Out-of-Core Solution of Linear Systems on Graphics Processors

Maribel Castillo Francisco Igual Rafael Mayo Enrique S. Quintana-Ortí Gregorio Quintana-Ortí Robert van de Geijn

Departamento de Ingeniería y Ciencia de los Computadores.
University Jaume I.
Castellón (Spain)

Motivation (I)

- The Out-of-Core (OOC) solution arises in a wide variety of fields
- Boundary Element Methods (BEM), radial function methods,. . .
- Data structures too large to fit in memory $\Rightarrow$ rely on disk storage
- Parallel OOC libraries: ScaLAPACK, SOLAR
- POOCLAPACK $\Rightarrow$ object-oriented API for OOC

Motivation (II)

- GPUs are gaining wide appeal as general-purpose accelerators
- The GPU architecture is suitable for large-scale problems
- Excellent ratio cost/performance
- Newest GPU models introduce interesting programmability features
- Still difficult to use for the scientific community

OOC approach for linear algebra from friendly environments

Motivation (II)

- GPUs are gaining wide appeal as general-purpose accelerators
- The GPU architecture is suitable for large-scale problems
- Excellent ratio cost/performance
- Newest GPU models introduce interesting programmability features
- Still difficult to use for the scientific community

OOC approach for linear algebra from friendly environments

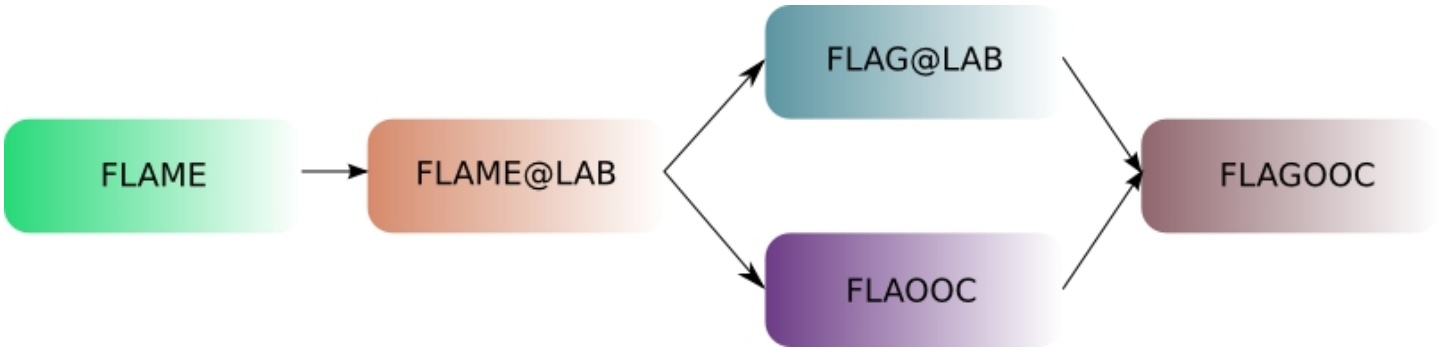
Outline

Introduction

Our approach introduces three main differences:

1. Based on the FLAME project $\Rightarrow$ **OO API** for dense linear algebra
2. Used from **user-friendly environments** (Matlab, Octave or LabView)
3. Targets a different architecture: **the GPU**

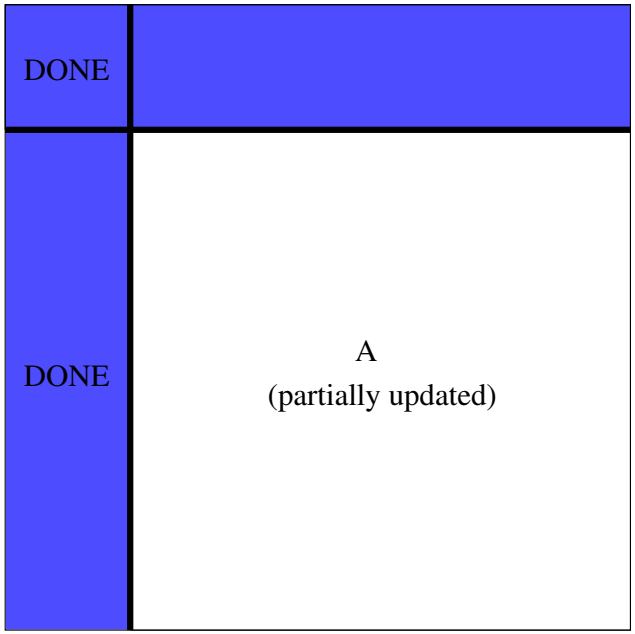
General overview:



