Design of Scalable Dense Linear Algebra Libraries for Multicore Processors and Multi-GPU Platforms

Enrique S. Quintana-Ortí
quintana@icc.uji.es

High Performance Computing & Architectures Group
Universidad Jaime I de Castellón (Spain)

Braunschweig – July, 2008

Joint work with:

Sergio Barrachina	Ernie Chan
Maribel Castillo	Robert van de Geijn
Francisco D. Igual	Field G. Van Zee
Rafael Mayo	
Gregorio Quintana-Ortí	
Rafael Rubio	
Universidad Jaime I (Spain)	The University of Texas at Austin

Supported by:

National Science Foundation (NSF)
Spanish Office of Science
National Instruments
NVIDIA
...

General Motivation

Who has a multicore processor on the desktop/laptop?

Who has a recent graphics card on the desktop/laptop?

General Motivation

Who has a multicore processor on the desktop/laptop?

Are you using more than 1 core?

Who has a recent graphics card on the desktop/laptop?

Are you using it for something else than games? ;-)

Outline

Part I: Multicore processors

Part II: GPUs

Motivation

New dense linear algebra libraries for multicore processors

- Scalability for manycore
- Data locality
- Heterogeneity?

Motivation

LAPACK (*Linear Algebra Package*)

- Fortran-77 codes
- One routine (algorithm) per operation in the library
- Storage in column major order

- Parallelism extracted from calls to multithreaded BLAS

- Extracting parallelism only from BLAS limits the amount of parallelism and, therefore, the scalability of the solution!
- Column major order does hurt data locality

Motivation

FLAME (*Formal Linear Algebra Methods Environment*)

- Libraries of algorithms, not codes
- Notation reflects the algorithm
- APIs to transform algorithms into codes
- Systematic derivation procedure (automated using MATHEMATICA)
- Storage and algorithm are independent

- Parallelism dictated by data dependencies, extracted at execution time
- Storage-by-blocks

Outline

Part I: Multicore processors

1. Motivation
2. Cholesky factorization (Overview of FLAME)
3. Parallelization
4. Other matrix factorizations: LU & QR
5. Experimental results
6. Concluding remarks

Part II: GPUs

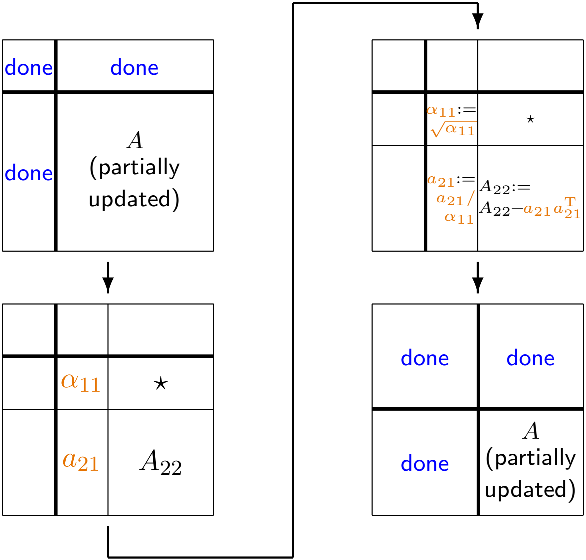
The Cholesky Factorization

Definition

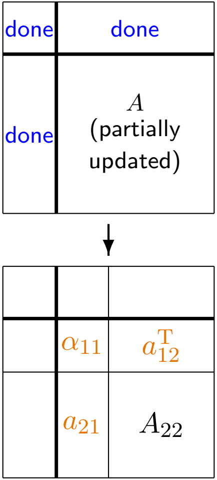
Given $A \rightarrow n \times n$ symmetric positive definite, compute

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

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


done
done
done
A
(partially
updated)
α_{11}
$\star$
a_{21}
A_{22}
$\alpha_{11} := \sqrt{\alpha_{11}}$
$\star$
$a_{21} := a_{21} / \alpha_{11}$
$A_{22} := A_{22} - a_{21} a_{21}^T$
done
done
done
A
(partially
updated)

FLAME Notation



Repartition

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

where α_{11} is a scalar

FLAME Notation

Algorithm: $[A] := \text{CHOL\_UNB}(A)$

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**

Repartition

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

where α_{11} is a scalar

$\alpha_{11} := \sqrt{\alpha_{11}}$

$a_{21} := a_{21} / \alpha_{11}$

$A_{22} := A_{22} - a_{21} a_{21}^T$

Continue with

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

endwhile

FLAME/C Code

From algorithm to code...

FLAME notation

Repartition

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

where α_{11} is a scalar

FLAME/C code

```
FLA_Repart_2x2_to_3x3(
        ATL, /**/ ATR,       &A00,  /**/ &a01,     &A02,
     /* ************* */ /* ************************** */
                             &a10t, /**/ &alpha11, &a12t,
        ABL, /**/ ABR,       &A20,  /**/ &a21,     &A22,
        1, 1, FLA_BR );
```

(Unblocked) FLAME/C Code

```
int FLA_Cholesky_unb( FLA_Obj A )
{
  /* ... FLA_Part_2x2( ); ... */
  while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) ){

    FLA_Repart_2x2_to_3x3(
            ATL, /**/ ATR,       &A00,  /**/ &a01,     &A02,
         /* ************* */   /* ************************** */
                                 &a10t, /**/ &alpha11, &a12t,
            ABL, /**/ ABR,       &A20,  /**/ &a21,     &A22,
            1, 1, FLA_BR );

    /*------------------------------------------------------------*/
    FLA_Sqrt( alpha11 );                /* a21 := sqrt( alpha11 )  */
    FLA_Inv_Scal( alpha11, a21 );       /* a21 := a21 / alpha11    */
    FLA_Syr     ( FLA_MINUS_ONE,
                  a21, A22 );           /* A22 := A22 - a21 * a21t */
    /*------------------------------------------------------------*/
    /* FLA_Cont_with_3x3_to_2x2( ); ... */
  }
}
```

