BLASFEO

Gianluca Frison

University of Freiburg

BLIS retreat
September 19, 2017
Basic Linear Algebra Subroutines For Embedded Optimization
BLASFEO performance

- Intel Core i7 4800MQ
  - BLASFEO HP
  - OpenBLAS 0.2.19
  - MKL 2017.2.174
  - ATLAS 3.10.3
  - BLIS 0.1.6

- CAVEAT: BLASFEO is not API compatible with BLAS & LAPACK
Framework: embedded optimization

- Framework: embedded optimization and control
Embedded control

Model Predictive Control (and Moving Horizon Estimation)

- solve an optimization problem on-line
- sampling times in milli- or micro-second range
Karush-Kuhn-Tucker optimality conditions

KKT system (for $N = 2$)

\[
\begin{bmatrix}
Q_0 & S_0^T & A_0^T \\
S_0 & R_0 & B_0^T \\
A_0 & B_0 & -I
\end{bmatrix}
\begin{bmatrix}
x_0 \\
u_0 \\
\lambda_0
\end{bmatrix}
= 
\begin{bmatrix}
- q_0 \\
r_0 \\
b_0
\end{bmatrix}

\begin{bmatrix}
Q_1 & S_1^T & A_1^T \\
S_1 & R_1 & B_1 \\
A_1 & B_1 & -I
\end{bmatrix}
\begin{bmatrix}
x_1 \\
u_1 \\
\lambda_1
\end{bmatrix}
= 
\begin{bmatrix}
- q_1 \\
r_1 \\
b_1
\end{bmatrix}

\begin{bmatrix}
-1 & Q_2
\end{bmatrix}
\begin{bmatrix}
x_2
\end{bmatrix}
= 
\begin{bmatrix}
- q_2
\end{bmatrix}

- Large, structured system of linear equations
- Sub-matrices are assumed dense or diagonal
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.
- Limitations of embedded optimization hardware and toolchain: no external libraries and (static/dynamic) memory allocation
The origin

- HPMPC: library for High-Performance implementation of solvers for Model Predictive Control
How to optimize syrk + potrf (for embedded optimization)

- How to optimize dsyrk + dpotrf (for embedded optimization)
- Test operation:

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

- NetlibBLAS
How to optimize syrk + potrf (for embedded optimization)

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
How to optimize syrk+potrf (for embedded optimization)

OpenBLAS

Gianluca Frison

BLASFEO
How to optimize syrk + potrf (for embedded optimization)

HPMPC - register blocking
How to optimize syrk + potrf (for embedded optimization)

HPMPC - SIMD instructions

- performance drop for $n$ multiple of 32 - limited cache associativity
How to optimize syrk + potrf (for embedded optimization)

HPMPC - panel-major matrix format

- panel-major matrix format
- 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 format**: can do any operation, also on sub-matrices
- **matrix elements** are stored in (almost) the same order such as the `gemm` kernel accesses them
- **optimal ’NT’ `gemm` variant** (\(A\) not-transposed, \(B\) transposed)
- **panels width** \(p_s\) 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.
How to optimize syrk + potrf (for embedded optimization)

HPMPC - merging of linear algebra routines

- specialized kernels for complex operations
- improves small-scale performance
- worse large-scale performance
Merging of linear algebra routines: syrk + potrf

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

\[
\begin{bmatrix}
\mathcal{L}_{00} & * & * \\
\mathcal{L}_{10} & \mathcal{L}_{11} & * \\
\mathcal{L}_{20} & \mathcal{L}_{21} & \mathcal{L}_{22}
\end{bmatrix} = \left( \begin{bmatrix}
Q_{00} & * & * \\
Q_{10} & Q_{11} & * \\
Q_{20} & Q_{21} & Q_{22}
\end{bmatrix} + \begin{bmatrix}
A_0 \\
A_1 \\
A_2
\end{bmatrix} \cdot \begin{bmatrix}
A_0^T & A_1^T & A_2^T
\end{bmatrix} \right)^{1/2}
\]