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



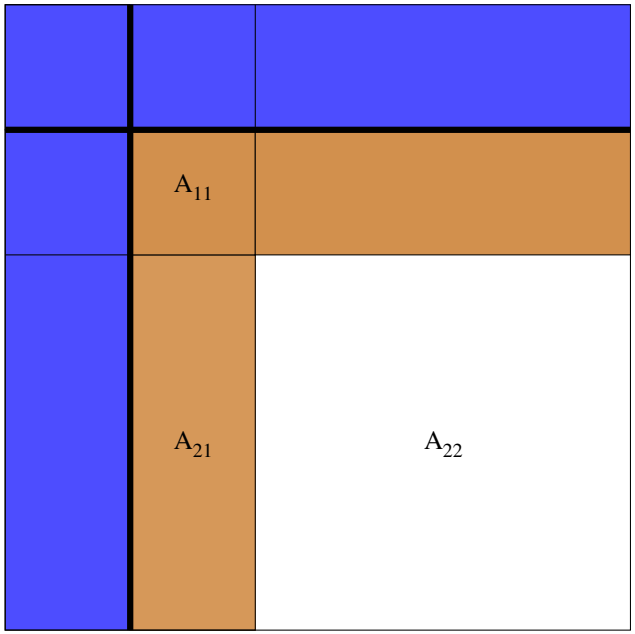
Definition

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

$$A = LL^T$$

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

Cholesky factorization with FLAME

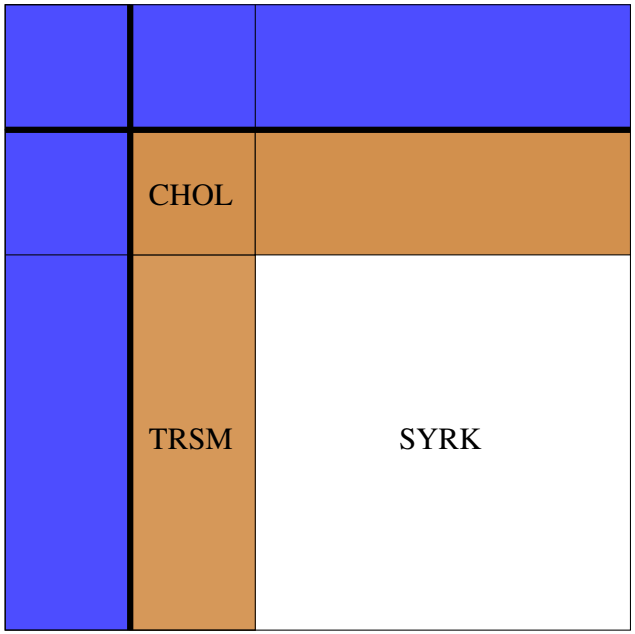


Repartition

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

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

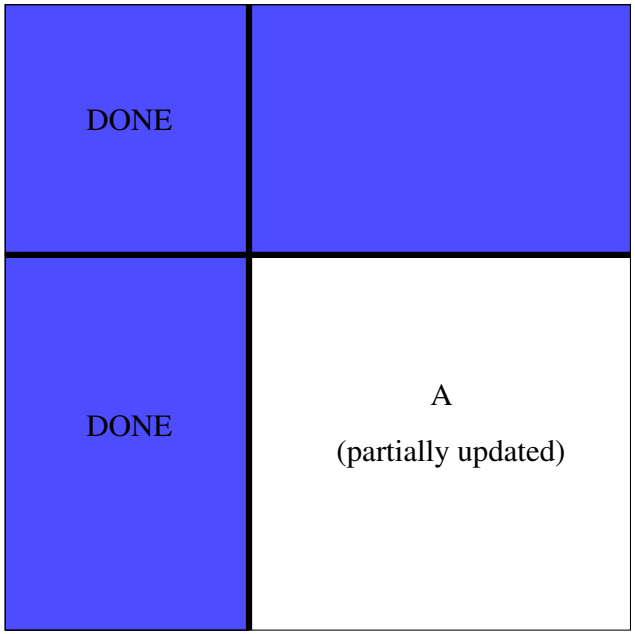
Cholesky factorization with FLAME



Operations per iteration:

