Solving Dense Linear Systems on Platforms with Multiple Hardware Accelerators

Francisco D. Igual
Enrique S. Quintana-Ortí
Gregorio Quintana-Ortí

Universidad Jaime I de Castellón (Spain)

Robert A. van de Geijn

The University of Texas at Austin

PPoPP09 – February, 2009

Joint work with:

Universidad Jaime I (Spain)	UT-Austin
Sergio Barrachina	Ernie Chan
Maribel Castillo	Robert A. van de Geijn
Francisco D. Igual	Field G. Van Zee
Rafael Mayo	
Enrique S. Quintana-Ortí	
Gregorio Quintana-Ortí	
Rafael Rubio	

Supported by:

National Science Foundation (NSF)
Spanish Office of Science
nVIDIA
Anonymous corporation

Motivation

The sky is falling

- Parallelism is thrust upon the masses
- Popular libraries like LAPACK must be completely rewritten
- Give us money or the world as we know it will come to an end

Motivation

Evolution vs intelligent design

- Parallelism is thrust upon the masses
- Popular libraries like LAPACK must be completely rewritten (end of an evolutionary path)
- Great, let's finally toss them and start over
- Cheaper than trying to evolve?

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

Problems

- Extracting parallelism increases synchronization and thus limits performance
- Column major order hurts data locality

Motivation

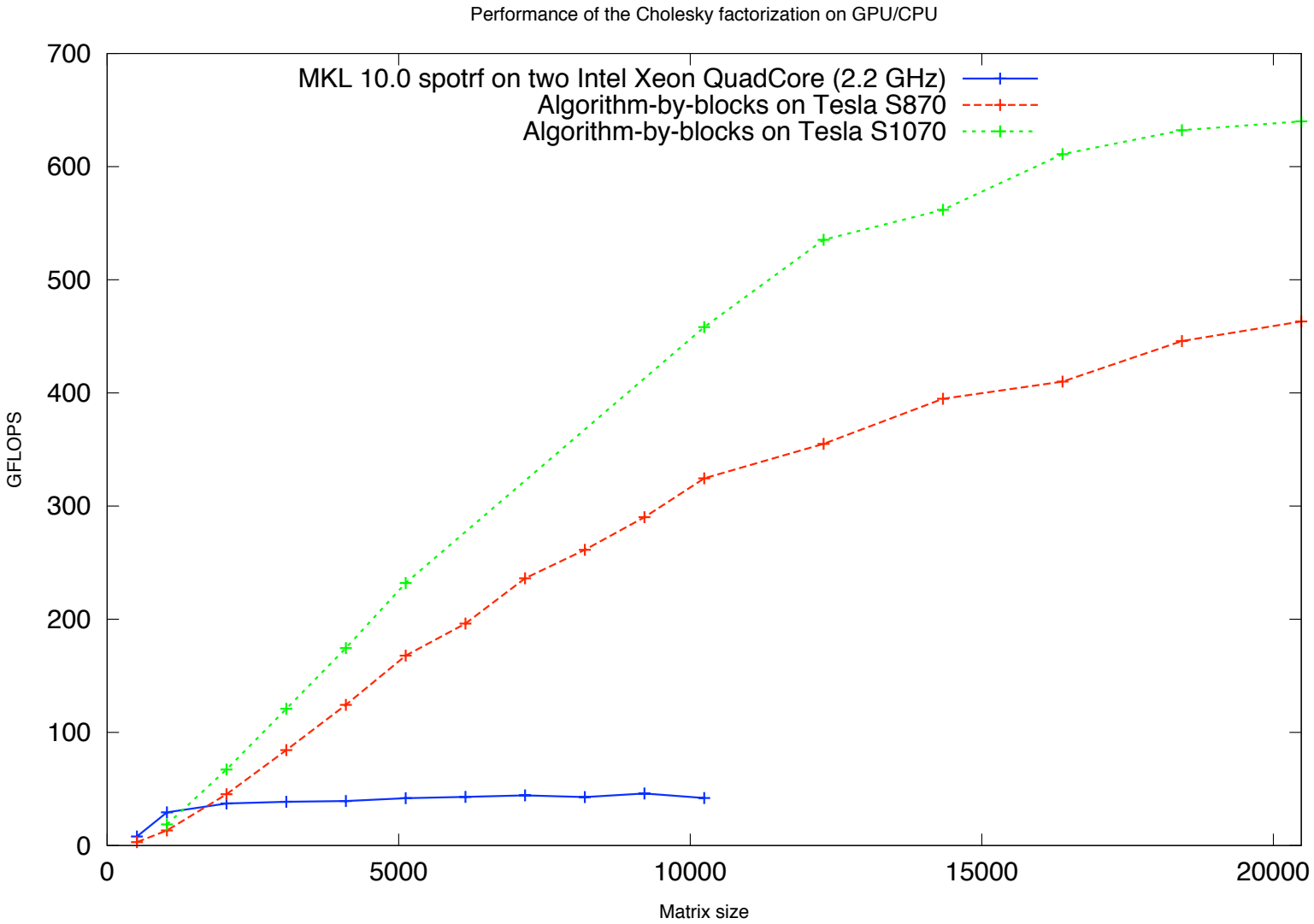
FLAME (*Formal Linear Algebra Methods Environment*)

- Notation for expressing algorithms
- Systematic derivation procedure (automated using MATHEMATICA)
- Families of algorithms for each operation
- APIs to transform algorithms into codes
- Storage and algorithm are independent

We will show FLAME elegantly supports

- Storage-by-blocks
- Parallelism with data dependencies
- High performance even on "exotic" architectures like multiGPUs

A preview ...


