FLAG@lab

An M-script API for Linear Algebra Operations on GPUs

Sergio Barrachina Maribel Castillo Francisco Igual
Rafael Mayo Enrique S. Quintana-Ortí

High Performance Computing & Architectures Group
Universitat Jaume I de Castellón (Spain)

Motivation (I)

Current graphics processors (GPUs) are gaining wide appeal as HPC accelerators

Performance

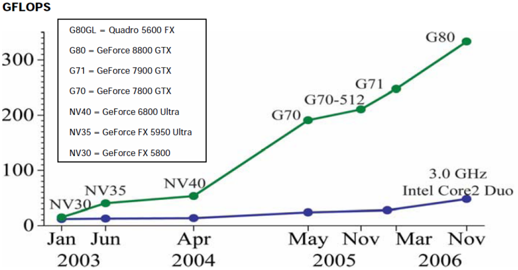
Programmability

- Hardware:
  - More functionality
- Software:
  - Cg/OpenGL vs. CUDA
  - CUBLAS/CUFFT
  - ...

Motivation (I)

Current graphics processors (GPUs) are gaining wide appeal as HPC accelerators

Performance



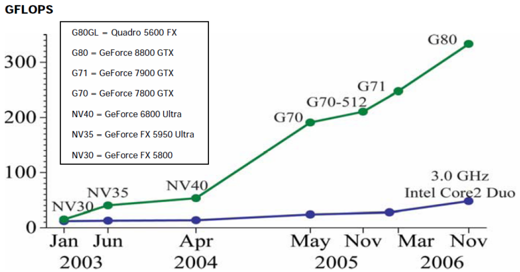
Programmability

- Hardware:
  - More functionality
- Software:
  - Cg/OpenGL vs. CUDA
  - CUBLAS/CUFFT
  - ...

Motivation (I)

Current graphics processors (GPUs) are gaining wide appeal as HPC accelerators

Performance



Programmability

- Hardware:
  - More functionality
- Software:
  - Cg/OpenGL vs. CUDA
  - CUBLAS/CUFFT
  - ...

Motivation (II)

However, it is still difficult to use them from widely used environments (MATLAB/OCTAVE)

Our proposal: FLAG@lab

An advanced API for Linear Algebra Operations on GPUs

- **Easy to use**: MATLAB based
- **Fast**: GPU based
- **Reliable**: FLAME based

Outline

The FLAG@lab API

FLAG@lab

- FLAG@lab: M-script API combining ease of use and performance
- Can be combined with any type of accelerator: Cell B.E., ClearSpeed boards or GPUs

FLAG@lab provides two different interfaces:

1. FLAG@lab: an Advanced API for Linear Algebra Operations
2. FLAGS@lab: a Simplified API (no user-managed transfers)

The FLAG@lab API

FLAG@lab

- FLAG@lab: M-script API combining ease of use and performance
- Can be combined with any type of accelerator: Cell B.E., ClearSpeed boards or GPUs

FLAG@lab provides two different interfaces:

1. FLAG@lab: an Advanced API for Linear Algebra Operations
2. FLAGS@lab: a Simplified API (no user-managed transfers)

FLAG@lab objects

Objects in FLAG@lab

- Object: key concept in FLAG@lab
- Representation of a matrix in the accelerator (GPU) memory
- The API allows the user to:
  1. Create, destroy and inquiry properties of objects
  2. Set the contents of an object
  3. Define *views*
  4. Operate with objects through BLAS

Typical FLAG@lab script structure (I)

Initialization

`FLAG_Init()` / `FLAG_Finalize()`

Do the necessary environment setup

↓

Object Creation

`FLAG_Obj_create( 'TYPE', m, n )`

Allocate space for the object in the GPU memory

↓

Typical FLAG@lab script structure (I)

Initialization

`FLAG_Init()` / `FLAG_Finalize()`

Do the necessary environment setup

↓

Object Creation

`FLAG_Obj_create( 'TYPE', m, n )`

Allocate space for the object in the GPU memory

↓

Typical FLAG@lab script structure (II)

↓

CPU -> GPU Transfers

`FLAG_Axpy_matrix_to_object( alpha, B, A, i, j )`

Fill the entries of the object A in GPU memory space with the result of the product `alpha` · `B`

`FLAG_Obj_set( A, value )`

Initialize all entries of the object A to equal `value`

↓

Matrix Computation

Complete BLAS interface implementation

`FLAG_Gemm( transA, transB, alpha, A, B, beta, C )`

Typical FLAG@lab script structure (II)

