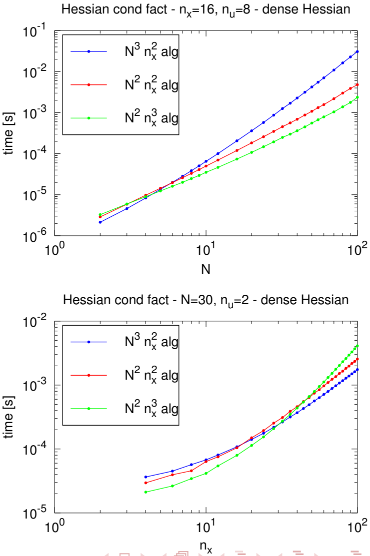




Hessian condensing & factorization - MPC case

Still three algorithms

- $\mathcal{O}(N^3)$ and $\mathcal{O}(n_x^2)$
- $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$
- $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$



Hessian condensing - MHE case

State space equations (no initial state constraint)

$$x_{n+1} = A_n x_n + B_n u_n + b_n$$

rewritten as

$$\bar{A}\bar{x} = \bar{B}\bar{u} + \bar{b}$$

where ($N = 3$)

$$\bar{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \bar{u} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix}, \quad \bar{b} = \begin{bmatrix} \color{red}{0} \\ b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

$$\bar{A} = \begin{bmatrix} \color{red}{0} & & & \\ -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix}, \quad \bar{B} = \begin{bmatrix} & & \\ B_0 & & \\ & B_1 & \\ & & B_2 \end{bmatrix}$$

Recover invertibility of $\bar{A}$

$$\bar{A}\bar{x} = \begin{bmatrix} I & & & \\ -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} - \begin{bmatrix} I \\ 0 \\ 0 \\ 0 \end{bmatrix} x_0 = \mathcal{\bar{A}}\bar{x} - \mathcal{E}_0 x_0$$

gives

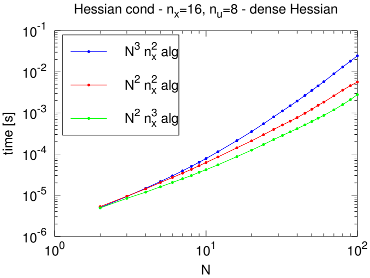
$$\begin{aligned} \bar{x} &= \bar{\mathcal{A}}^{-1}\mathcal{E}_0 x_0 + \bar{\mathcal{A}}^{-1}\bar{B}\bar{u} + \bar{\mathcal{A}}^{-1}\bar{b} \\ &= \bar{\mathcal{A}}^{-1}\bar{\mathcal{B}}\bar{v} + \bar{\mathcal{A}}^{-1}\bar{b} \end{aligned}$$

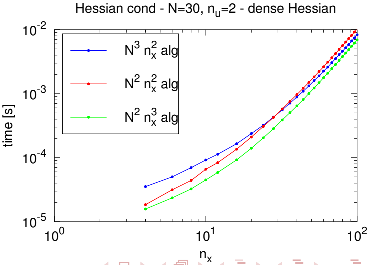
where

$$\bar{\mathcal{B}} = \left[\begin{array}{c|ccc} I & & & \\ & B_0 & & \\ & & B_1 & \\ & & & B_2 \end{array}\right], \quad \bar{v} = \left[\begin{array}{c} x_0 \\ \hline u_0 \\ u_1 \\ u_2 \end{array}\right]$$

Hessian condensing - MHE case

- x_0 as additional input (of size n_x) at stage -1
- all algorithms for MPC can be employed
- $\mathcal{O}(n_x^3)$ can not be avoided
- one algorithm is always better
- same applies for condensed Hessian factorization





Part III

Algorithms for Constrained and Nonlinear MPC Problems

Linear MPC problem

$$
\begin{aligned}
\min_{u,x} \quad & \sum_{n=0}^{N-1} \frac{1}{2} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}^T \begin{bmatrix} R_n & S_n & r_n \\ S_n^T & Q_n & q_n \\ r_n^T & q_n^T & \rho_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_N \\ 1 \end{bmatrix}^T \begin{bmatrix} Q_N & q_N \\ q_N^T & \rho_N \end{bmatrix} \begin{bmatrix} x_N \\ 1 \end{bmatrix} \\
s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n, \quad n = 0, \dots, N-1 \\
& x_0 = \hat{x}_0 \\
& u_n^{\mathrm{l}} \leq u_n \leq u_n^{\mathrm{u}}, \quad n = 0, \dots, N-1 \\
& x_n^{\mathrm{l}} \leq x_n \leq x_n^{\mathrm{u}}, \quad n = 1, \dots, N
\end{aligned}
$$

- only box constraints considered here

Interior Point Methods (IPMs) - general idea

- General QP program & KKT system

$$
\begin{aligned}
\min_{x,u} \quad & \frac{1}{2}x^T Hx + g^T x \\
s.t. \quad & Ax = b \\
& Cx \geq d
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
Hx + g - A^T\pi - C^T\lambda &= 0 \\
Ax - b &= 0 \\
Cx - d - t &= 0 \\
\lambda^T t &= 0 \\
(\lambda, t) &\geq 0
\end{aligned}
$$