Performance of the Cholesky factorization on GPU/CPU
MKL 10.0 spotrf on two Intel Xeon QuadCore (2.2 GHz)
Algorithm-by-blocks on Tesla S870
Algorithm-by-blocks on Tesla S1070
GFLOPS
Matrix size
0
100
200
300
400
500
600
700
0
5000
10000
15000
20000

Outline

1. Motivation
2. Cholesky factorization (Overview of FLAME)
3. Parallelization
4. Targeting platforms with multiple accelerators
5. Experimental results
6. Concluding remarks

The Cholesky Factorization

Definition

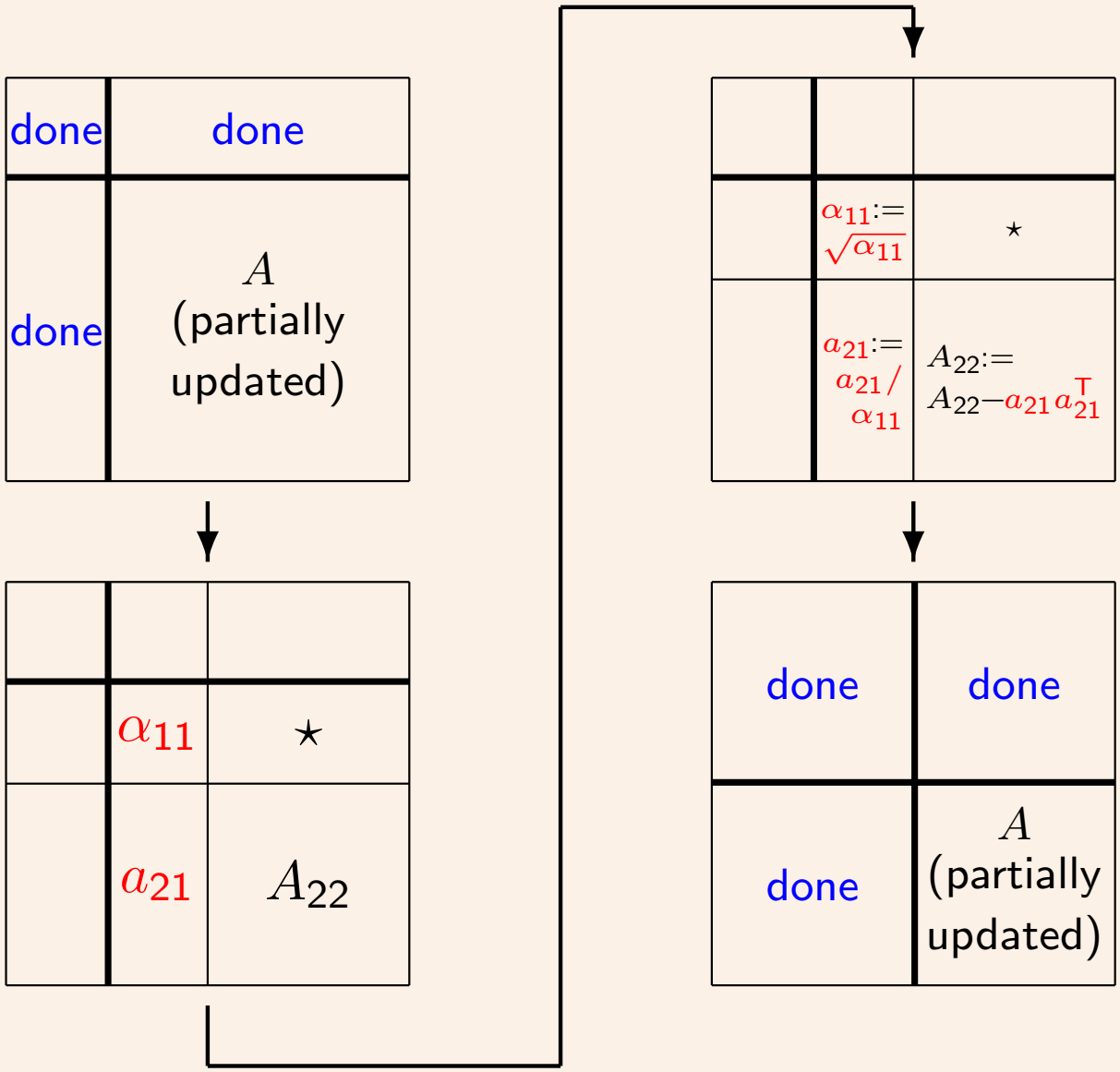
Given $A \rightarrow n \times n$ symmetric positive definite, compute