↓

CPU -> GPU Transfers

`FLAG_Axpy_matrix_to_object( alpha, B, A, i, j )`

Fill the entries of the object A in GPU memory space with the result of the product $\texttt{alpha} \cdot \texttt{B}$

`FLAG_Obj_set( A, value )`

Initialize all entries of the object A to equal `value`

↓

Matrix Computation

Complete BLAS interface implementation

`FLAG_Gemm( transA, transB, alpha, A, B, beta, C )`

Typical FLAG@lab script structure (III)

↓

Utils

`FLAG_Obj_show( A )`

Prints the entries of the object A

`Aview = FLAG_Obj_view( A, m, n, i, j )`

Create the view `Aview` consisting of the $m \times n$ submatrix (block) of A starting at coordinate (i,j)

↓

GPU -> CPU Transfers

`B = FLAG_object_to_matrix( m, n, A, i, j )`

Retrieve the entries of the object A in the GPU memory space

Typical FLAG@lab script structure (III)

↓

Utils

`FLAG_Obj_show( A )`

Prints the entries of the object A

`Aview = FLAG_Obj_view( A, m, n, i, j )`

Create the view `Aview` consisting of the $m \times n$ submatrix (block) of A starting at coordinate (i,j)

↓

GPU -> CPU Transfers

`B = FLAG_object_to_matrix( m, n, A, i, j )`

Retrieve the entries of the object A in the GPU memory space

Example: $C = A^T \cdot B$

```
FLAG_Init ( ) ;

Aobj = FLAG_Obj_create ( 'FLAG_FLOAT' , n , n ) ;
Bobj = FLAG_Obj_create ( 'FLAG_FLOAT' , n , n ) ;
Cobj = FLAG_Obj_create ( 'FLAG_FLOAT' , n , n ) ;

FLAG_Axpy_matrix_to_object ( 1.0 , A , Aobj , 1 , 1 ) ;
FLAG_Axpy_matrix_to_object ( 1.0 , B , Bobj , 1 , 1 ) ;
FLAG_Obj_set ( Cobj , 0.0 ) ;

FLAG_Gemm ( 'FLAG_TRANSPOSE' , ...
            'FLAG_NO_TRANSPOSE' , ...
            1.0 , Aobj , Bobj , 0.0 , Cobj ) ;

FLAG_Obj_show ( Cobj ) ;

FLAG_Obj_free ( Aobj ) ;
FLAG_Obj_free ( Bobj ) ;
FLAG_Obj_free ( Cobj ) ;

FLAG_Finalize ( ) ;
```

Initialization

Object creation

CPU -> GPU transfers and initialization

GPU computations

Object deallocation

Example: $C = A^T \cdot B$

```
FLAG_Init();

Aobj = FLAG_Obj_create( 'FLAG_FLOAT', n, n );
Bobj = FLAG_Obj_create( 'FLAG_FLOAT', n, n );
Cobj = FLAG_Obj_create( 'FLAG_FLOAT', n, n );

FLAG_Axpy_matrix_to_object( 1.0, A, Aobj, 1, 1 );
FLAG_Axpy_matrix_to_object( 1.0, B, Bobj, 1, 1 );
FLAG_Obj_set( Cobj, 0.0 );

FLAG_Gemm( 'FLAG_TRANSPOSE', ...
           'FLAG_NO_TRANSPOSE', ...
           1.0, Aobj, Bobj, 0.0, Cobj );

FLAG_Obj_show( Cobj );

FLAG_Obj_free( Aobj );
FLAG_Obj_free( Bobj );
FLAG_Obj_free( Cobj );

FLAG_Finalize();
```

Initialization

Object creation

CPU -> GPU transfers and initialization

GPU computations

Object deallocation

Example: $C = A^T \cdot B$

```
FLAG_Init();

Aobj = FLAG_Obj_create( 'FLAG_FLOAT', n, n );
Bobj = FLAG_Obj_create( 'FLAG_FLOAT', n, n );
Cobj = FLAG_Obj_create( 'FLAG_FLOAT', n, n );

FLAG_Axpy_matrix_to_object( 1.0, A, Aobj, 1, 1 );
FLAG_Axpy_matrix_to_object( 1.0, B, Bobj, 1, 1 );
FLAG_Obj_set( Cobj, 0.0 );

FLAG_Gemm( 'FLAG_TRANSPOSE', ...
           'FLAG_NO_TRANSPOSE', ...
           1.0, Aobj, Bobj, 0.0, Cobj );

FLAG_Obj_show( Cobj );

FLAG_Obj_free( Aobj );
FLAG_Obj_free( Bobj );
FLAG_Obj_free( Cobj );

FLAG_Finalize();
```