- Newton method (2nd order method) for the KKT system

$$
\begin{bmatrix} H & -A^T & -C^T & 0 \\ A & 0 & 0 & 0 \\ C & 0 & 0 & -I \\ 0 & 0 & T_k & \Lambda_k \end{bmatrix}
\begin{bmatrix} \Delta x \\ \Delta \pi \\ \Delta \lambda \\ \Delta t \end{bmatrix}
= -
\begin{bmatrix} Hx_k - A^T\pi_k - C^T\lambda_k + g \\ A\pi_k - b \\ Cx_k - t_k - d \\ \Lambda_k T_k e + \sigma\mu_k e \end{bmatrix}
$$

Interior Point Methods (IPMs) - general idea

- structured system, can be rewritten as (augmented system)

$$\begin{bmatrix} H + C^T(T_k^{-1}\Lambda_k)C & -A^T \\ -A & 0 \end{bmatrix} \begin{bmatrix} x_k \\ \pi_k \end{bmatrix} =$$
$$= -\begin{bmatrix} g - C^T(\Lambda_k e + T_k^{-1}\Lambda_k d + T_k^{-1}\sigma\mu_k e) \\ b \end{bmatrix}$$

- KKT system of an equality constrained QP

Riccati-based IPM for the linear MPC problem

- In the linear MPC problem, KKT system of a LTV-OCP
- Most expensive operation: compute prediction-correction search directions (factorization of KKT system uses level 3 BLAS & LAPACK)
- Backward Riccati recursion (cubic & quadratic number of flops in stage variables number)
- All other operations in IPMs: linear number of flops in stage variables number

Riccati-based IPM in HPMPC

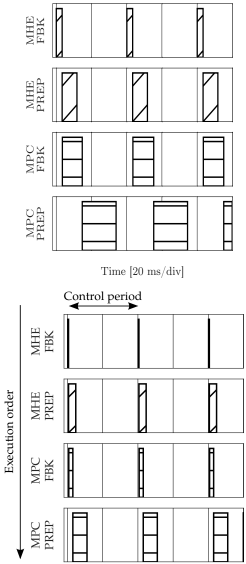
Table : Comparison of solvers for the box-constrained linear MPC problem: low- and high-level interfaces for the IPM in HPMPC, FORCES IPM and FORCES_Pro IPM. Run times are presented in seconds. For each problem size and solver, the number of IPM iterations is fixed to 10.

n_x	n_u	n_b	N	HPMPC low-level	HPMCP high-level	FORCES	FORCES Pro
4	1	5	10	$5.39 \cdot 10^{-5}$	$6.31 \cdot 10^{-5}$	$1.1 \cdot 10^{-4}$	$1.0 \cdot 10^{-4}$
8	3	11	10	$9.05 \cdot 10^{-5}$	$1.04 \cdot 10^{-4}$	$3.4 \cdot 10^{-4}$	$3.1 \cdot 10^{-4}$
12	5	17	30	$5.07 \cdot 10^{-4}$	$5.74 \cdot 10^{-4}$	$2.11 \cdot 10^{-3}$	$1.84 \cdot 10^{-3}$
22	10	32	10	$3.94 \cdot 10^{-4}$	$4.60 \cdot 10^{-4}$	$3.96 \cdot 10^{-3}$	$3.29 \cdot 10^{-3}$
30	14	44	10	$7.03 \cdot 10^{-4}$	$8.17 \cdot 10^{-4}$	$9.47 \cdot 10^{-3}$	$7.49 \cdot 10^{-3}$
60	29	89	30	$1.10 \cdot 10^{-2}$	$1.26 \cdot 10^{-2}$	$1.67 \cdot 10^{-1}$	$1.25 \cdot 10^{-1}$

Conclusion

Arrival point of the PhD work:

- High-performance QP solvers for linear MPC
- Riccati-based IPM for MPC and Schur-complement recursion for MHE interfaced with ACADO
- NMPC of a rotational start-up of a airbone wind energy system



Possible future directions - library

- split the library
  - BLASFEO (?): linear algebra routines for embedded optimization
  - HPMPC: algorithms for MPC built on top of it
- expand the library
  - add LU factorization for e.g. implicit integrators
  - add LDL factorization
  - embed partial condensing into Riccati-based IPM
- improve the library
  - agree on (and fix) interfaces
  - kernels in assembly to reduce code size
  - (re-)add single-precision support
  - add support for embedded hardware (e.g. Cortex M)
  - multi CPU cores

Possible future directions - direct sparse solvers

Direct sparse solvers (e.g. MA57 in IPOPT)

- built on top of level 3 BLAS (e.g. dgemm)
- analyzes the sparsity pattern of the problem, and gathers the non-zero elements into dense sub-matrices
- trade-off between sparsity exploitation (small sub-matrices) and BLAS performance (large sub-matrices): small-scale linear algebra performance is the key
- may lack the right routine in standard BLAS (e.g. in MA57, `dsyrk` with different factor matrices)

Re-implement MA57 on top of BLASFEO?