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

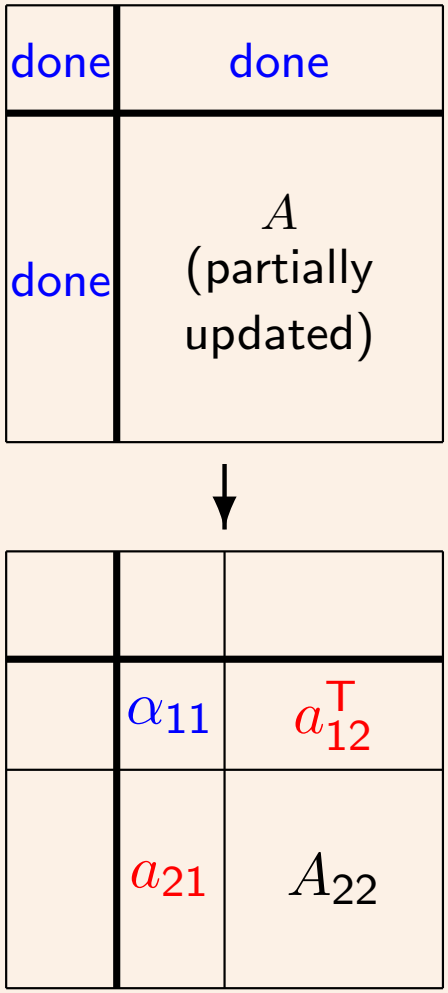
with $L \rightarrow n \times n$ lower triangular

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

The Cholesky Factorization: On the Whiteboard



FLAME Notation



Repartition

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

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

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 representation in 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

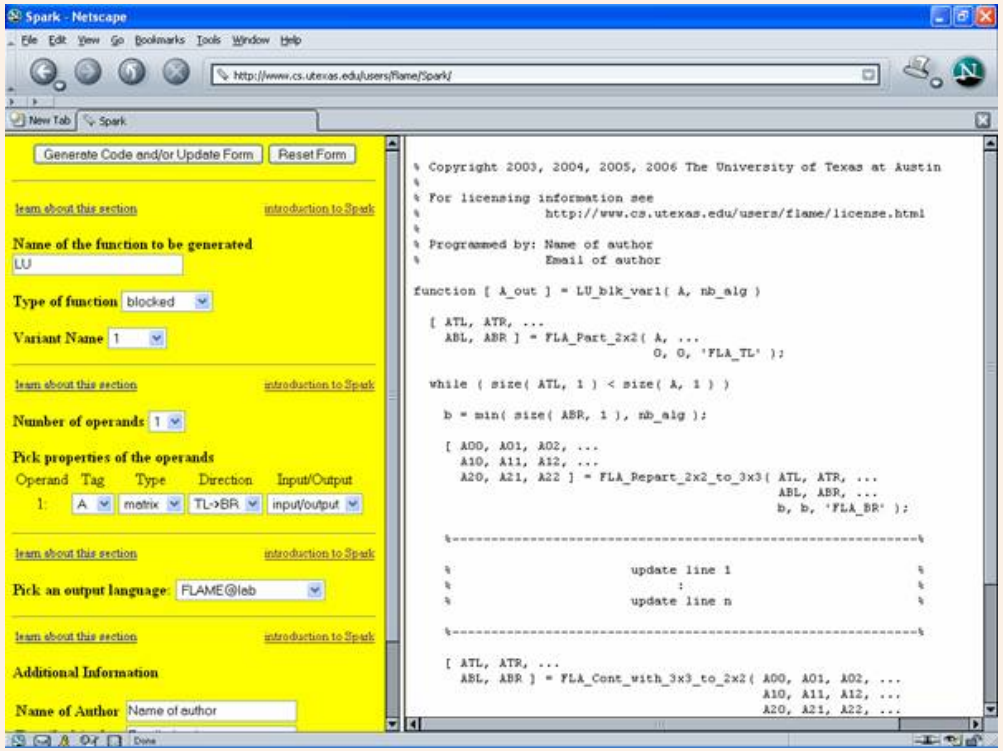
```
int FLA_Cholesky_unb( FLA_Obj A )
{
  /* ... FLA_Part_2x2( ); ... */
  while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) ){

    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

1. Motivation
2. Cholesky factorization (Overview of FLAME)
3. Parallelization
4. Targeting platforms with multiple accelerators
5. Experimental results
6. Concluding remarks

LAPACK parallelization: multithreaded BLAS

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

- Pro?:
  - Evolve legacy code
- Con:
  - Continue to code in the LINPACK style (1970s)
  - Each call to BLAS (compute kernels) 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 $O(n^2b)$

Separation of concern

- Improve parallelism and data locality: algorithms-by-blocks
  - Matrix of matrix blocks
  - Matrix blocks as unit of data
  - Computation with matrix blocks as unit of computation
- Execute sequential code to generate DAG of tasks
- SuperMatrix
  - Runtime system for scheduling tasks to threads
- Sequential kernels to be executed by the threads
  - Always be sure to make the machine-specific part someone else's problem

Matrix of blocks

$$A = \begin{pmatrix} A^{(0,0)} & \star & \star & \cdots & \star \\ A^{(1,0)} & A^{(1,1)} & \star & \cdots & \star \\ A^{(2,0)} & A^{(2,1)} & A^{(2,2)} & \cdots & \star \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A^{(M-1,0)} & A^{(M-1,1)} & A^{(M-1,2)} & \cdots & A^{(M-1,N-1)} \end{pmatrix}$$

FLAME implementation of algorithm-by-blocks: no change

```
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

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

$$\left(\begin{array}{c|c|ccc} A^{(0,0)} & \star & \star & \cdots & \star \\ \hline A^{(1,0)} & A^{(1,1)} & \star & \cdots & \star \\ \hline A^{(2,0)} & A^{(2,1)} & A^{(2,2)} & \cdots & \star \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A^{(M-1,0)} & A^{(M-1,1)} & A^{(M-1,2)} & \cdots & A^{(M-1,N-1)} \end{array}\right)$$