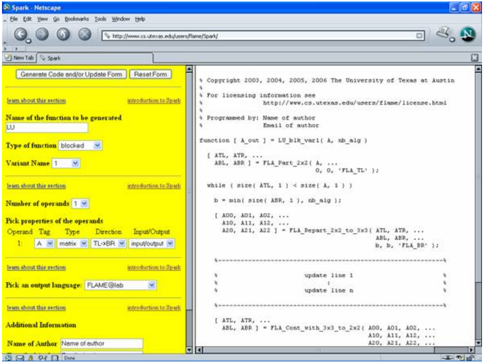
Blocked FLAME/C Code

```
int FLA_Cholesky_blk( FLA_Obj A, int nb_alg )
{
  /* ... FLA_Part_2x2( ); ... */
  while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) ){
    b = min( FLA_Obj_length( ABR ), nb_alg );
    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_rltn( FLA_ONE, A11,
                        A21 );      /* A21 := A21 * inv( A11 )'*/
    FLA_Syrk_ln ( FLA_MINUS_ONE, A21,
                             A22 );/* A22 := A22 - A21 * A21' */
    /*---------------------------------------------------------------*/
    /* FLA_Cont_with_3x3_to_2x2( ); ... */
  }
}
```

FLAME Code

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



- C code: FLAME/C
- M-script code for MATLAB: FLAME@lab
- Other APIs:
  - FLaTeX
  - Fortran-77
  - LabView
  - Message-passing parallel: PLAPACK
  - FLAG: GPUs

Outline

Parallelization on Multithreaded Architectures

LAPACK parallelization: kernels in multithread BLAS

$$\left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array}\right), \quad A_{11} \text{ is } b \times b$$

- Advantage: Use legacy code
- Drawbacks:
  - Each call to BLAS is a synchronization point for threads
  - As the number of threads increases, serial operations with cost $O(nb^2)$ are no longer negligible compared with $O(n^2b)$

Parallelization on Multithreaded Architectures

FLAME parallelization: SuperMatrix

- Traditional (and pipelined) parallelizations are limited by the control dependencies dictated by the code
- The parallelism should be limited only by the data dependencies between operations!
- In dense linear algebra, imitate a superscalar processor: dynamic detection of data dependencies

FLAME Parallelization: SuperMatrix

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

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

The *FLAME runtime system* “pre-executes” the code:

- Whenever a routine is encountered, a pending task is annotated in a global task queue

FLAME Parallelization: SuperMatrix

$$\left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) \quad \overset{\text{Runtime}}{\longrightarrow}$$

1. FLA_Cholesky_unb(A_{00})
2. $A_{10} := A_{10}\ \text{TRIL}\,(A_{00})^{-T}$
3. $A_{20} := A_{20}\ \text{TRIL}\,(A_{00})^{-T}$
4. $A_{11} := A_{11} - A_{10} A_{10}^T$
5. ...

SuperMatrix

- Once all tasks are annotated, the real execution begins!
- Tasks with all input operands available are runnable; other tasks must wait in the global queue
- Upon termination of a task, the corresponding thread updates the list of pending tasks

FLAME Storage-by-Blocks: FLASH

- Algorithm and storage are independent
- Matrices stored by blocks are viewed as matrices of matrices
- No significative modification to the FLAME codes

Outline

1. Motivation
2. Cholesky factorization (Overview of FLAME)
3. Parallelization
4. Other matrix factorizations: LU & QR
5. Experimental results
6. Concluding remarks

Part II: GPUs

LU factorization

- Pivoting for stability limits the amount of parallelism

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right), \quad A_{11} \text{ is } b \times b$$

All operations on A_{22} must wait till $\left(\frac{A_{11}}{A_{21}}\right)$ is factorized

- Algorithms-by-blocks for the Cholesky factorization do not present this problem
- Is it possible to design an algorithm-by-blocks for the LU factorization while maintaining pivoting?

LU factorization with incremental pivoting

$$\left(\begin{array}{c|c|c|c} & & & \\ \hline & A_{11} & A_{12} & A_{13} \\ \hline & A_{21} & A_{22} & A_{23} \\ \hline & A_{31} & A_{32} & A_{33} \end{array}\right), \quad A_{ij} \text{ is } t \times t$$

1. Factorize $P_{11} A_{11} = L_{11} U_{11}$
2. Apply permutation P_{11} and factor L_{11}:

$$L_{11}^{-1} P_{11} A_{12} \;\Big|\; L_{11}^{-1} P_{11} A_{13}$$

3. Factorize $P_{21} \left(\dfrac{A_{11}}{A_{21}} \right) = L_{21} U_{21}$,
4. Apply permutation P_{21} and factor L_{21}:

$$L_{21}^{-1} P_{21} \left(\frac{A_{12}}{A_{22}} \right) \;\Bigg|\; L_{21}^{-1} P_{21} \left(\frac{A_{13}}{A_{23}} \right)$$