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

Cholesky factorization with FLAME

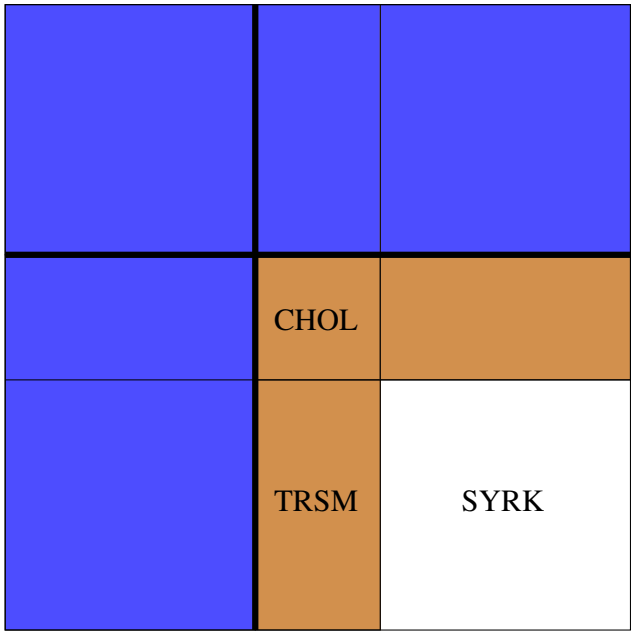


Continue with

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

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

Cholesky factorization with FLAME



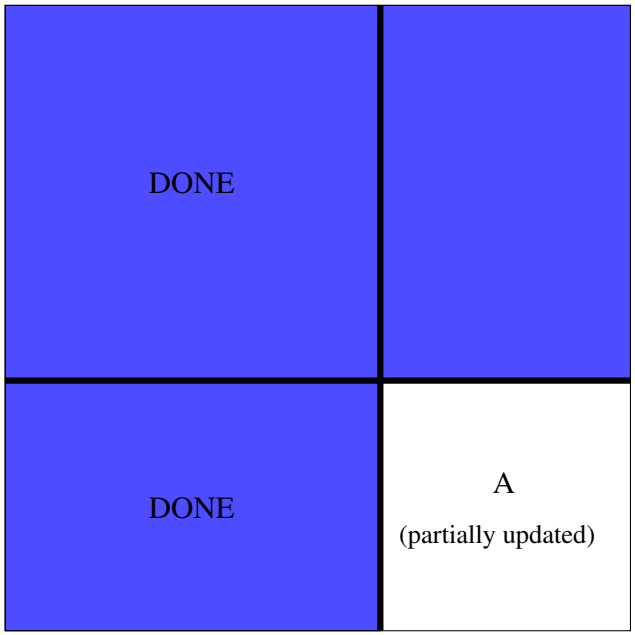
Repartition

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

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



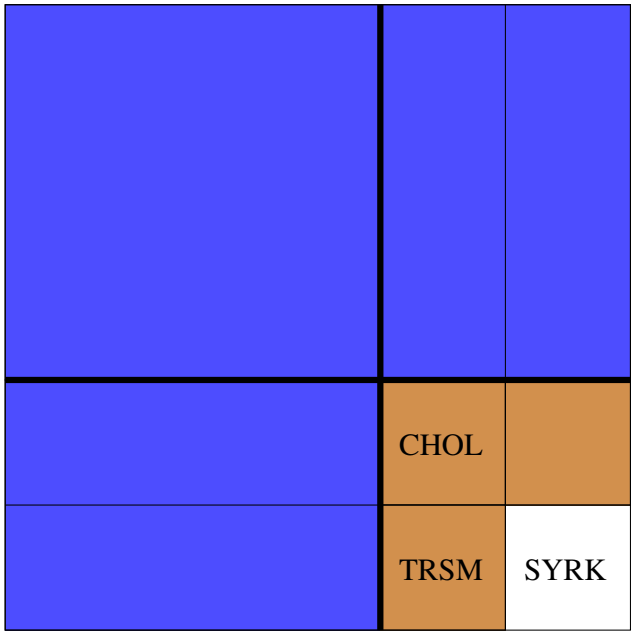
Continue with

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

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' );
```

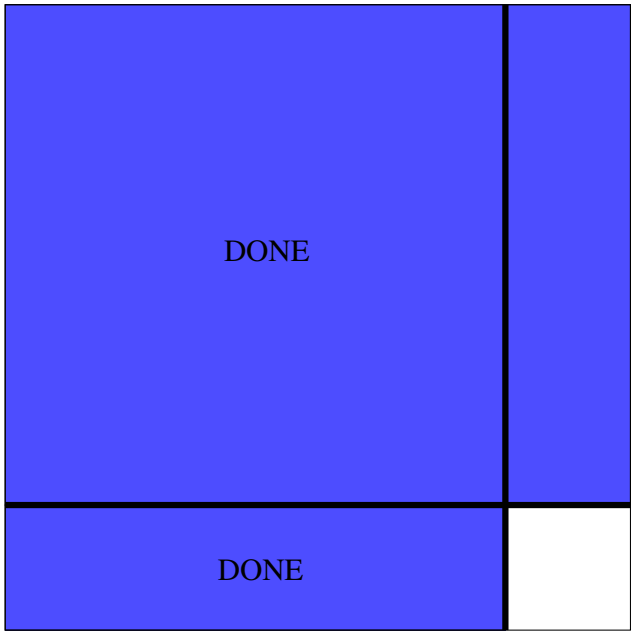
Cholesky factorization with FLAME



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 $n_b \times n_b$

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

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' );

    %-------------------------------------------------------------%

    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
```

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` $\cdot$ `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 `alpha` $\cdot$ `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

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

Out-of-core Cholesky factorization

$$A = \begin{pmatrix} \bar{A}_{0,0} & \bar{A}_{0,1} & \cdots & \bar{A}_{0,N-1} \\ \bar{A}_{1,0} & \bar{A}_{1,1} & \cdots & \bar{A}_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ \bar{A}_{N-1,0} & \bar{A}_{N-1,1} & \cdots & \bar{A}_{N-1,N-1} \end{pmatrix},$$