FLAME Parallelization: SuperMatrix

$A^{(0,0)}$	$\star$	$\star$	$\cdots$
$A^{(1,0)}$	$A^{(1,1)}$	$\star$	$\cdots$
$A^{(2,0)}$	$A^{(2,1)}$	$A^{(2,2)}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

$\longrightarrow$ at runtime build DAG

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

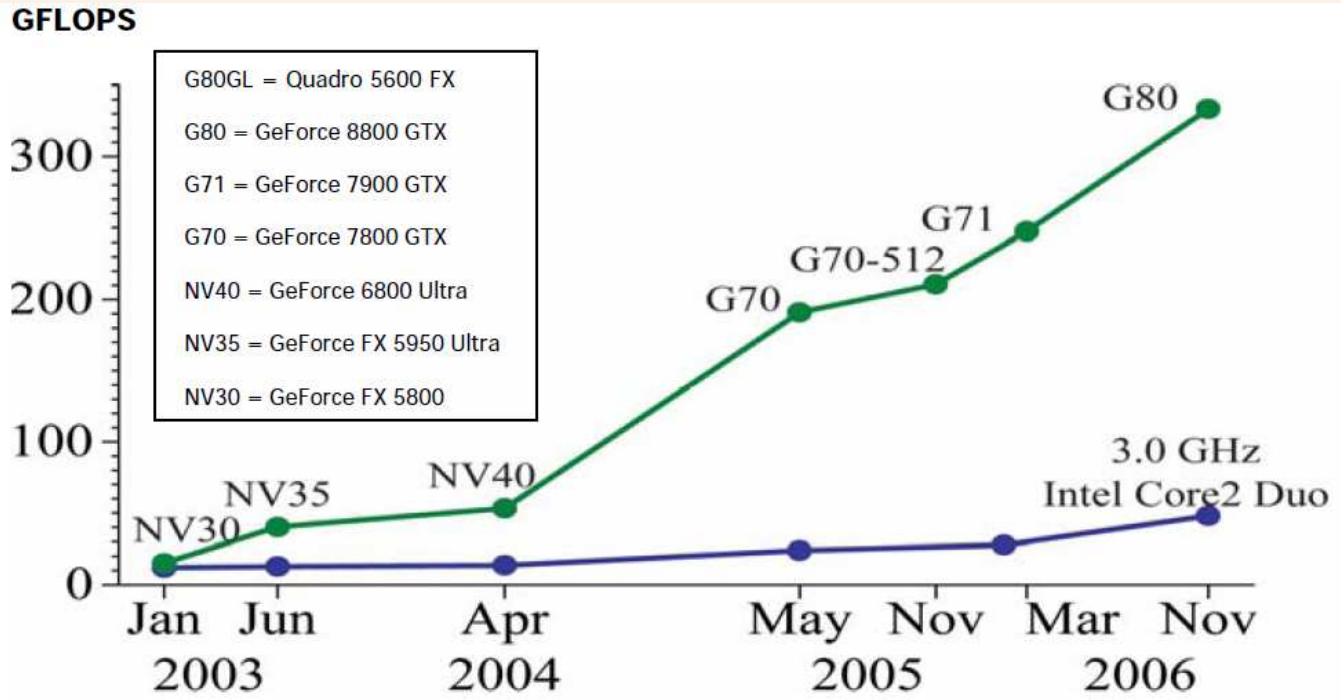
SuperMatrix

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

Outline

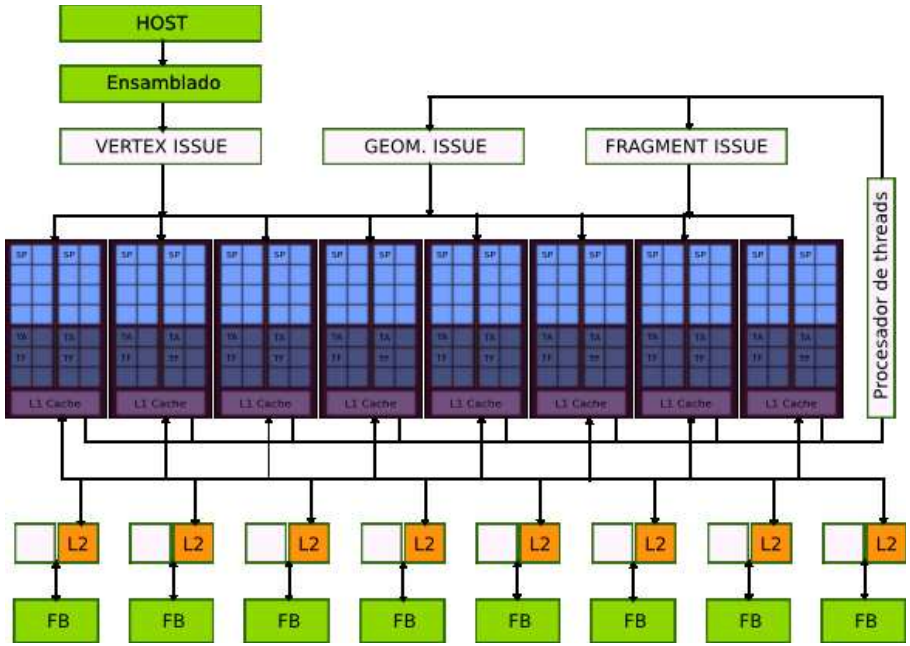
1. Motivation
2. Cholesky factorization (Overview of FLAME)
3. Parallelization
4. Targeting platforms with multiple accelerators
5. Experimental results
6. Concluding remarks