\[
\begin{bmatrix}
(Q_{00} + A_0 \cdot A_0^T)^{1/2} \\
(Q_{10} + A_1 \cdot A_0^T) \mathcal{L}_{00}^{-T} \\
(Q_{20} + A_2 \cdot A_0^T) \mathcal{L}_{00}^{-T}
\end{bmatrix}
= \begin{bmatrix}
\begin{bmatrix}
Q_{11} + A_1 \cdot A_1^T - \mathcal{L}_{10} \cdot \mathcal{L}_{10}^T\end{bmatrix}^{1/2} & * \\
\begin{bmatrix}
Q_{21} + A_2 \cdot A_1^T - \mathcal{L}_{20} \cdot \mathcal{L}_{10}^T\end{bmatrix} \mathcal{L}_{11}^{-T} & \begin{bmatrix}
Q_{22} + A_2 \cdot A_2^T - \mathcal{L}_{20} \cdot \mathcal{L}_{20} - \mathcal{L}_{21} \cdot \mathcal{L}_{21}\end{bmatrix}^{1/2}
\end{bmatrix}
\]

- Each sub-matrix computed using a single specialized kernel
- Reduce number of function calls
- Reduce number of load and store of the same data
The present

- BLASFEO
- aim: satisfy the need for high-performance linear algebra routines for small dense matrices
- mean: adapt high-performance computing techniques to embedded optimization framework
- keep:
  - LA kernels (register-blocking, SIMD)
  - optimized data layout
- drop:
  - cache-blocking
  - on-line data packing
  - multi-thread (at least for now)
- add:
  - optimized (panel-major) matrix format
  - off-line data packing
  - assembly subroutines: tailored LA kernels & code reuse
Linear algebra level

- LA level definition
  - level 1: $O(n)$ storage, $O(n)$ flops
  - level 2: $O(n^2)$ storage, $O(n^2)$ flops
  - level 3: $O(n^2)$ storage, $O(n^3)$ flops
- in level 1 and level 2
  - reuse factor $O(1)$
  - memory-bounded
- in level 3
  - reuse factor $O(n)$
  - compute-bounded for large $n$
    - disregard $O(n^2)$ terms
  - typically memory-bounded for small $n$
    - minimize $O(n^2)$ terms $\Rightarrow$ BLASFEO playground!
BLASFEO Reference

- 2x2 blocking for registers
- column-major matrix format
- small code size
- ANSI C code, no external dependencies
- good performance for tiny matrices
BLASFEO High-Performance

- optimal blocking for registers + vectorization
- no blocking for cache
- panel-major matrix format
- hand-written assembly kernels
- optimized for highest performance for matrices fitting in cache
BLASFEO WRapper to BLAS and LAPACK

- provides a performance basis
- column-major matrix format
- optimized for many architectures
- possibly multi-threaded
- good performance for large matrices
Matrix structure

- problem: how to switch between panel-major and column-major formats?
- solution: use a C struct for the matrix ”object”
  
  (...) struct d_strmat *sA, int ai, int aj, ...)

- if column-major, the first matrix element is at
  
  int lda = sA->m;
  double *ptr = sA->pA + ai + aj*lda;

- if panel-major, the first matrix element is at
  
  int ps = sA->ps;
  int sda = sA->cn;
  int air = ai&(ps-1);
  double *ptr = sA->pA + (ai-air)*sda + air + aj*bs;

- custom matrix struct allows other tricks
  
  - extra linear storage for inverse of diagonal in factorizations
Some tests on Intel Core i7 4800MQ (Haswell)

Gianluca Frison

BLASFEO
Some tests on Intel Core i7 4800MQ (Haswell)

- dpotrf_l
- dgetrf
- spotrf_l
- dgelqf

Gianluca Frison

BLASFEO
Some tests on ARM Cortex A15 and ARM Cortex A57

![Graphs showing performance metrics for dgemm_nt and sgemm_nt operations on ARM Cortex A15 and A57.]
BLASFEO + PLASMA on Intel Core i7 4800MQ

Gianluca Frison
The end (for now)

- thank you for your attention!
- questions?