- Typical OOC Cholesky: matrix partitioned into $t \times t$ tiles **stored in disk**
- Bringing in core only the tiles involved in the current suboperation
- Goal: avoid as much as possible the use of virtual memory

OO API for OOC linear algebra

Collection of routines structured in three groups:

OOC data structure handling: `FLAOOC_`

- Routines in the FLAOOC@lab API
- Create, destroy matrices-by-tiles on disk
- Set and retrieve their contents
- Track the movement to main memory

I/O transfer: `FLAGOOC_`

- Routines to move data from disk to GPU memory and viceversa

GPU computation: `FLAG_`

- Routines from FLAG@lab to compute with data on the GPU memory

OO API for OOC linear algebra

Collection of routines structured in three groups:

OOC data structure handling: `FLAOOC_`

- Routines in the FLAOOC@lab API
- Create, destroy matrices-by-tiles on disk
- Set and retrieve their contents
- Track the movement to main memory

I/O transfer: `FLAGOOC_`

- Routines to move data from disk to GPU memory and viceversa

GPU computation: `FLAG_`

- Routines from FLAG@lab to compute with data on the GPU memory

OO API for OOC linear algebra

Collection of routines structured in three groups:

OOC data structure handling: `FLAOOC_`

- Routines in the FLAOOC@lab API
- Create, destroy matrices-by-tiles on disk
- Set and retrieve their contents
- Track the movement to main memory