The potential of a GPU



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

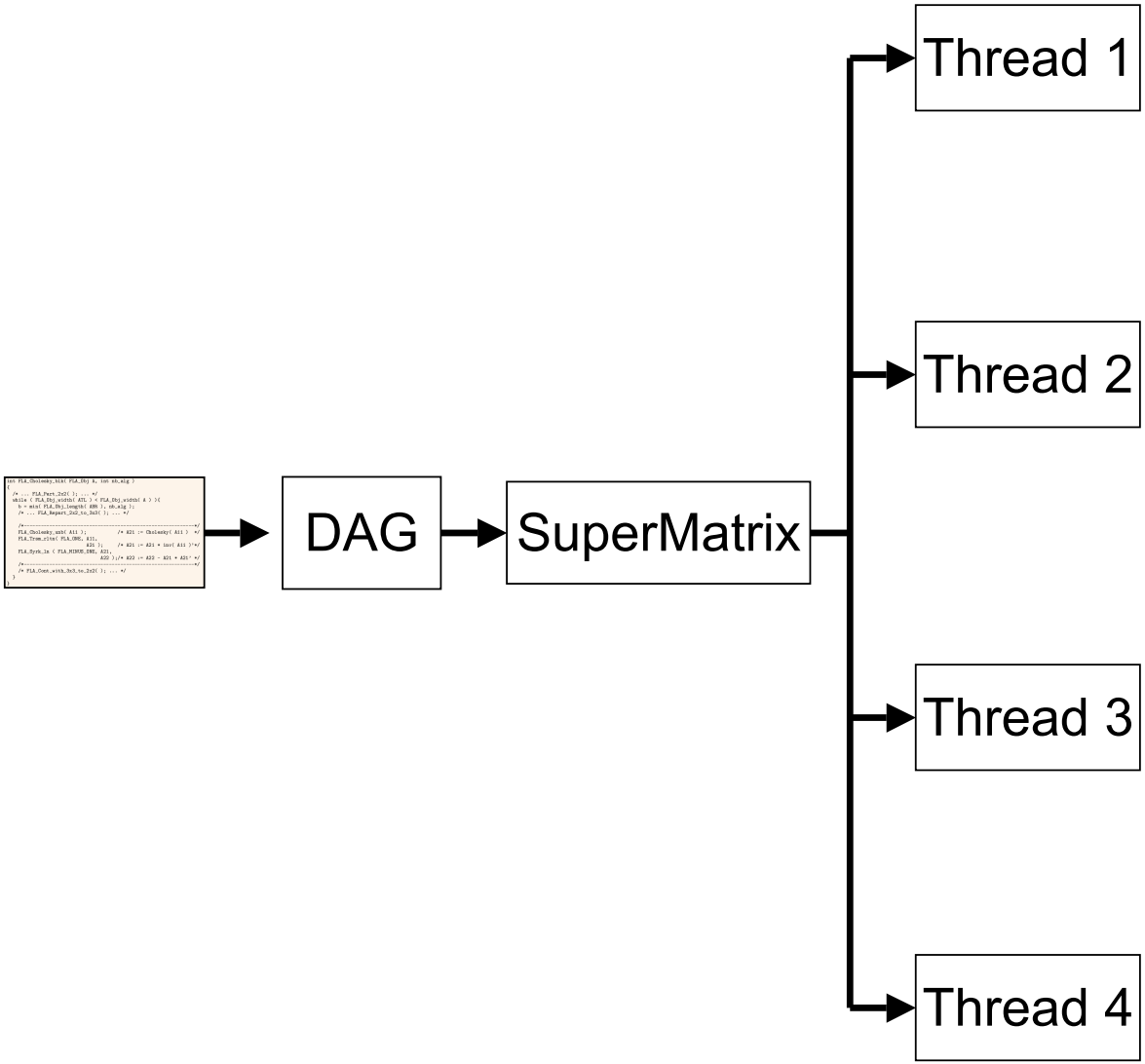
CUBLAS Example

```
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);
  ...
}
```