Initialization

Object creation

CPU -> GPU transfers and initialization

GPU computations

Object deallocation

Example: $C = A^T \cdot B$

```
FLAG_Init ( ) ;

Aobj = FLAG_Obj_create ( 'FLAG_FLOAT' , n , n ) ;
Bobj = FLAG_Obj_create ( 'FLAG_FLOAT' , n , n ) ;
Cobj = FLAG_Obj_create ( 'FLAG_FLOAT' , n , n ) ;

FLAG_Axpy_matrix_to_object ( 1.0 , A, Aobj , 1 , 1 ) ;
FLAG_Axpy_matrix_to_object ( 1.0 , B, Bobj , 1 , 1 ) ;
FLAG_Obj_set ( Cobj , 0.0 ) ;

FLAG_Gemm ( 'FLAG_TRANSPOSE' , . . .
            'FLAG_NO_TRANSPOSE' , . . .
            1.0 , Aobj , Bobj , 0.0 , Cobj ) ;

FLAG_Obj_show ( Cobj ) ;

FLAG_Obj_free ( Aobj ) ;
FLAG_Obj_free ( Bobj ) ;
FLAG_Obj_free ( Cobj ) ;

FLAG_Finalize ( ) ;
```

- Initialization
- Object creation
- CPU -> GPU transfers and initialization
- GPU computations
- Object deallocation

Example: $C = A^T \cdot B$

```
FLAG_Init();

Aobj = FLAG_Obj_create( 'FLAG_FLOAT', n, n );
Bobj = FLAG_Obj_create( 'FLAG_FLOAT', n, n );
Cobj = FLAG_Obj_create( 'FLAG_FLOAT', n, n );

FLAG_Axpy_matrix_to_object( 1.0, A, Aobj, 1, 1 );
FLAG_Axpy_matrix_to_object( 1.0, B, Bobj, 1, 1 );
FLAG_Obj_set( Cobj, 0.0 );

FLAG_Gemm( 'FLAG_TRANSPOSE',...
           'FLAG_NO_TRANSPOSE',...
           1.0, Aobj, Bobj, 0.0, Cobj );

FLAG_Obj_show( Cobj );

FLAG_Obj_free( Aobj );
FLAG_Obj_free( Bobj );
FLAG_Obj_free( Cobj );

FLAG_Finalize();
```

Initialization

Object creation

CPU -> GPU transfers and initialization

GPU computations

Object deallocation

FLAGS@lab: a Simple Interface

Ease of use Vs. Performance

- FLAG@lab is efficient from the transfers point of view . . .
- . . . but we are looking for ease of use: **FLAGS@lab**

FLAGS@lab

- FLAGS@lab: simplified interface based on FLAG@lab
- Only provides access to the BLAS kernels
- Input matrices transferred $\Rightarrow$ Computation is performed $\Rightarrow$ Results are transferred back

FLAGS@lab: a Simple Interface

Ease of use Vs. Performance

- FLAG@lab is efficient from the transfers point of view . . .
- . . . but we are looking for ease of use: **FLAGS@lab**

FLAGS@lab

- FLAGS@lab: simplified interface based on FLAG@lab
- Only provides access to the BLAS kernels
- Input matrices transferred ⇒ Computation is performed ⇒ Results are transferred back

FLAGS@lab example

FLAG@lab

```
%%% Object creation
A=FLAG_Obj_create('FLA_FLOAT',n,n)
B=FLAG_Obj_create('FLA_FLOAT',n,n)
C=FLAG_Obj_create('FLA_FLOAT',n,n)
...
FLAG_Gemm( transA, transB,
      alpha, A, B,
      beta,  C )
```

FLAGS@lab

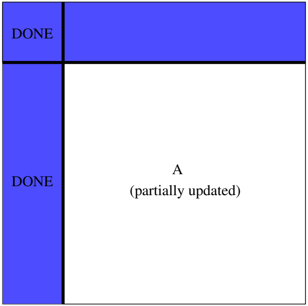
```
%%% Object creation
A=rand(n)
B=rand(n)
C=zeros( n )
...
FLAGS_Gemm( transA, transB,
        alpha, A, B,
        beta,  C )
```

