low-rank approximation on a GPU and its integration into an HSS solver

Ichi Yamazaki, Stan Tomov, Tim Dong

ICL friday lunch talk
03.08.2013

Outline: low-rank approximation on a GPU

1. QP3 - QR with column pivoting
   - reviewing QP3 algorithm
   - performance of QP3 on a GPU
2. HSS solver - Hierarchically Semi-Separable solver
   - HSS with QP3 on a GPU
3. RRQR - Rank-revealing QR
   - performance of RRQR in HSS on CPU
   - some tweaking to improve performance
   - preliminary results on a GPU

QP3 algorithm (QR with column pivoting)

1. compute column norms
2. `while` more columns to factor `do`

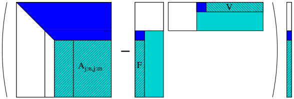
    2.1. panel factorization

    `for` each column in the panel `do`

    a) pivot column with largest norm

    b) update and generate j-th reflector v_j (left-look)

    d) generate j-th column of F with GEMV for $A^T_{j:n,j:m} v_j$

    e) update norms by $A_{j,j:n} := A_{j,j:n} - V_{j,:} F^T_{j:n,1:j}$

    `if` something went wrong `then` break;

    `end for`

    2.2. trailing submatrix update (right-look)

    $$\widehat{A} := \widehat{A} - VF^T$$

    where $F = \widehat{A}^T V T^T$ of $\widehat{A} := (I - VTV^T)\widehat{A} = \widehat{A} - V(\widehat{A}^T V T^T)^T$ in QRF

    recompute norms if needed

    `end while`

QP3 algorithm (QR with column pivoting)

1. compute column norms
2. `while` more columns to factor `do`
   2.1. panel factorization

   `for` each column in the panel `do`

   a) pivot column with largest norm

   b) update and generate j-th reflector v_j (left-look)

   d) generate j-th column of F

   $$F_{j:m,j} = \tau(A^T_{j:n,j:m} - F_{:,1:j-1}V^T_{:,1:j-1})v_j$$

   e) update norms with $A_{j,j:n} := A_{j,j:n} - V_{j,:}F^T_{j:n,1:j}$

   `if` something went wrong `then` break;

   2.2. trailing submatrix update (right-look)

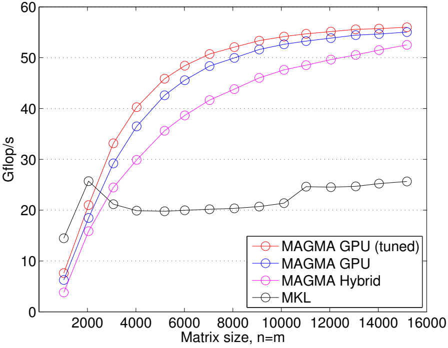
   $$\widehat{A} := \widehat{A} - VF^T \qquad (F = \widehat{A}^T VT^T \text{ of } \widehat{A} := (I - VTV^T)\widehat{A} = \widehat{A} - V(\widehat{A}^T VT^T)^T)$$

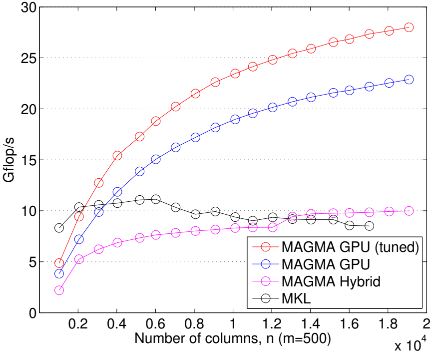
   recompute norms if needed

- about the same number of flops as in a blocked QR
  (i.e., generation of $F \approx$ computation of T and $TV^T\widehat{A}$)

  if not too much norm recomputations
- rich in BLAS-2 operation
  - about 50% in $\widehat{A}^T v_j$, and the other 50% in GEMM for update
- synch to enable global pivoting at each step

QP3 performance on a GPU (bunsen: SandyBridge+Kepler)





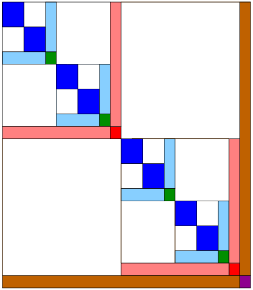
especially for small matrices,

- doing everything on GPU improved performance
  - avoid CPU-GPU communication
- tuning (using specialized kernels) got nice speedups (1.2)
  - merging small kernels, removing error checks, keeping scalars on GPU, etc.