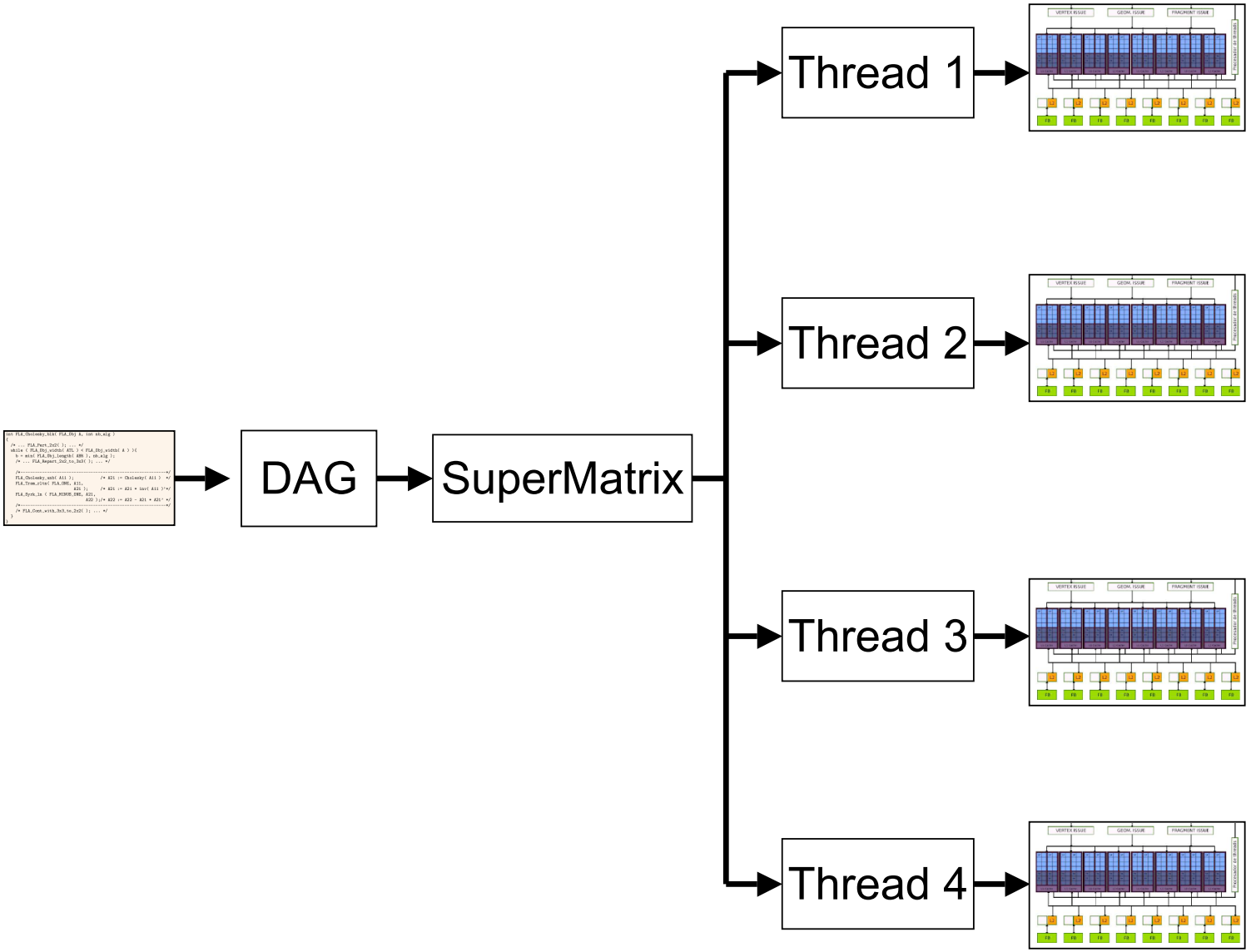
A typical CUDA (and CUBLAS) program has 3 phases:

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

How to use multiple GPUs?



How to use multiple GPUs?



Porting SuperMatrix to multiple GPUs

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

Programming effort

Porting SuperMatrix to multiple GPUs

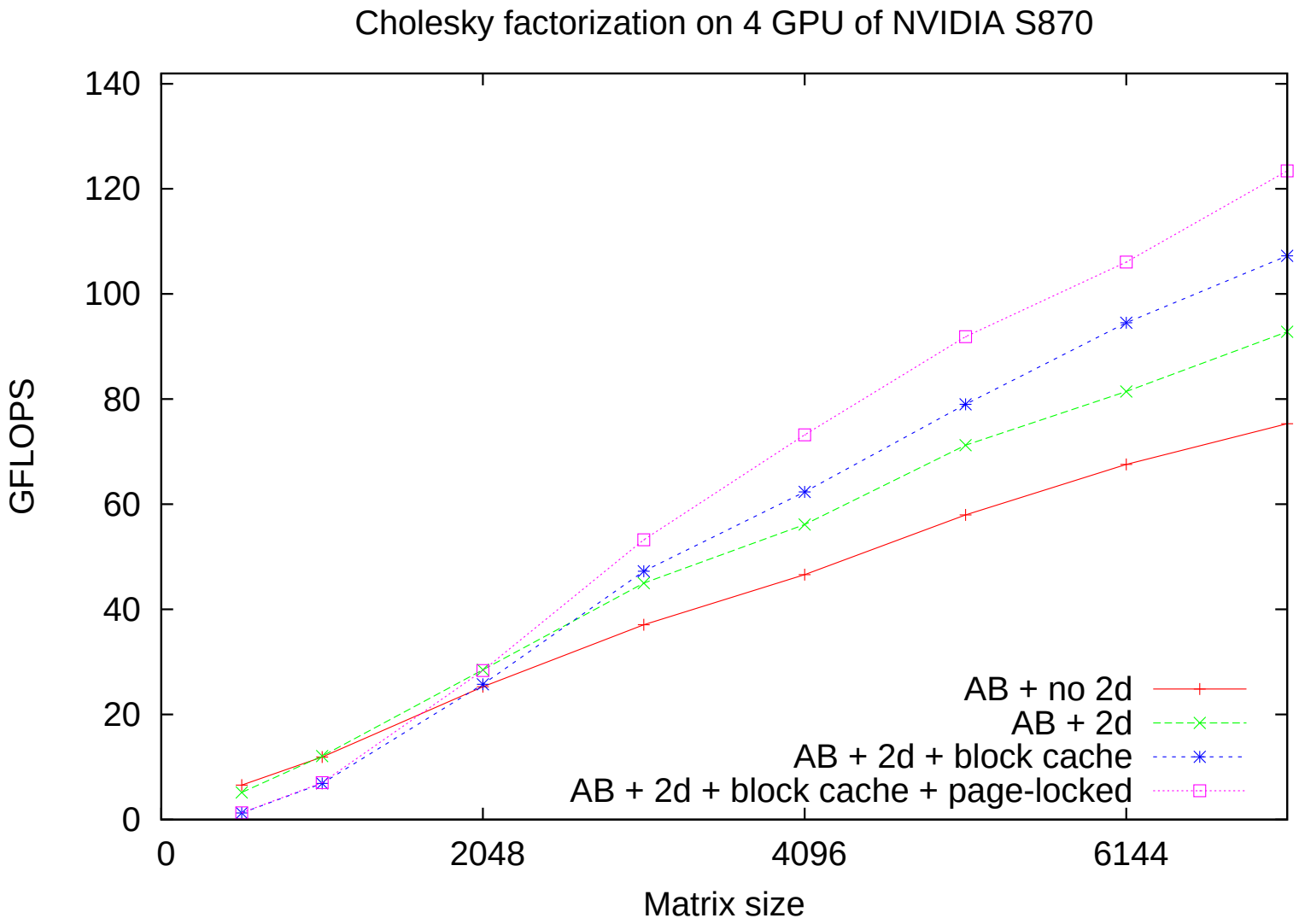
- Employ the equivalence: 1 core $\equiv$ 1 GPU
- Difference: Explicitly transfer data from RAM to video memory
- Run-time system (scheduling), storage, and code are independent
- No significative modification to the FLAME codes:
  - Interfacing to CUBLAS
- Run-time system can implement different scheduling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