Thanks for your attention

Questions and comments?

Trend in (Intel) computing architectures

Table : Intel computer architectures: from 2-years cycle to 3 years-cycle

year	arch.	proc.	ISA	DP flops/cycle
2006/07	Merom	65 nm	SSSE3	4
2007/08	Penryn	45 nm	SSE4.1	4
2008/09	Nehalem	45 nm	SSE4.2	4
2010	Westmere	32 nm	SSE4.2	4
2011	Sandy-Bridge	32 nm	AVX	8
2012	Ivy-Bridge	22 nm	AVX	8
2013	Haswell	22 nm	AVX2/FMA3	16
2014	Haswell-refresh	22 nm	AVX2/FMA3	16
2014/15	Broadwell	14 nm	AVX2/FMA3	16
2015/16	Skylake	14 nm	AVX2/FMA3	16
2016/17	Kaby Lake	14 nm	AVX2/FMA3?	16?
2017/18?	Cannonlake	10 nm	AVX512?	32?

Code stack in HPMPC

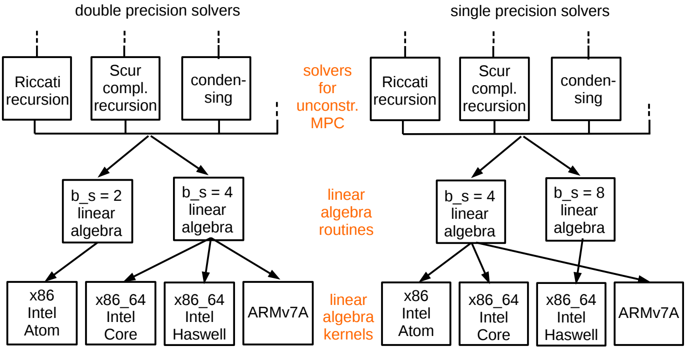
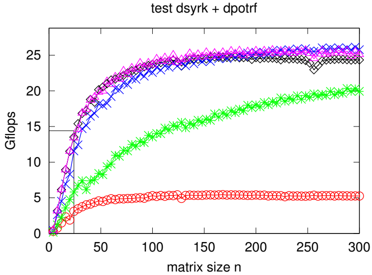


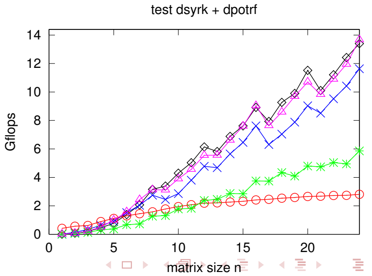
Figure : Structure of the linear algebra routines in HPMPC. The linear algebra kernels are tailored to each computer architecture. The linear algebra routines depend only on the panel height b_s (that may be different for single and double precision). The routines at higher levels in the routines hierarchy are completely architecture-independent.

Implementation of dsyrk + dpotrf on Intel Ivy-Bridge

HPMPC - swapping the order of outer loops

- has to be considered in case of not-squared kernels
- improves the L1 cache reuse
- machine-dependent code





Backward Riccati recursion

- Main operations per stage:
  - update

$$Q + A \cdot P \cdot A^T = Q + A \cdot (L \cdot L^T) \cdot A^T = Q + (A \cdot L) \cdot (A \cdot L)^T$$

    $\frac{7}{3}n_x^3 + 3n_x^2 n_u + n_x n_u^2$ flops
  - factorization-solution-downgrade

$$\mathcal{L} \leftarrow R^{-1}$$
$$L \leftarrow M \cdot \mathcal{L}^{-T}$$
$$P \leftarrow P - L \cdot L^T$$

    $n_x^2 n_u + n_x n_u^2 + \frac{1}{3}n_u^3$ flops
- Total flops: $N(\frac{7}{3}n_x^3 + 4n_x^2 n_u + 2n_x n_u^2 + \frac{1}{3}n_u^3)$

Froward Schur-complement recursion

- Main operations per stage:
  - computation of Schur complement

$$Q+A\cdot P^{-1}\cdot A^T = Q+A\cdot(L\cdot L^T)^{-1}\cdot A^T = Q+(A\cdot L^{-T})\cdot(A\cdot L^{-T})^T$$

    $\frac{7}{3}n_x^3 + 4n_x^2 n_u + 2n_x n_u^2 + \frac{1}{3}n_u^3$ flops
  - inversion of positive definite matrix

$$Q^{-1} = (L\cdot L^T)^{-1} = L^{-T}\cdot L^{-1}$$

    n_x^3 flops
- Total flops:
  - dense Hessian $N(\frac{10}{3}n_x^3 + 4n_x^2 n_u + 2n_x n_u^2 + \frac{1}{3}n_u^3)$
  - diagonal Hessian $N(\frac{10}{3}n_x^3 + n_x^2 n_u)$