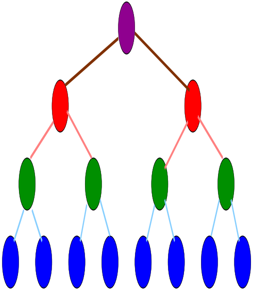
HSS solver for a sparse linear system

“algebraic” HSS preconditioner for a sparse linear system

after a nested-disection, we have



- matrix view -

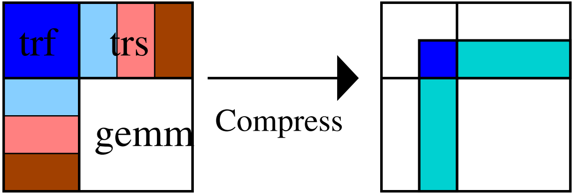


- tree view -

- “eliminateion” tree provides the dependencies among panel factorizations

HSS preconditioner based on a multi-frontal factorization

"compress" interface of a dense frontal matrix at each node of tree

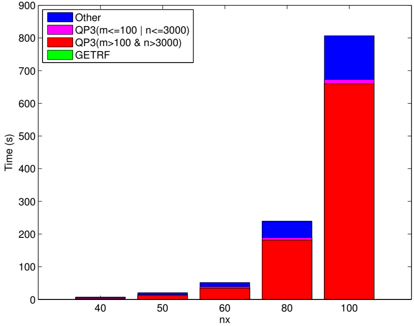


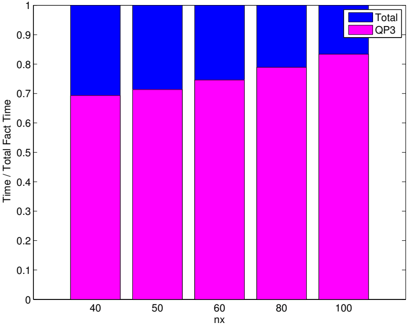
- hierarchically compress by splitting the diagonal block into a smaller binary tree
- use QP3 for the compression
  - the diagonals of R are ordered: $|r_{1,1}| \geq |r_{2,2}| \geq \ldots, |r_{m,m}|$,
    (good estimate of singular values).
  - we truncate QR with a user-specified threshold τ:

    $$AP = QR = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R_{1,1} & R_{1,2} \\ & R_{2,2} \end{pmatrix} \approx Q_1 \begin{pmatrix} R_{1,1} & R_{1,2} \end{pmatrix}$$

    such that $|r_{k+1,k+1}| \leq \tau |r_{1,1}|$ and $R_{11} \in \mathcal{R}^{k \times k}$.
  - we can integrate τ into QP3 to compute a "partial" factorization.

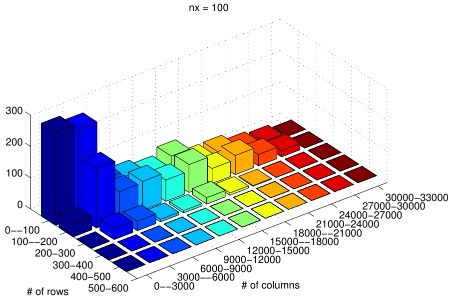
Profiling a serial HSS solver by A. Napov, S. Li, and M. Gu [SIAM PP12]

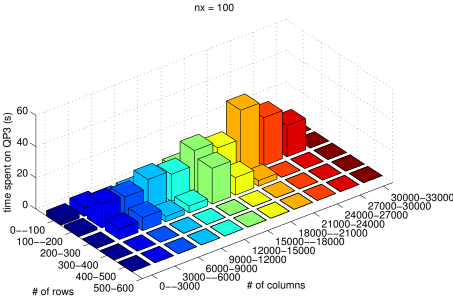




- 7-point 3D poison equation (similar results using other test matrices)
  - the same parameter used in the talk (no cheating)
- HSS spends a lot of time in QP3, especially for a big matrix.