5. Repeat steps 2–4 with A_{31}

LU factorization with incremental pivoting

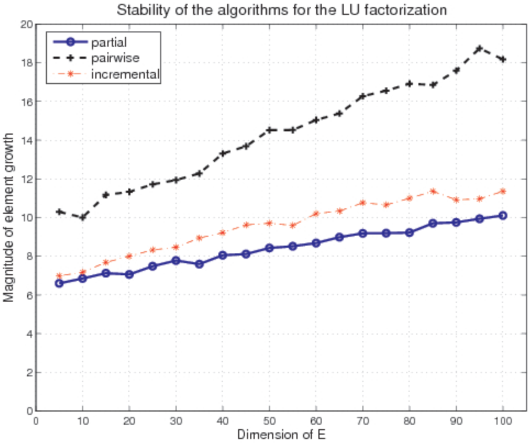
$$\left(\begin{array}{c|c|c|c}
 & & & \\ \hline
 & A_{11} & A_{12} & A_{13} \\ \hline
 & A_{21} & A_{22} & A_{23} \\ \hline
 & A_{31} & A_{32} & A_{33}
\end{array}\right), \quad A_{ij} \text{ is } t \times t$$

Different from LU factorization with column pivoting

- To preserve structure, permutations only applied to blocks on the right!
- To obtain high performance a blocked algorithm with block size $b \ll t$, is used in the factorization and application of factors
- To maintain the computational cost, the upper triangular structure of A_{11} is exploited during the factorization

LU factorization with incremental pivoting

Stability? Element growth with random matrices:



QR factorization

- Same problem as with LU: proceeding by blocks of columns limits the amount of parallelism

$$\left(\begin{array}{c|c|c} \hline & A_{11} & A_{12} \\ \hline & A_{21} & A_{22} \end{array}\right), \quad A_{11} \text{ is } b \times b$$

All operations on A_{22} must wait till $\left(\frac{A_{11}}{A_{21}}\right)$ is factorized

- Is it possible to design an algorithm-by-blocks for the QR factorization while maintaining pivoting?

QR factorization by blocks

$$\left(\begin{array}{c|c|c|c} & & & \\ \hline & A_{11} & A_{12} & A_{13} \\ \hline & A_{21} & A_{22} & A_{23} \\ \hline & A_{31} & A_{32} & A_{33} \end{array}\right), \quad A_{ij} \text{ is } t \times t$$

1. Factorize $Q_{11} A_{11} = R_{11}$
2. Apply factor Q_{11}:
$$Q_{11}^{\mathrm{T}} A_{12} \mid Q_{11}^{\mathrm{T}} A_{13}$$
3. Factorize $Q_{21} \left(\frac{A_{11}}{A_{21}}\right) = R_{21}$,
4. Apply factor Q_{21}:
$$Q_{21}^{\mathrm{T}} \left(\frac{A_{12}}{A_{22}}\right) \Bigg| \; Q_{21}^{\mathrm{T}} \left(\frac{A_{13}}{A_{23}}\right)$$
5. Repeat steps 2–4 with A_{31}

Outline

Experimental Results

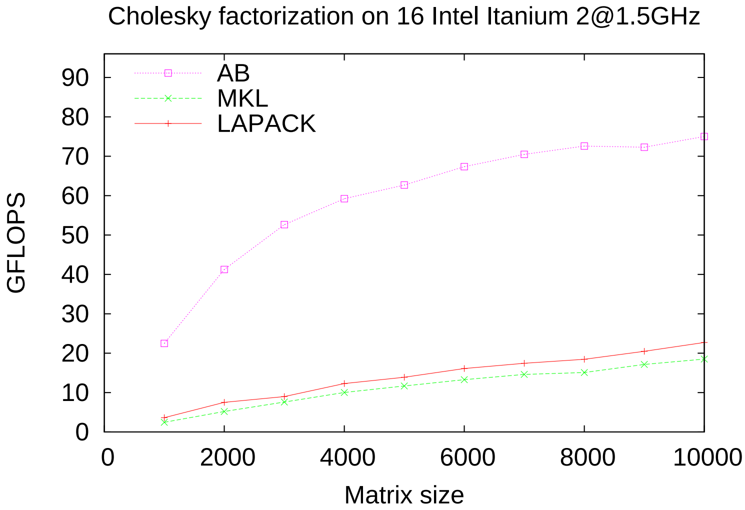
General

Platform	Specs.
SET	CC-NUMA with 16 Intel Itanium-2 processors
NEUMANN	SMP with 8 dual-core Intel Pentium4 processors

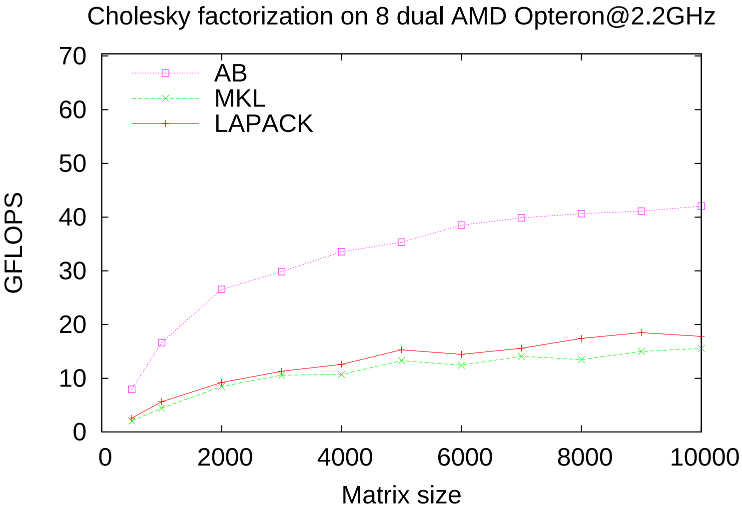
Implementations

- LAPACK 3.0 routine + multithreaded MKL
- Multithreaded routine in MKL
- AB + serial MKL
- AB + serial MKL + storage-by-blocks

