Solving "Large" Dense Matrix Problems on Multi-Core Processors

Mercedes Marqués
Enrique S. Quintana-Ortí
Gregorio Quintana-Ortí
Universidad Jaime I de Castellón (Spain)

Robert A. van de Geijn
The University of Texas at Austin

PDSEC – May, 2009

Motivation

Linear systems $Ax = b$ with $A \rightarrow n \times n$ dense and $n \rightarrow O(10,000 - 100,000)$:

- Estimation of the Earth gravitational field
- BEM in electromagnetics and acoustics
- Molecular dynamic simulations

- Only two years ago, these problems were considered "large" and a cluster was employed for their solution
- Current multi-core processors provide sufficient *computational power* to tackle them...
    - Computational cost is $O(n^3) \rightarrow$ be patient
    - Storage cost is $O(n^2) \rightarrow$ use disk

Motivation

Conventional wisdom against use of disk

- Disk is too slow to feed the processor
- OS can deal with data on disk, but careful design is required to reduce I/O $\longrightarrow$ Out-of-Core (OOC) algorithms
- Programming OOC is cumbersome (e.g., asynchronous I/O)

No longer true!

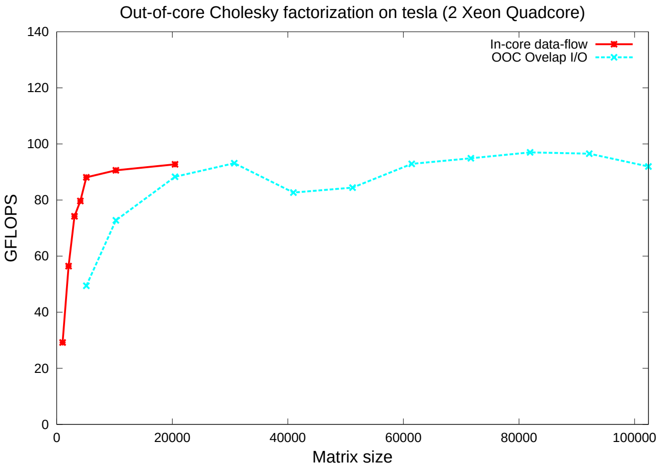
Motivation

Conventional wisdom against use of disk

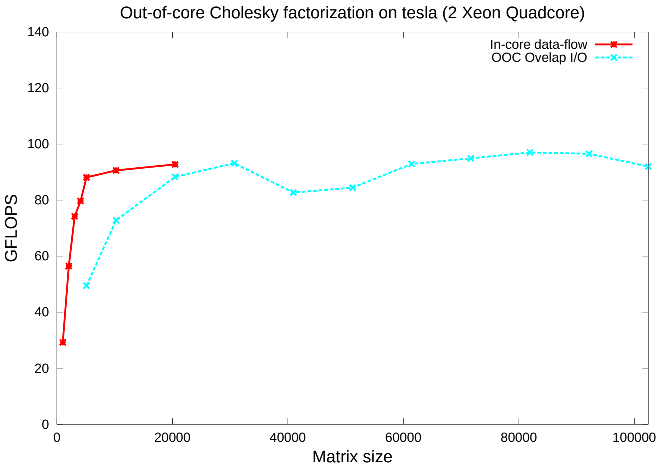
- Disk is too slow to feed the processor
- OS can deal with data on disk, but careful design is required to reduce I/O $\longrightarrow$ Out-of-Core (OOC) algorithms
- Programming OOC is cumbersome (e.g., asynchronous I/O)

No longer true!

A preview ...



Solution of a 100,000×100,000 s.p.d. linear system in 1 hour and 4 minutes!



Solution of a 100,000×100,000 s.p.d. linear system in 1 hour and 4 minutes!

Outline

The Cholesky Factorization

Definition

Given $A \rightarrow n \times n$ symmetric positive definite, compute

$$A = L \cdot L^T,$$

with $L \rightarrow n \times n$ lower triangular

Algorithms should ideally be represented in a way that captures how we reason about them

The Cholesky Factorization

Definition

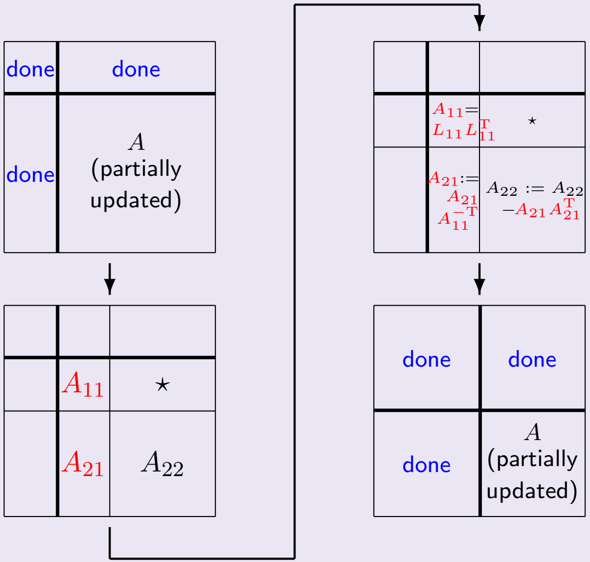
Given $A \rightarrow n \times n$ symmetric positive definite, compute