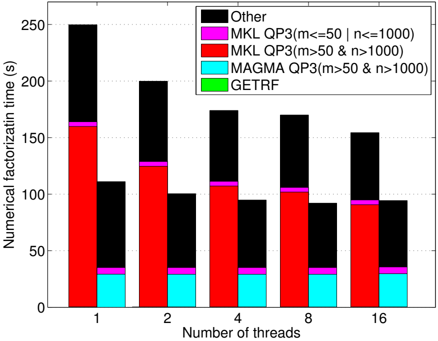
Profiling QP3 in HSS





- 3D poison equation, $n_x = 100$ and $n = 10^6$.
- we have a lot of short-wide QP3s.

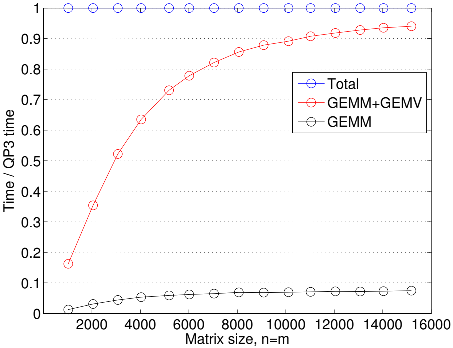
Integrating MAGMA QP3 into HSS (3D poison, $n_x = 100$)



- good speedups using a GPU (speedups of $2.2 - 1.6$)
  - used threashold-version of QP3 on CPU and GPU
  - kept small matrices on CPU
  - used a simple integration of CPU-interface

RRQR on a GPU

QP3 performance on a GPU (bunsen: SandyBridge+Kepler)



- about 50% of flops but over 90% of time on GEMVs with trailing submatrices
- QR without pivoting runs x10 faster (over 600Gflop/s)..

RRQR algorithm

1. factorization

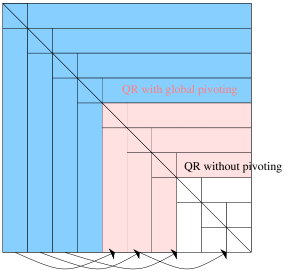
   1.1 QR with restricted pivoting

   - pivots within a window
     1-rank update within a window (right-look),
     and gemm update the rest
   - use condest to guarantee stability
     (i.e., condest$(R_{1,1}) \leq \tau$)
   - delay "numerically unstable" columns to end

   1.2 QR with "global" pivoting

   - use condest to delay unstable columns (column-wise QP3)

   1.3 QR on the "null" columns



2. postprocessing

- since the columns are never removed from Q_1, iterative process is used to guarantee

$$\begin{array}{lll} \text{task-1:} & \text{condest}(R_{1:k,1:k}) & < \tau \\ \text{task-2:} & \text{condest}(R_{1:k+1,1:k+1}) & \geq \tau \end{array}$$

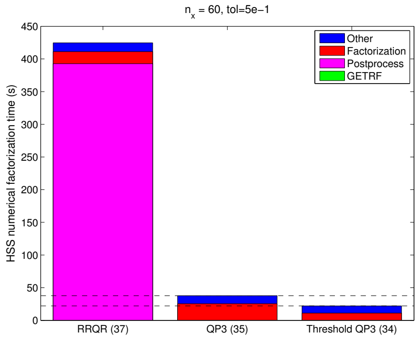
$$AP = QR = \begin{pmatrix} Q_{:,1:k} & Q_{:,k+1:m} \end{pmatrix} \begin{pmatrix} R_{1:k,1:k} & R_{1:k,k+1:n} \\ & R_{k+1:n,k+1:n} \end{pmatrix}$$

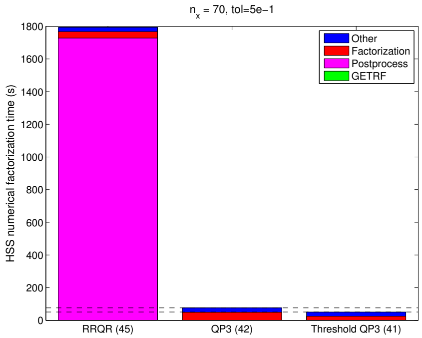
RRQR factorization

"Experimental results on IBM RS/6000 and SGI R8000 platforms show that this approach performs up to **three times faster** than the **less reliable** QR factorization with column pivoting as it is currently implemented in LAPACK, and comes **within 15%** of the performance of the LAPACK block algorithm for computing a QR factorization without any column exchanges."