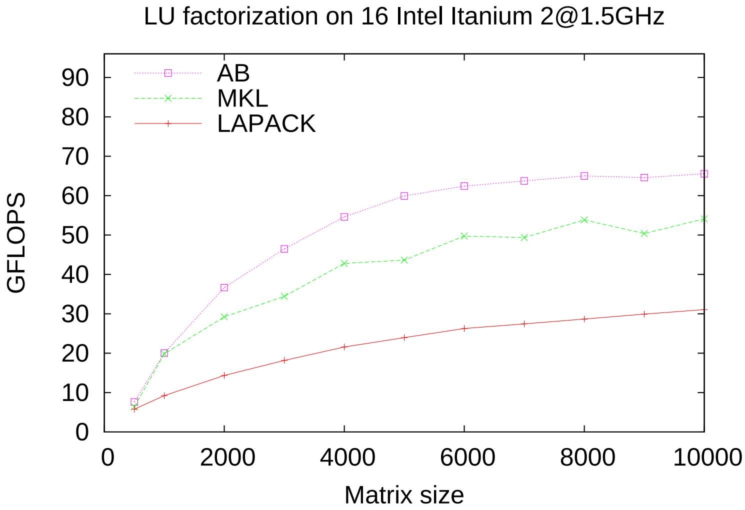
Experimental Results



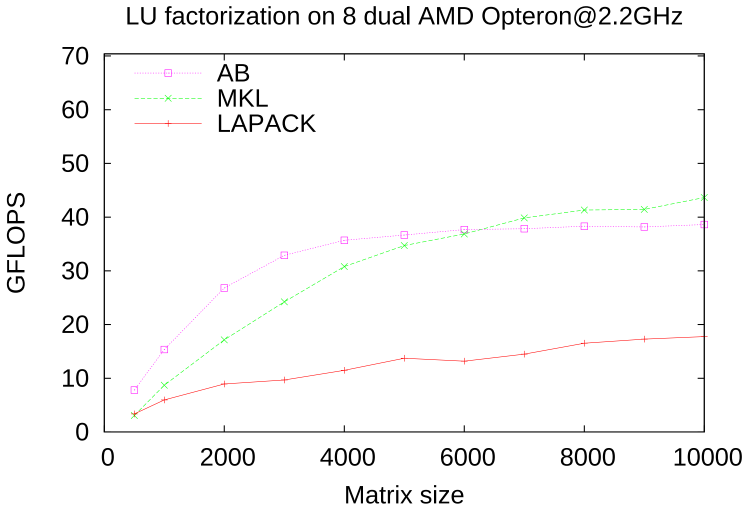
Experimental Results



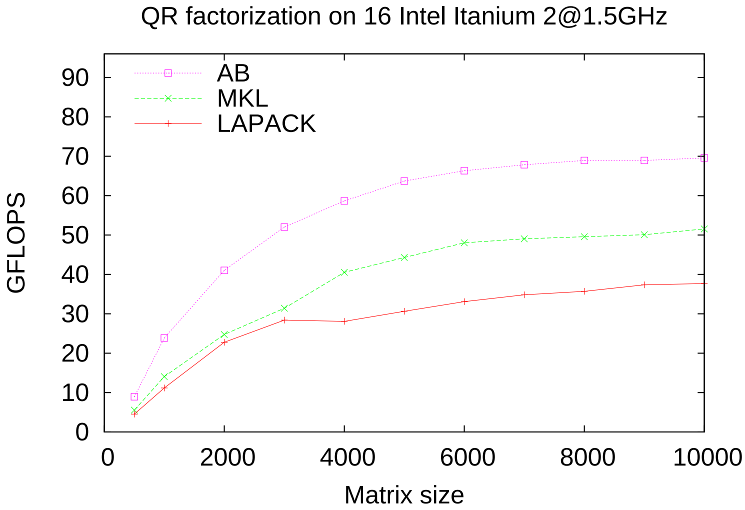
Experimental Results



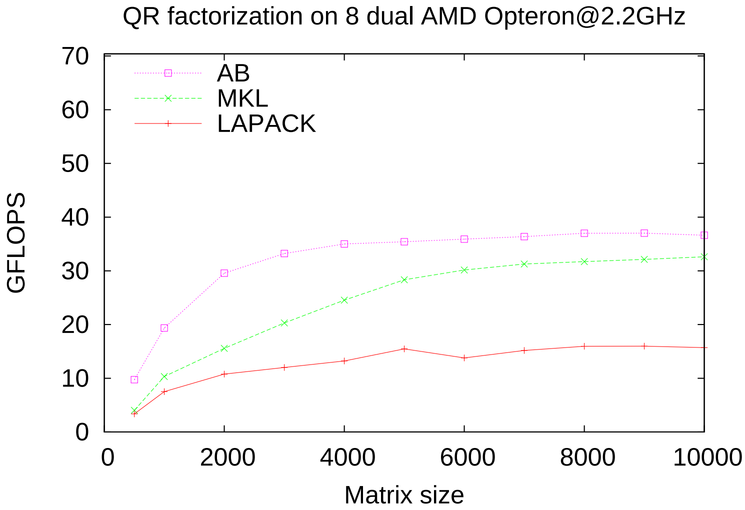
Experimental Results



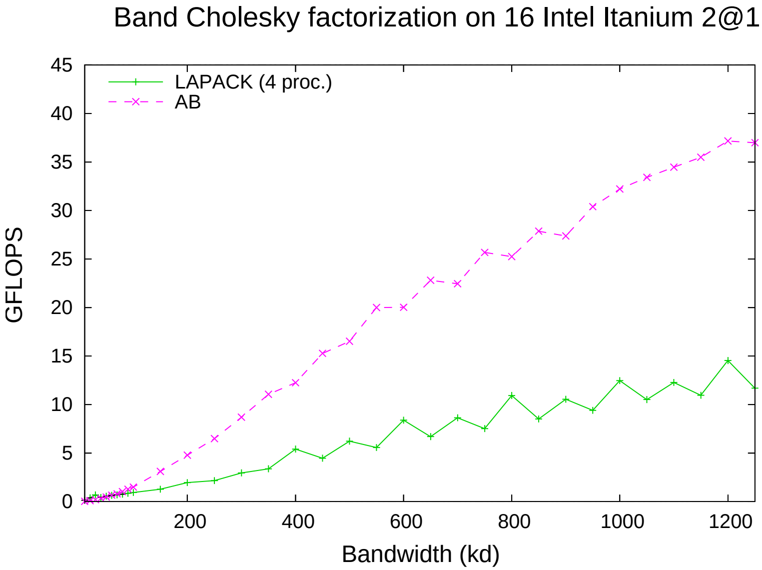
Experimental Results



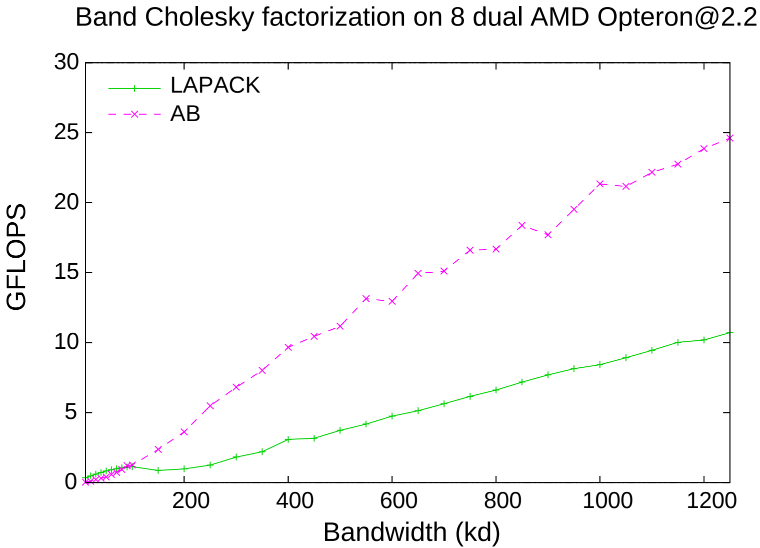
Experimental Results



Experimental Results



Experimental Results



Outline

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

Part II: GPUs

Concluding Remarks

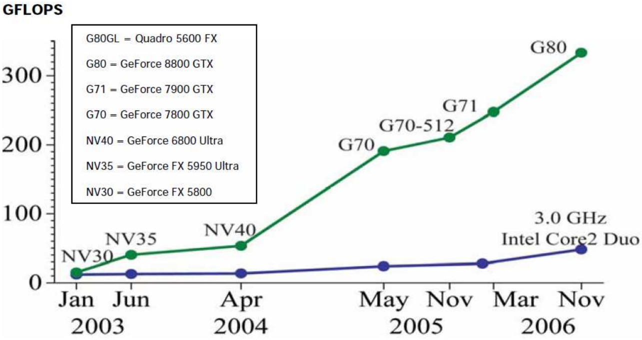
- More parallelism is needed to deal with the large number of cores of future architectures and data locality issued: traditional dense linear algebra libraries will have to be rewritten
- Some operations require new algorithms to better expose parallelism: LU with incremental pivoting, tiled QR,. . .
- The FLAME infrastructure (FLAME/C API, FLASH, and SuperMatrix) reduces the time to take an algorithm from whiteboard to high-performance parallel implementation

Outline

Part I: Multicore processors

Part II: GPUs

Motivation