Programming effort

Two hours!

Outline

Porting SuperMatrix. Experimental results



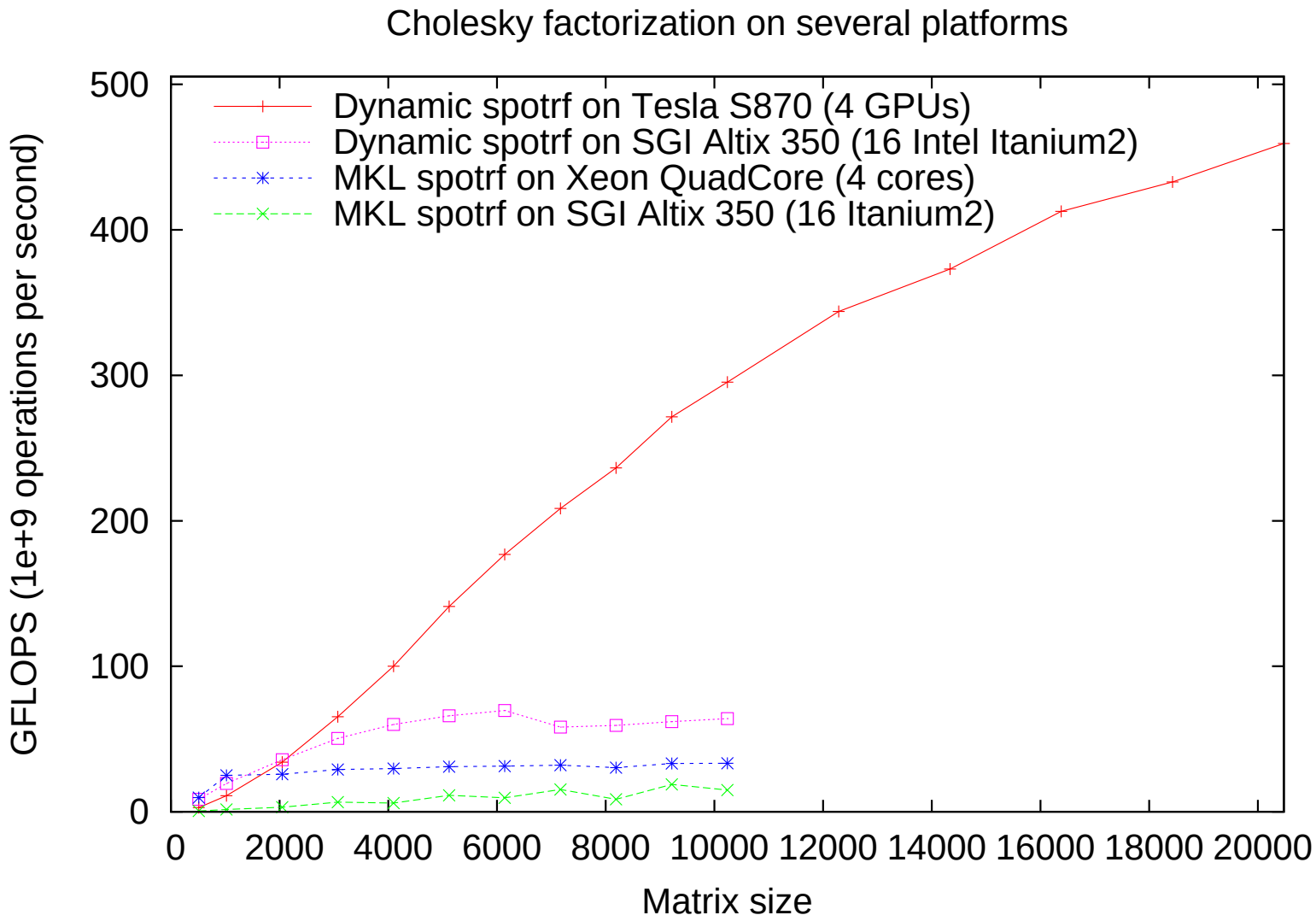
Porting SuperMatrix. Experimental results