Note that A, B and C are not the same type of objects in both cases

FLAME

- FLAME: high level abstraction and notation for dense linear algebra algorithms
- Not only a library:
  - Notation for expressing algorithms
  - Methodology for systematic derivation of algorithms
  - Application Program Interfaces (APIs) for representing the algorithms in code
  - Tools and more
- M-script API: FLAME@lab
- Example: Cholesky factorization

Cholesky factorization with FLAME



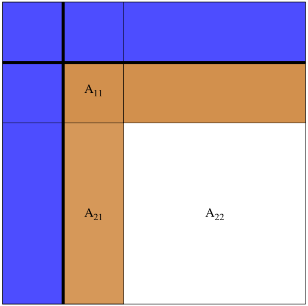
Definition

Given $A \rightarrow n \times n$ s.p.d.

$$A = LL^T$$

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

Cholesky factorization with FLAME

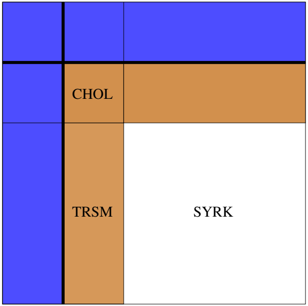


Repartition

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

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

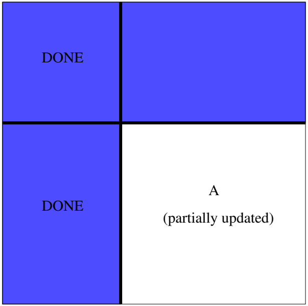
Cholesky factorization with FLAME



Operations per iteration:

```
%==================%
A11 = chol( A11 );
A21 = A21 / tril( A11 )';
A22 = A22 - tril( A21 * A21' );
%==================%
```

Cholesky factorization with FLAME

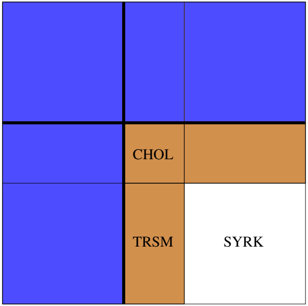


Continue with

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

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

Cholesky factorization with FLAME



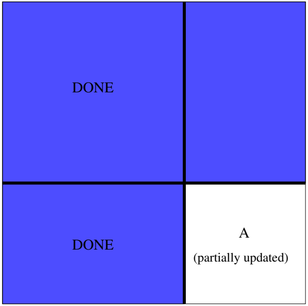
Repartition

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

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

```
[ A00, A01, A02, ...
  A10, A11, A12, ...
  A20, A21, A22 ] = ...
        FLA_Repart_2x2_to_3x3( ATL, ATR, ...
                               ABL, ABR, ...
                               b, b, 'FLA_BR' );
```

Cholesky factorization with FLAME



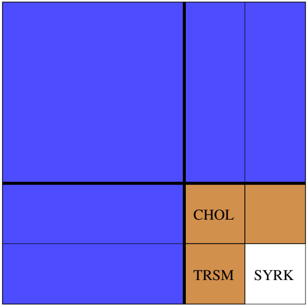
Continue with

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

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

```
[ ATL, ATR, ...
  ABL, ABR ] = ...
  FLA_Cont_with_3x3_to_2x2( A00, A01, A02, ...
                            A10, A11, A12, ...
                            A20, A21, A22, ...
                           'FLA_TL' );
```

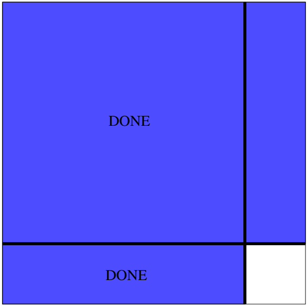
Cholesky factorization with FLAME



The algorithm iterates...

```
while ( size( ATL, 1 ) < size( A, 1 ) )
...
```

Cholesky factorization with FLAME



The algorithm iterates...

```
while ( size( ATL, 1 ) < size( A, 1 ) )
...
```

FLAME notation and FLAME@lab

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 $m(A_{TL}) < m(A)$ **do**

Determine block size n_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 $n_b \times n_b$

$$A_{11} := \text{CHOL\_UNB}(A_{11})$$
$$A_{21} := A_{21}\,\text{TRIL}\,(A_{11})^{-T}$$
$$A_{22} := A_{22} - A_{21}A_{21}^{T}$$