The power and versatility of modern GPU have transformed them into the first widely extended HPC platform

Outline

Part I: Multicore processors

Part II: GPUs

Outline

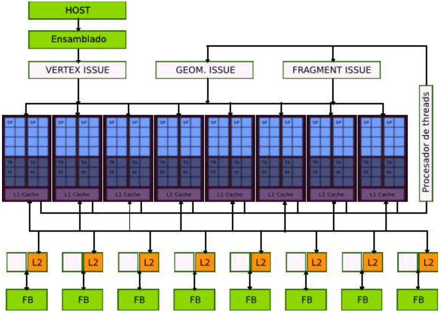
Part I: Multicore processors

Part II: GPUs

1. Motivation
2. Introduction
3. LAPACK on 1 GPU
4. LAPACK on multiple GPUs
5. FLAG@lab
6. Concluding remarks

CUDA Hardware

- A CUDA-enabled device is seen as a coprocessor to the CPU, capable of executing a very high number of threads in parallel
- Example: nVIDIA G80 as a set of SIMD Multiprocessors with On-Chip Shared Memory



- Up to 128 *Streaming Processors* (SP), grouped in clusters
- SP are SIMD processors
- Small and fast Shared Memory shared per SP cluster
- Local 32-bit registers per processor

CUDA Software

- The CUDA API provides a simple framework for writing C programs for execution on the GPU
- Consists of:
  - A minimal set of extensions to the C language
  - A runtime library of routines for controlling the transfers between video and main memory, run-time configuration, execution of device-specific functions, handling multiple GPUs,. . .