A more elaborate port required for high-performance:

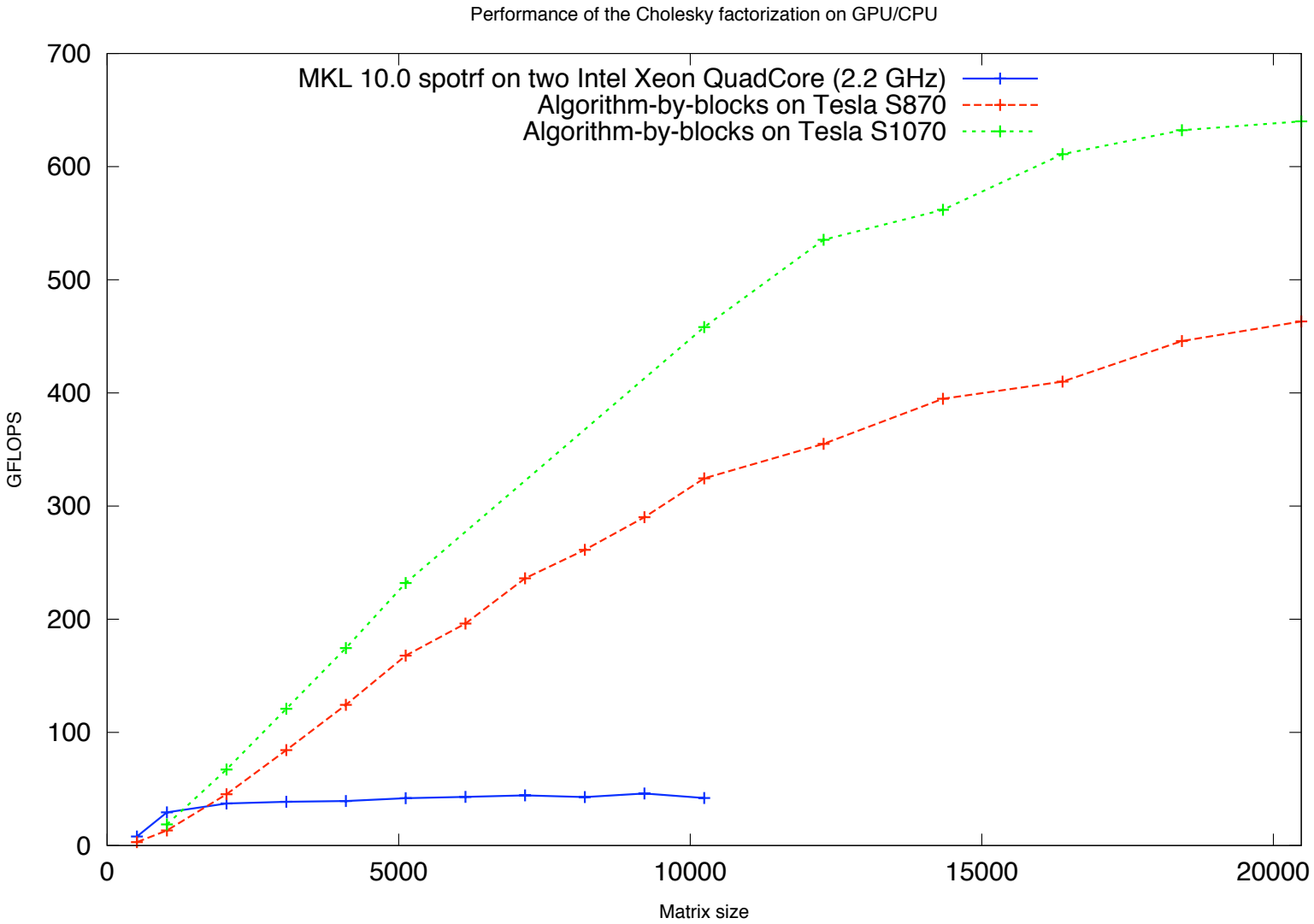
- 2-D work distribution
- Memory/cache coherence techniques to reduce transferences between RAM and video memory: write-back and write-invalidate

For details, see PPoPP09 paper.

Porting SuperMatrix. Experimental results



More recent results



Other Operations

Level-3 BLAS (matrix-matrix operations)

- Follow very similarly

LU/QR factorization

- New algorithms-by-blocks have been developed
- Require kernels for an individual GPU to be written

Outline

1. Motivation
2. Cholesky factorization (Overview of FLAME)
3. Parallelization
4. Targeting platforms with multiple accelerators
5. Experimental results
6. Concluding remarks

Related Approaches

Cilk (MIT) and SMPSs (Barcelona SuperComputing Center)

- General-purpose parallel programming
  - Cilk → irregular/recursive problems
  - SMPSs → more general, also manages dependencies
- High-level language based on OpenMP-like pragmas + compiler + runtime system
- Modest results for dense linear algebra

PLASMA Project

- Next step in the LAPACK evolutionary path
- Traditional style of implementing algorithms
- Does not solve the programmability problem

Outline

1. Motivation
2. Cholesky factorization (Overview of FLAME)
3. Parallelization
4. Targeting platforms with multiple accelerators
5. Experimental results
6. Concluding remarks

Conclusion

A success story

- For the domain of dense linear algebra libraries, FLAME+SuperMatrix appears to solve the programmability problem
- Very good performance can be achieved with multiple accelerators in this domain

For more information...

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