Continue with

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

endwhile

```
function [ A_out ] = FLA_Cholesky_blk( A, nb_alg )

 [ ATL, ATR, ...
   ABL, ABR ] = FLA_Part_2x2( A, 0, 0, 'FLA_TL' );

 while ( size( ATL, 1 ) < size( A, 1 ) )

   b = min( size( ABR, 1 ), nb_alg );

   [ A00, A01, A02, ...
     A10, A11, A12, ...
     A20, A21, A22 ] = ...
              FLA_Repart_2x2_to_3x3( ATL, ATR, ...
                                     ABL, ABR, ...
                                     b, b, 'FLA_BR' );

   %------------------------------------------------------------%
   A11 = FLA_Cholesky_unb( A11 );
   A21 = A21 * inv( tril( A11 ) )';
   A22 = A22 - tril( A21 * A21' );
   %------------------------------------------------------------%

   [ ATL, ATR, ...
     ABL, ABR ] = ...
          FLA_Cont_with_3x3_to_2x2( A00, A01, A02, ...
                                    A10, A11, A12, ...
                                    A20, A21, A22, ...
                                   'FLA_TL' );

 end
 A_out = [ ATL, ATR
           ABL, ABR ];
return
```

FLAME@lab - FLAG@lab comparison

```
function [ A ] = FLAG_Cholesky_blk( A, nb_alg )

  [ ATL, ATR, ...
    ABL, ABR ] = FLAG_Part_2x2( A, 0, 0, 'FLAG_TL' );

  while ( FLAG_Obj_length( ATL, 1 ) < FLAG_Obj_length( A, 1 ) )

    b = min( FLAG_Obj_length( ABR, 1 ), nb_alg );

    [ A00, A01, A02, ...
      A10, A11, A12, ...
      A20, A21, A22 ] = FLAG_Repart_2x2_to_3x3( ATL, ATR, ...
                                                 ABL, ABR, ...
                                                 b, b, 'FLAG_BR' );

    %------------------------------------------------------------%
    FLAG_Cholesky_unb( A11 );

    FLAG_Trsm( 'FLAG_RIGHT', 'FLAG_LOWER_TRIANGULAR',
               'FLAG_TRANSPOSE', 'FLAG_NONUNIT_DIAG',
               1.0, A11, A21 );

    FLAG_Syrk( 'FLAG_LOWER_TRIANGULAR', 'FLAG_NO_TRANSPOSE',
               -1.0, A21, 1.0, A22 );
    %------------------------------------------------------------%

    [ ATL, ATR, ...
      ABL, ABR ] = FLAG_Cont_with_3x3_to_2x2( A00, A01, A02, ...
                                               A10, A11, A12, ...
                                               A20, A21, A22, ...
                                               'FLAG_TL' );

  end
return
```

Main differences:

1. Indices -> views
2. BLAS routines of FLAG@lab
3. Matrices must have been allocated and transferred to GPU

Implementation on top of CUBLAS

Why CUBLAS

- CUBLAS: full implementation of BLAS for Nvidia GPUs (G80 and newer)
- Auxiliary functions to transfer, allocate, deallocate (actually CUDA API calls)
- CUBLAS 1.1 is not fully optimized ([Barrachina et al. 2008])

Flexibility

However, FLAG@lab can be easily adapted to any other platform/BLAS implementation

Implementation on top of CUBLAS

Why CUBLAS

- CUBLAS: full implementation of BLAS for Nvidia GPUs (G80 and newer)
- Auxiliary functions to transfer, allocate, deallocate (actually CUDA API calls)
- CUBLAS 1.1 is not fully optimized ([Barrachina et al. 2008])

Flexibility

However, FLAG@lab can be easily adapted to any other platform/BLAS implementation

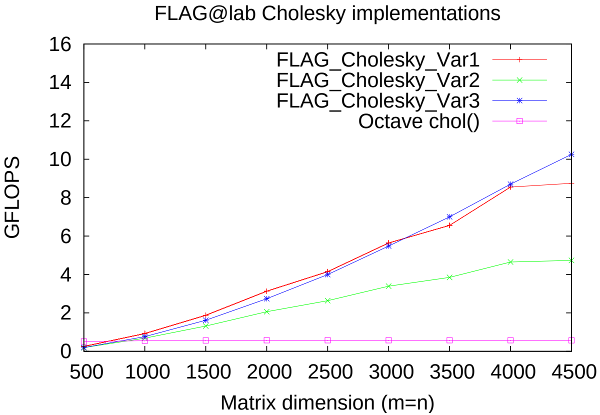
Experimental setup

- Hardware setup:

	CPU	GPU
Processor	Intel Core 2 Duo	NVIDIA 8800 Ultra
Codename	Crusoe E6320	G80
Clock frequency	1.86 GHz	575 MHz
Memory speed	2×333 MHz	2×900 MHz
Bus width	64 bits	384 bits
Max. bandwidth	5.3 GB/s	86.4 GB/s
Memory	1024 MB DDR2	768 MB GDDR3
Bus	PCI Express x16 (4 GB/s)	

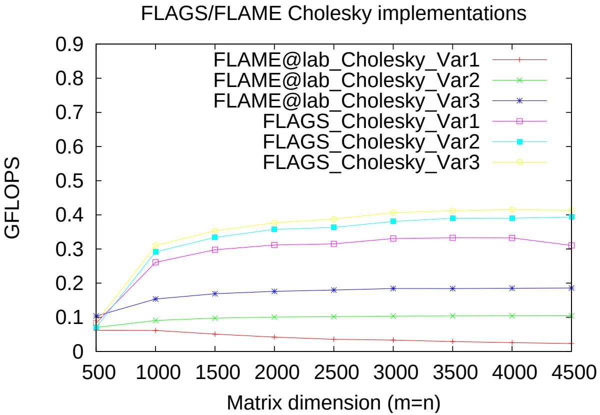
- Software setup:
  - CUBLAS 1.1
  - Octave 2.9.19

Results: FLAG@lab



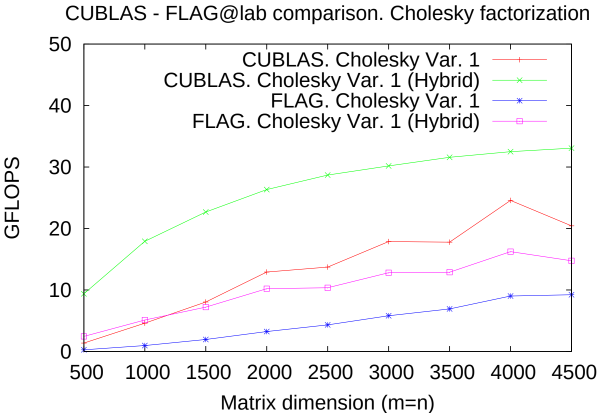
- Compares three different Cholesky variants
- Achieved speedup of $\approx 20x$ for large matrices

Results: FLAGS@lab



- Compares FLAGS@lab/FLAME@lab
- Even simplified interfaces achieve better results

FLAG@lab vs. pure CUBLAS



- Compares “pure” CUBLAS implementations with FLAG@lab
- We lose performance. . .

FLAG@lab: inefficiency sources

There are three main inefficiency sources in FLAG@lab:

1. Single precision ⇔ Double precision conversions (Octave)
2. CUBLAS (1.1) based ⇒ Not fully optimized
3. Matlab based ⇒ Inefficient in some ways

However, most of them can be solved:

1. Waiting for the next generation of GPUs with DP support
2. Using tuned CUBLAS implementations (padding, Volkov Gemm, Gemm based routines, . . .)
3. Preparing a FLAME/C API port to GPUs

FLAG@lab: inefficiency sources

There are three main inefficiency sources in FLAG@lab:

1. Single precision $\Leftrightarrow$ Double precision conversions (Octave)
2. CUBLAS (1.1) based $\Rightarrow$ Not fully optimized
3. Matlab based $\Rightarrow$ Inefficient in some ways

However, most of them can be solved:

1. Waiting for the next generation of GPUs with DP support
2. Using tuned CUBLAS implementations (padding, Volkov Gemm, Gemm based routines, . . .)
3. Preparing a FLAME/C API port to GPUs

Conclusions

- Programmability of GPU has been improved, but it is not enough
- We propose FLAG@lab as an easy way to develop high performance codes
- Results achieved for a GPU implementation show the benefits of the API
- However, there are still some performance penalties that will be solved
- Further work: C implementation + Other platforms adaptation

Thank you...

For more information. . .

http://www3.uji.es/˜figual
figual@icc.uji.es