CUDA libraries

On top of CUDA, nVIDIA provides two optimized libraries: CUFFT and CUBLAS

CUBLAS Example

```
int main( void ){
  ...
  float* h_vector, * d_vector;

  h_vector = (float*)malloc(M*sizeof(float));

  ...
  cublasAlloc(M, sizeof(float),
              (void**) &d_vector);

  cublasSetVector(M, sizeof(float), h_vector,
                  d_vector, 1);
  cublasSscal(M, ALPHA, d_vector, 1);
  cublasGetVector(M, sizeof(float), d_vector,
                  h_vector, 1);

  cublasFree(d_vector);
  ...
}
```

A typical CUDA (and CUBLAS) program has 3 phases:

1. Allocation and transfer of data to GPU
2. Execution of the BLAS kernel
3. Transfer of results back to main memory

Outline

Part I: Multicore processors

Part II: GPUs

1. Motivation
2. Introduction
3. LAPACK on 1 GPU
4. LAPACK on multiple GPUs
5. FLAG@lab
6. Concluding remarks

Cholesky factorization. Blocked variants

Algorithm: $A := \text{CHOL\_BLK}(A)$

Partition ...
  where ...
while $m(A_{TL}) < m(A)$ **do**
  Determine block size b
  Repartition

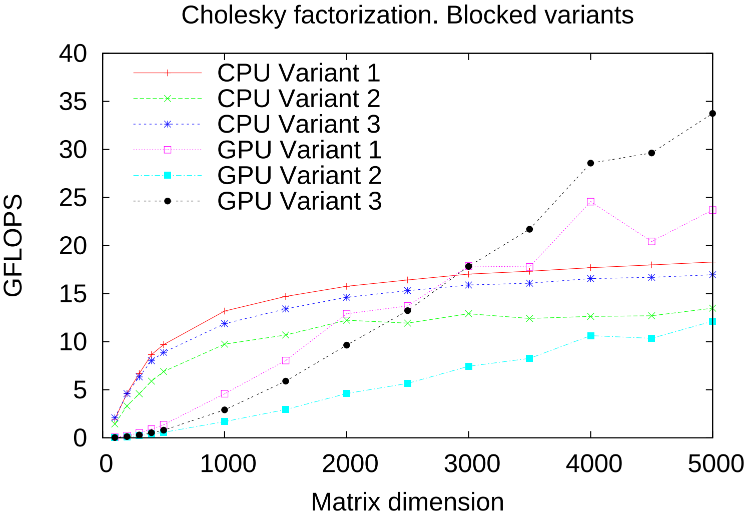
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

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

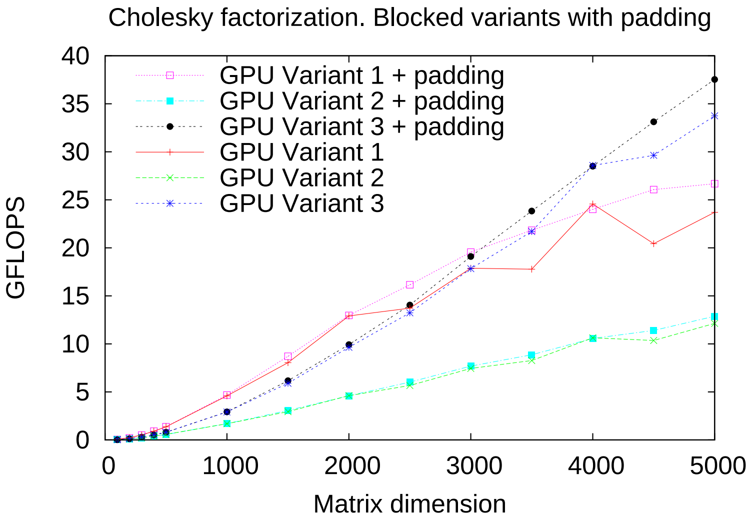
Variant 1:	Variant 2:	Variant 3:
$A_{11} := \text{CHOL\_UNB}(A_{11})$	$A_{10} := A_{10}\,\text{TRIL}\,(A_{00})^{-T}$	$A_{11} := A_{11} - A_{10}A_{10}^T$
$A_{21} := A_{21}\,\text{TRIL}\,(A_{11})^{-T}$	$A_{11} := A_{11} - A_{10}A_{10}^T$	$A_{11} := \text{CHOL\_UNB}(A_{11})$
$A_{22} := A_{22} - A_{21}A_{21}^T$	$A_{11} := \text{CHOL\_UNB}(A_{11})$	$A_{21} := A_{21} - A_{20}A_{10}^T$
		$A_{21} := A_{21}\,\text{TRIL}\,(A_{11})^{-T}$