I/O transfer: `FLAGOOC_`

- Routines to move data from disk to GPU memory and viceversa

GPU computation: `FLAG_`

- Routines from FLAG@lab to compute with data on the GPU memory

OOC data structure handling. Typical driver

```
n = ...;                                    % Matrix size
t = n/2;                                    % Tile size
B = diag([1:t]);                            % Building block

%_____________________________________________

A = FLAOOC_Obj_create( 'FLA_REAL',          % Entries are real numbers
                       n, n, t,             % n x n matrix with tile size t
                       'File_for_A' );      % File name on disk

%_____________________________________________

FLAOOC_Axpy_matrix_to_object( 1,   B,       % B in A(1:t,1:t)
                              A, t,   t );

FLAOOC_Axpy_matrix_to_object( 1/t, B,       % B/t in A(t+1:n,1:n)
                              A, t+1, 1 );

FLAOOC_Axpy_matrix_to_object( 1,   B,       % B in A(t+1:n,t+1:n)
                              A, t+1, t+1 );

%_____________________________________________

A = FLAGOOC_Chol( A );                      % Compute the Cholesky factor
...                                         % Somehow use the result
FLAOOC_Obj_destroy( A );                    % Free storage
```

FLAGOOC Cholesky

```
function [ A_out ] = FLAGOOC_Chol( A )
  [ ATL, ATR, ...
    ABL, ABR ] = FLAOOC_Part_2x2( A, ...
                                  0, 0, 'FLA_TL' );

  while ( FLAOOC_Obj_length( ATL ) < FLAOOC_Obj_length( A ) )
    [ A00, A01, A02, ...
      A10, A11, A12, ...
      A20, A21, A22 ] = FLAOOC_Repart_2x2_to_3x3( ATL, ATR, ...
                                                  ABL, ABR, ...
                                                  1, 1, 'FLA_BR' );
    %-------------------------------------------------------------------------%
    A10  = FLAGOOC_Trsm( 'FLA_RIGHT', 'FLA_LOWER_TRIANGULAR',
                         'FLA_TRANSPOSE', 'FLA_NONUNIT_DIAG',
                         1, A00, A10 );                          % A10 := A10 * TRIL(A00)^-1

    AGPU = FLAGOOC_OOC_to_GPU( A11 );                            % Copy A11 from OOC to GPU

    AGPU = FLAGOOC_Syrk( 'FLA_LOWER_TRIANGULAR', 'FLA_NO_TRANSPOSE',
                         -1, A10,
                          1, AGPU );                             % A11 := A11 - A10 * A10^T

    AGPU = FLAG_Chol(AGPU);                                      % A11 := chol( A11 )

    FLAGOOC_GPU_to_OOC( AGPU, A11 );                             % Copy AGPU from GPU to OOC
    %-------------------------------------------------------------------------%
    [ ATL, ATR, ...
      ABL, ABR ] = FLAOOC_Cont_with_3x3_to_2x2( A00, A01, A02, ...
                                                A10, A11, A12, ...
                                                A20, A21, A22, ...
                                                'FLA_TL' );
  end
  A_out = ATL;
return
```

I/O transfer and GPU calculations

Data movement between disk and GPU memory is done by:

- `FLAG_OOC_to_GPU`
- `FLAG_GPU_to_OOC`

GPU computation is done using FLAG@lab in core routines:

- `FLAGOOC_Trsm` and `FLAGOOC_Syrk` use the FLAGOOC@lab API
- Decomposed into calls to FLAG@lab and transfer routines calls

FLAGOOC Syrk

```
function [ C_out ] = FLAGOOC_Syrk( uplo, trans, alpha, A, beta, C )

  C = FLAG_Scal( beta, C );                    % C := C + beta * C

  [ AL, AR ] = FLAOOC_Part_1x2( A, ...
                                'FLA_LEFT' );

  while ( FLAOOC_Obj_width( AL ) < FLAOOC_Obj_width( A ) )

   [ A0, A1, A2 ]= FLAOOC_Repart_1x2_to_1x3( AL, AR, ...
                                             1, 'FLA_RIGHT' );

   %-------------------------------------------------------------------%
   AGPU = FLAGOOC_OOC_to_GPU( A1 );            % Copy A1 from OOC to GPU

   C = FLAG_Syrk( uplo, trans,
                  alpha, AGPU,
                  1, C );                      % C := C + alpha * A1 * A1'
   %-------------------------------------------------------------------%

   [ AL, AR ] = FLAOOC_Cont_with_1x3_to_1x2( A0, A1, A2, ...
                                             'FLA_LEFT' );

 end
 C_out = C;
return
```