[C. Bischof, G. Quintana-Orti, '98]

RRQR in HSS





- one core of SandyBridge is used.
- it reduced the factorization time, but the postprocessing is expensive.

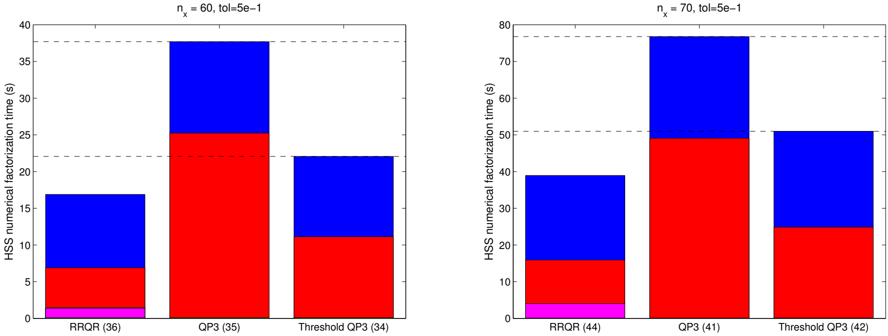
code seems to be designed base on the assumption:

"in most cases the approximate RRQR factorization computed by the block algorithm is very close to the desired RRQR factorization, requiring little postprocessing."

our tweaks to speedup RRQR for HSS on our machine

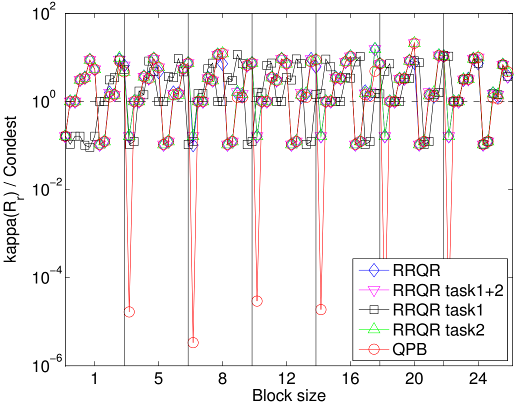
- Preprocessing: reorder columns in descending order of their norms and scale them to have unit-length
- Factorization with restriced pivoting: use more accurate condest
  (condest overestimates acutal condition number)
- Factorization with a global pivoting: use a blocked version
- Postprocessing: update the norms instead of recomputing
  (when submatrix is resized and when orthogoal transformation is applied),
  and use task-1 or task-2 postprocessing.

tweaked RRQR in HSS



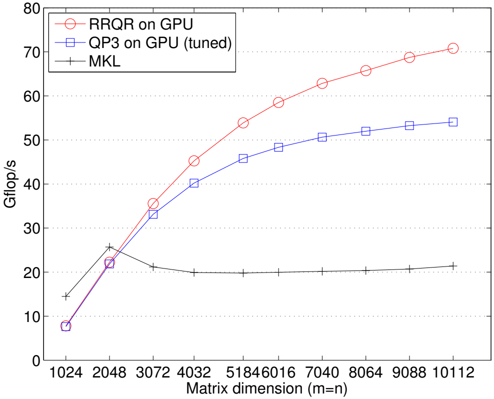
- one core of SandyBridge is used.
- now, RRQR seems to be faster, and should scale better (?)..
  - factorization time is reduced by a factor of two
  - can we do better with more tweaks?

Just to make sure...



- we reran numerical experiments in the paper with tweaked RRQR.
- singular values seem to match..

QR with restricted pivoting on GPU, random square matrices



- got only up to 1.3 speedup. shouldn't it be better?
- look-ahead on a GPU using streams
  - it is not overlapping right now

Our short todo-list

- profile and tune the code, especially for small matrices
  - same as in QP3, and make sure look-ahead is working.
  - tweak more?
- do whole RRQR on GPU by combining with QP3 with threshold and QR
- integrate RRQR into HSS
- move other BLAS kernels of HSS (GEMM) onto a GPU?
- other low-rank algorithms:
  e.g., semi-graded QR + postordering? randomization?