  Continue with
   ...
endwhile

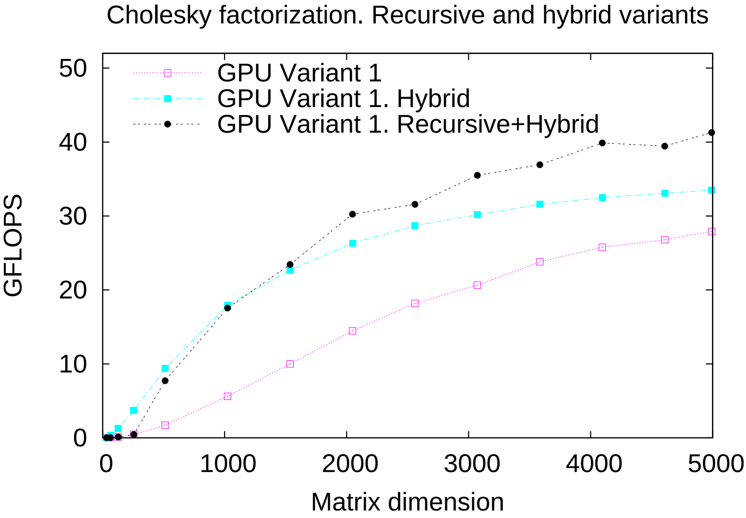
Cholesky factorization. Experimental results



Cholesky padding. Experimental results



Cholesky hybrid and recursive. Experimental results



Iterative refinement for extended precision

Compute the Cholesky factorization $A = LL^T$ and solve $(LL^T)x = b$ in the GPU

$\rightarrow$ 32 bits of accuracy!

$i \leftarrow 1$

repeat

$r^{(i)} \leftarrow b - A \cdot x^{(i)}$

$r^{(i)}_{(32)} \leftarrow r^{(i)}$

$z^{(i)}_{(32)} \leftarrow L^{-T}_{(32)}(L^{-1}_{(32)} r^{(i)}_{(32)})$

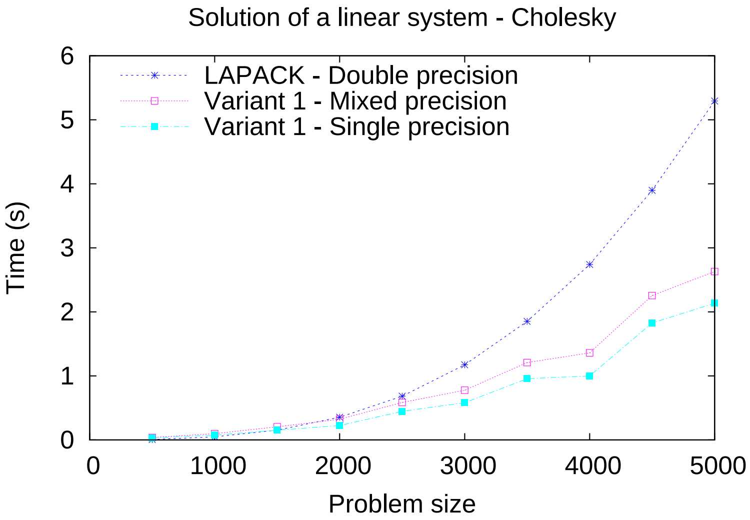
$z^{(i)} \leftarrow z^{(i)}_{(32)}$

$x^{(i+1)} \leftarrow x^{(i)} + z^{(i)}$

$i \leftarrow i + 1$

until $\|r^{(i)}\| < \sqrt{\varepsilon}\|x^{(i)}\|$

Iterative refinement. Experimental results



Outline

What if multiple GPUs are available?

Already here:

- Multiple ClearSpeed boards
- Multiple NVIDIA cards
- nVIDIA Tesla series

How are we going to program these?

What if multiple GPUs are available?

Already here:

- Multiple ClearSpeed boards
- Multiple NVIDIA cards
- nVIDIA Tesla series

How are we going to program these?

Porting SuperMatrix to multiple GPUs

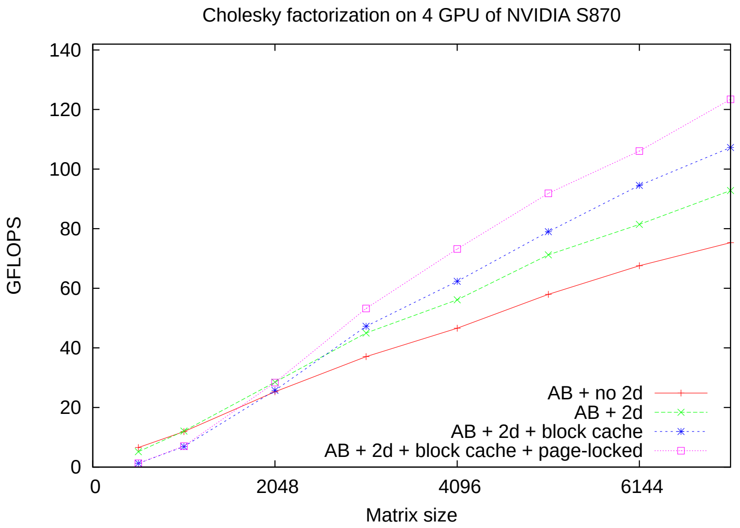
- Employ the equivalence: 1 core $\equiv$ 1 GPU
- Difference: Transference from RAM to video memory
- Run-time system (scheduling), storage, and code are independent
- No significative modification to the FLAME codes: Interfacing to CUBLAS

A software effort of two hours!