Experimental setup

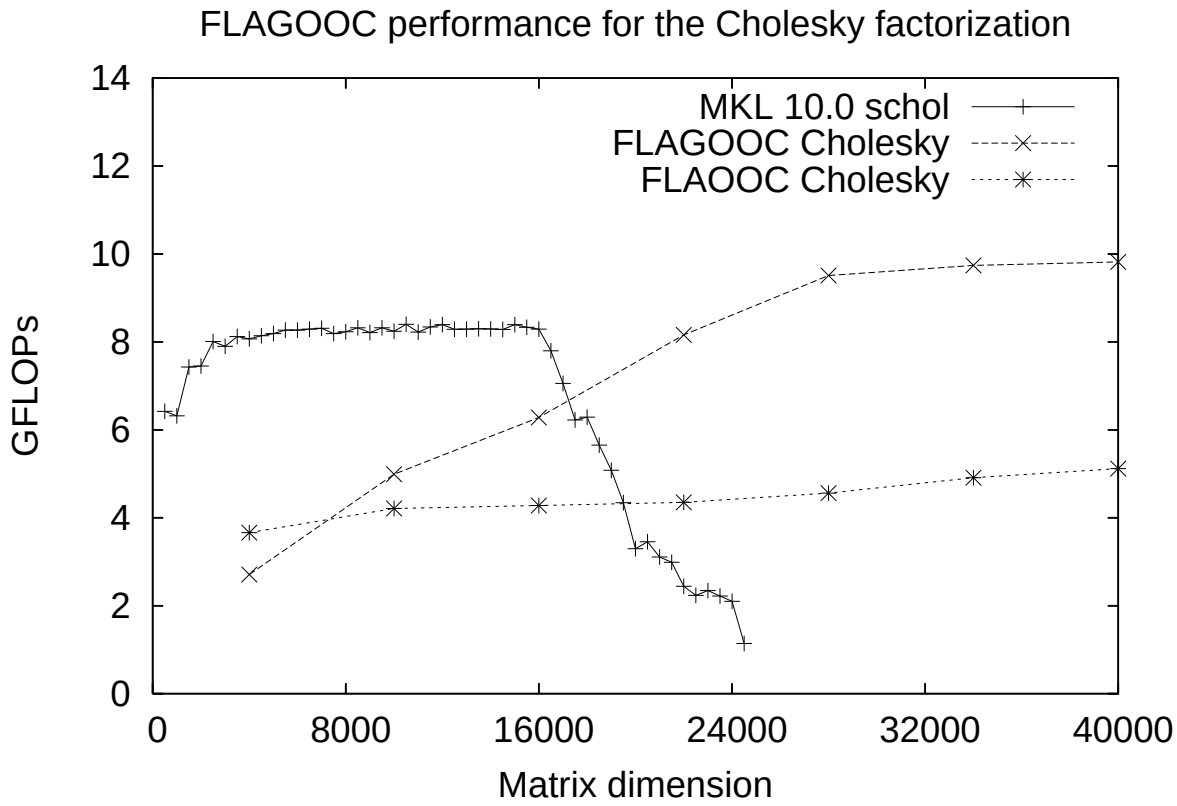
Commodity desktop machine

Processor type	Intel Core2Duo@1.86 Ghz
System memory	1 Gbyte DDR2
Storage	Samsung SP2504D SATA-II
GPU	Nvidia Geforce 8800 Ultra
Video memory	768 Mbytes
OS	Linux 2.6.18
Software	Octave 2.9.19 CUBLAS 1.1 Intel MKL 10