$$A = L \cdot L^T,$$

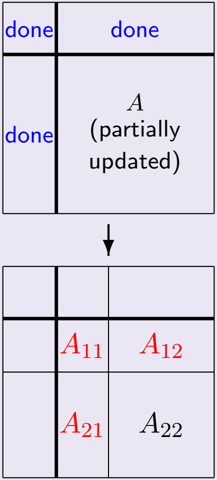
with $L \rightarrow n \times n$ lower triangular

Algorithms should ideally be represented in a way that captures how we reason about them

The Cholesky Factorization: On the Whiteboard



FLAME Notation



Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

FLAME Notation

Algorithm: $[A] := \textsc{Chol\_blk}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0

while $n(A_{BR}) \neq 0$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$$A_{11} = L_{11} L_{11}^{\mathrm{T}}$$
$$A_{21} := A_{21} L_{11}^{-\mathrm{T}}$$
$$A_{22} := A_{22} - A_{21} A_{21}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

FLAME/C Code

From algorithm to code...

FLAME notation

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

FLAME/C representation in code

```
FLA_Repart_2x2_to_3x3(
        ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
     /* ************** */ /* ************************** */
                             &A10, /**/ &A11, &A12,
        ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
        b, b, FLA_BR );
```

FLAME/C Code

```
int FLA_Cholesky_blk( FLA_Obj A, int nb_alg )
{
  /* ... FLA_Part_2x2( ); ... */
  while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) ){
    /* ... */
    FLA_Repart_2x2_to_3x3(
            ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
         /* ************* */   /* ******************** */
                                 &A10, /**/ &A11, &A12,
            ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
            b, b, FLA_BR );

    /*------------------------------------------------------------*/
    FLA_Cholesky_unb( A11 );          /* A21 := Cholesky( A11 )  */
    FLA_Trsm( FLA_ONE, A11,
                         A21 );       /* A21 := A21 * inv( A11 )'*/
    FLA_Syrk( FLA_MINUS_ONE, A21,
                               A22 ); /* A22 := A22 - A21 * A21' */
    /*------------------------------------------------------------*/
    /* ... FLA_Cont_with_3x3_to_2x2( ); ... */
  }
}
```

Outline

Algorithms-by-tiles and algorithms-by-blocks

Matrix of tiles/blocks

$$A \rightarrow \left(\begin{array}{ccccc} A^{(0,0)} & \star & \star & \cdots & \star \\ A^{(1,0)} & A^{(1,1)} & \star & \cdots & \star \\ A^{(2,0)} & A^{(2,1)} & A^{(2,2)} & \cdots & \star \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A^{(N-1,0)} & A^{(N-1,1)} & A^{(N-1,2)} & \cdots & A^{(N-1,N-1)} \end{array} \right)$$

Hierarchical organization

- Each tile is a matrix of blocks
- The tile is the unit of storage on disk
- The block is the unit of computation in the processor

FLAME implementation of algorithm-by-tiles/blocks

```
int FLA_Cholesky_blk( FLA_Obj A, int nb_alg )
{
  /* ... FLA_Part_2x2( ); ... */
  while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) ){
    /* ... */
    /* ... FLA_Repart_2x2_to_3x3( ); ... */

    /*------------------------------------------------------------*/
    FLA_Cholesky_unb( A11 );           /* A21 := Cholesky( A11 )  */
    FLA_Trsm( FLA_ONE, A11,
                         A21 );        /* A21 := A21 * inv( A11 )'*/
    FLA_Syrk( FLA_MINUS_ONE, A21,
                               A22 );  /* A22 := A22 - A21 * A21' */
    /*------------------------------------------------------------*/
    /* ... FLA_Cont_with_3x3_to_2x2( ); ... */
  }
}
```

- Basically no change!

Conventional implementation of OOC algorithm

```
int FLA_Cholesky_blk( FLA_Obj A, int nb_alg )
{
  /* ... FLA_Part_2x2( ); ... */
  while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) ){
    /* ... */
    /* ... FLA_Repart_2x2_to_3x3( ); ... */

    /*------------------------------------------------------------*/
    FLA_Cholesky_unb( A11 );           /* A21 := Cholesky( A11 )  */
    FLA_Trsm( FLA_ONE, A11,
                         A21 );        /* A21 := A21 * inv( A11 )'*/
    FLA_Syrk( FLA_MINUS_ONE, A21,
                               A22 );  /* A22 := A22 - A21 * A21' */
    /*------------------------------------------------------------*/
    /* ... FLA_Cont_with_3x3_to_2x2( ); ... */
  }
}
```

- Insert explicit I/O calls to move data between RAM and disk
- Use large tiles to occupy most of RAM
- Asynchronous I/O?