Experimental setup

	CPU	GPU
Processor	16 x Intel Itanium2	NVIDIA Tesla s870 (4 G80)
Clock frequency	1.5 GHz	575 MHz

Porting SuperMatrix. Experimental results

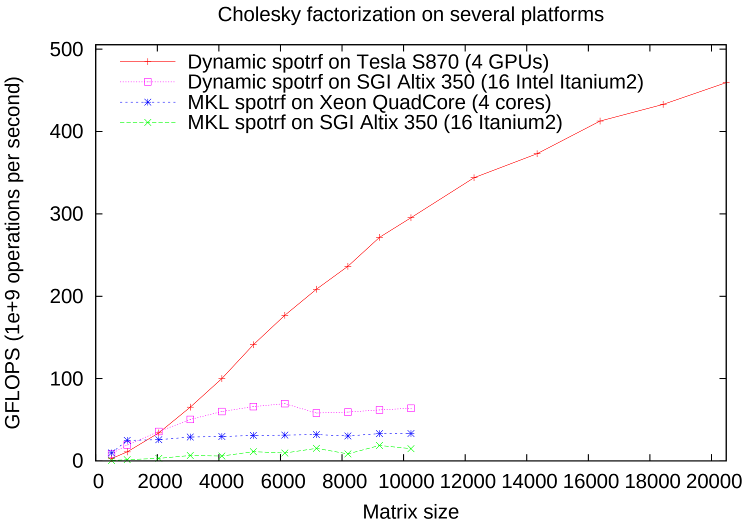


Porting SuperMatrix. Experimental results

A more elaborate port required for high-performance:

- 2-D work distribution
- Memory/cache coherence techniques to reduce transferences between RAM and video memory: write-back and write-invalidate

Porting SuperMatrix. Experimental results



Outline

FLAG@lab

Assorted flavours:

- FLAG: A M-script API for GPU computing from MATLAB/OCTAVE
- FLAGOOC: A M-script API for Out-of-Core GPU computing from MATLAB/OCTAVE

High-level APIs: FLAG

```
function [ A ] = FLAG_Chol( A )

  % [ ... ] = FLAG_Part_2x2( ... );

  while ( FLAG_Obj_length( ATL, 1 ) < FLAG_Obj_length( A, 1 ) )
    b = min( FLAG_Obj_length( ABR, 1 ), nb_alg );

    % [ ... ] = FLAG_Repart_2x2_to_3x3( ... );
    %------------------------------------------------------------%
    FLAG_Chol_unb( A11 );       % A11 = Chol( A11 )
    FLAG_Trsm_rltn( 1.0, A11,
                         A21 ); % A21 = A21 * tril( A11 )^-T
    FLAG_Syrk_ln( -1.0, A21,
                   1.0, A22 );  % A22 = A22 - A21 * A21';
    %------------------------------------------------------------%
    % [ ... ] = FLAG_Cont_with_3x3_to_2x2( ... );
  end
```

Just replace FLA_ in FLAME@lab by FLAG_!

Concluding Remarks

- Simple precision may not be enough. Double precision is coming, but at the expense of speed?
- Overlap transferences and computation is also needed (close?)
- Programming dense linear algebra using CUBLAS on NVIDIA hardware is easy
- Programming at CUDA level?

  I'll need to ask my student Francisco...

Concluding Remarks

- Simple precision may not be enough. Double precision is coming, but at the expense of speed?
- Overlap transferences and computation is also needed (close?)
- Programming dense linear algebra using CUBLAS on NVIDIA hardware is easy
- Programming at CUDA level?
  I'll need to ask my student Francisco...

Related Publications

- E. Chan, E.S. Quintana-Ortí, G. Quintana-Ortí, R. van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multicore architectures. *19th ACM Symp. on Parallelism in Algorithms and Architectures – SPAA'2007.*
- E. Chan, F. Van Zee, R. van de Geijn, E.S. Quintana-Ortí, G. Quintana-Ortí. Satisfying your dependencies with SuperMatrix. *IEEE Cluster 2007*.
- E. Chan, F.G. Van Zee, P. Bientinesi, E.S. Quintana-Ortí, G. Quintana-Ortí, R. van de Geijn. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. *Principles and Practices of Parallel Programming – PPoPP'2008.*
- E.S. Quintana-Ortí, R. van de Geijn. Updating an LU factorization with pivoting. *ACM Trans. on Mathematical Software, 2008.*

Related Publications

- S. Barrachina, M. Castillo, Francisco D. Igual, R. Mayo, E. S. Quintana-Ort´ i. Evaluation and tuning of the level 3 CUBLAS for graphics processors. *Workshop on Parallel and Distributed Scientific and Engineering Computing, – PDSEC'2008.*
- S. Barrachina, M. Castillo, F. Igual, R. Mayo, E. S. Quintana. Solving dense linear systems on graphics processors. *Euro-Par'2008.*
- M. Castillo, F. Igual, R. Mayo, R. Rubio, E. S. Quintana, G. Quintana, R. van de Geijn. Out-of-Core Solution of Linear Systems on Graphics Processors. *Parallel/High-Performance Object-Oriented Scientific Computing – POOSC'08 .*

Related Approaches

Cilk (MIT) and CellSs (Barcelona SuperComputing Center)

- General-purpose parallel programming
  - Cilk → irregular problems
  - CellSs → for the Cell B.E.
- High-level language based on OpenMP-like pramas + compiler + runtime system
- Moderate results for dense linear algebra

PLASMA (UTK – Jack Dongarra)

- Traditional style of implementing algorithms: Fortran-77
- Complicated coding
- Runtime system + ?

For more information...

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

Support...

- *National Science Foundation* awards CCF-0702714 and CCF-0540926 (ongoing till 2010)
- Spanish CICYT project TIN2005-09037-C02-02