Performances reported in GFLOPs (billions of flops per second)

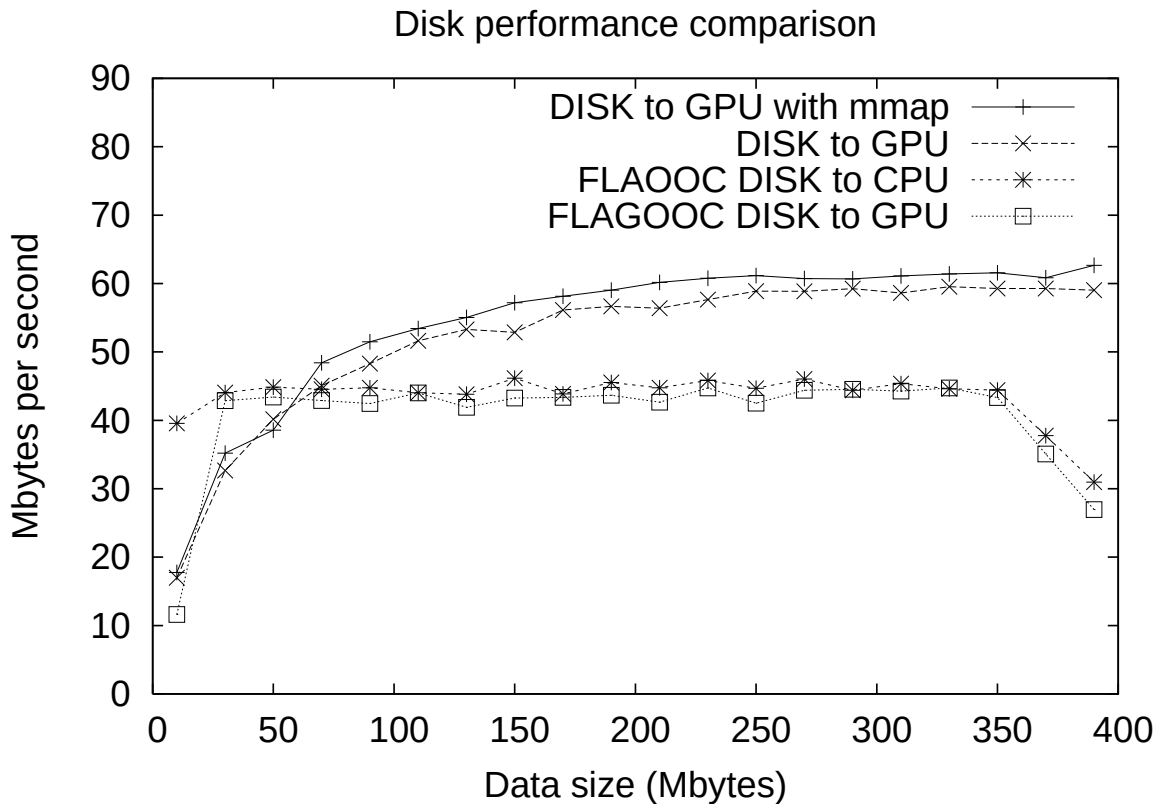
Optimal tile size determined experimentally

Performance of the OOC API



- Compare MKL, FLAOOC and FLAGOOC
- For small matrices, MKL is clearly more efficient
- If matrix exceeds RAM capacity, virtual memory is not efficient
- GPU accelerated version attains interesting speed-ups

Main performance penalties



1. Data transfer bottleneck
2. Single precision to double precision conversions
3. CUBLAS not fully optimized
4. M-script codes, interpreted

Conclusions and further work

- FLAGOOC@lab provides ease of use for dense linear algebra
- Combined with the performance of the most modern GPUs
- Simplicity $\Rightarrow$ FLAME notation
- Ease of use $\Rightarrow$ Matlab-like environments
- Combined with CUBLAS, allows an efficient solution for large-scale OOC problems
- However, serious performance penalties can be solved...
- C implementation of the OOC solution on GPUs, with impressive performance (up to 70 GFLOPS)

Thank you...

For more information (GPGPU)...

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

For more information (FLAME project)...

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