Unconventional OOC

. . . Conventional may be cumbersome or inefficient ;-)

Software cache

- RAM as a software cache of data on disk
- Algorithms-by-tiles also exploit data locality!
- Implementation:
  - Requires explicit OOC calls to move data from disk to RAM
  - Inside I/O routine, if cache hit perform no I/O
  - Inside I/O routine, when cache is full transfer tile to disk (LRU)

Unconventional OOC: Transparent OOC

Separation of concerns: LA library vs I/O

First stage: build list of tasks

$$
\left|\begin{array}{c|ccc}
A^{(0,0)} & \star & \star & \cdots \\
\hline
A^{(1,0)} & A^{(1,1)} & \star & \cdots \\
A^{(2,0)} & A^{(2,1)} & A^{(2,2)} & \cdots \\
\vdots & \vdots & \vdots & \ddots
\end{array}\right.
\quad \underset{\text{at runtime}}{\longrightarrow}
$$

1. $\text{FLA\_Cholesky\_unb}(A^{(0,0)})$
2. $A^{(1,0)} := A^{(1,0)} \text{ TRIL } \left(A^{(0,0)\,-T}\right)$
3. $A^{(2,0)} := A^{(2,0)} \text{ TRIL } \left(A^{(0,0)\,-T}\right)$
4. $\vdots$
5. $A^{(1,1)} := A^{(1,1)} - A^{(1,0)} A^{(1,0)\,T}$
6. $\vdots$

Unconventional OOC: Transparent OOC

Second stage: transparent I/O and overlap

- Once all tasks are entered on the list, the real execution begins!
- A *scout thread* brings data in-core for tasks in the list $\longrightarrow$ perfect prefetch on software cache
- The *worker threads* execute the tasks with data in-core
- Runtime yields the separation of concerns:
  - OOC transparent to the library developer
  - Asynchronous I/O intrinsic to scout/worker strategy

Outline

Parallelization of In-Core Task

Target: multi-core processor

$$\mathrm{FLA\_Cholesky\_unb}(A^{(k,k)})$$

- Standard parallelization $\longrightarrow$ multithreaded BLAS
- Advanced parallelization $\longrightarrow$ data-flow algorithm

Outline

1. Motivation
2. Cholesky factorization (Overview of FLAME)
3. OOC
4. Parallelization
5. Experimental results
6. Concluding remarks

Porting `libflame`. Experimental results

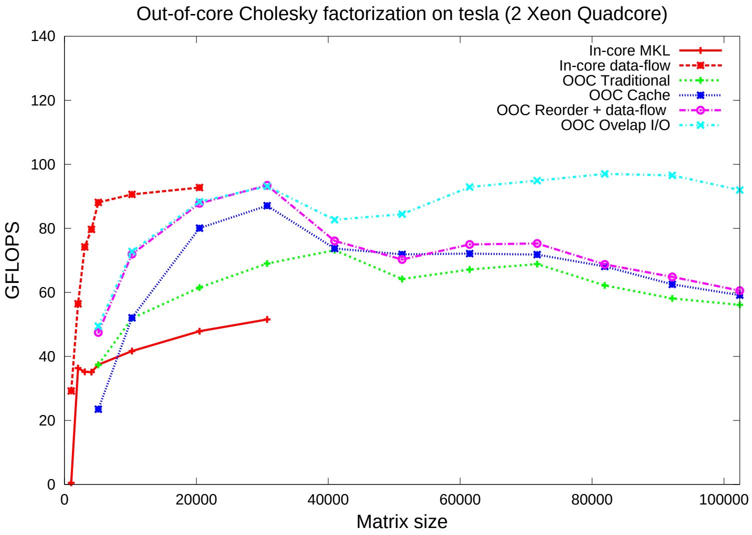
Hardware & Software

- Two Intel Xeon QuadCore E5405, 2.0 GHz, 8 GBytes RAM
- I/O interface of 1.5 Gbites/sec
- SATA-I disk with 160 GBytes
- MKL 10.1, single precision

Algorithms & Implementations

- In-core MKL
- In-core data-flow
- OOC Traditional
- OOC Cache
- OOC Reordered + data-flow
- OOC Overlap I/O

Porting `libflame`. Experimental results



Porting `libflame`. Experimental results

Matrix size	Time	MBytes of required RAM
10,240	4.9sec	400
51,200	8min 49.9sec	10,000
102,400	1h 4min 52.0sec	40,000

Conclusion

Against conventional wisdom

- Disk is fast enough to feed the processor
- Programming OOC is transparent to the library developer
  - Job for the runtime
  - Transparent port of the functionality of `libflame`
- Solving large problems possible using a few multi-core processors

For more information...

Visit `http://www.cs.utexas.edu/users/flame/`