Fast Development of Dense Linear Algebra Codes on Graphics Processors

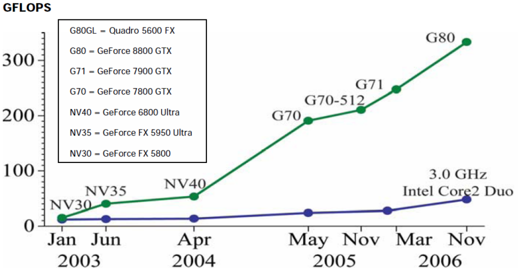
M. Jesús Zafont   Alberto Martín
Francisco Igual   Enrique S. Quintana-Ortí

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

Motivation (I)

Modern graphics processors have become a real and low cost alternative for HPC

Performance

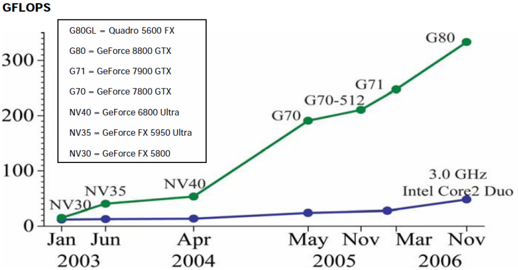


Easy to program

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

Modern graphics processors have become a real and low cost alternative for HPC

Performance

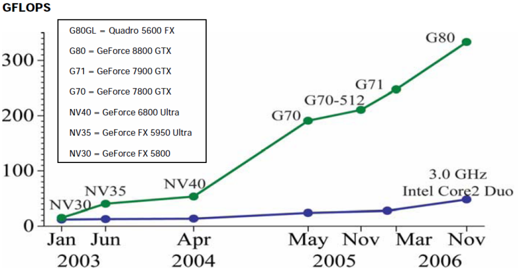


Easy to program

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

Modern graphics processors have become a real and low cost alternative for HPC

Performance



Easy to program

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

Motivation (II)

Goals

1. Allow the development of robust codes
2. Reduce develop time
3. Reduce execution time

Our proposal: FLAG/C

Programming interface for dense linear algebra routines on GPUs

- **Robust codes**: FLAME methodology
- **Ease of use**: hides the underlying architecture
- **Performance**: GPU-based

Motivation (II)

Goals

1. Allow the development of robust codes
2. Reduce develop time
3. Reduce execution time

Our proposal: FLAG/C

Programming interface for dense linear algebra routines on GPUs

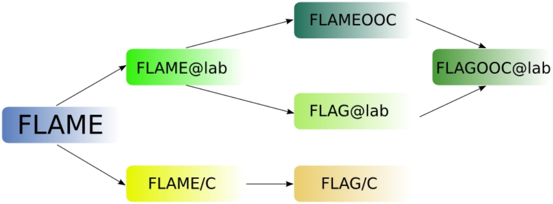
- **Robust codes**: FLAME methodology
- **Ease of use**: hides the underlying architecture
- **Performance**: GPU-based

Outline

The FLAG/C API

FLAG/C

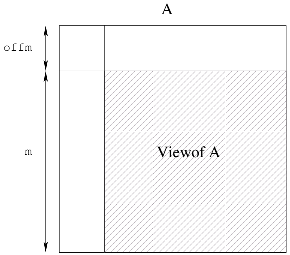
- FLAG/C: high level API, written in C; fast development of efficient dense linear algebra codes.
- Can be combined with any accelerator type: GPUs, ClearSpeed boards, Cell B.E.
- Prerequisite: efficient implementation of BLAS available for the architecture



FLAG/C Objects

FLAG/C Objects

- Object: key concept in FLAG/C
- Representation of a matrix in the accelerator memory
- FLAG/C allows:
  1. Create, destroy and query the object properties
  2. Initialize the contents of an object
  3. Define *views*
  4. Operate with objects through BLAS



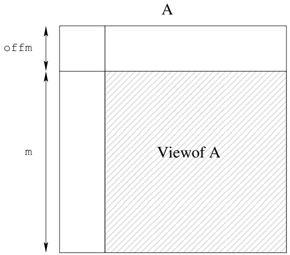
Views

- Block references inside an object
- Flexibility to handle matrices and matrix blocks
- Without additional storage cost

FLAG/C Objects

FLAG/C Objects

- Object: key concept in FLAG/C
- Representation of a matrix in the accelerator memory
- FLAG/C allows:
  1. Create, destroy and query the object properties
  2. Initialize the contents of an object
  3. Define *views*
  4. Operate with objects through BLAS



Views

- Block references inside an object
- Flexibility to handle matrices and matrix blocks
- Without additional storage cost

Typical structure of a FLAG/C program (I)

Initialization

`FLA_Init()` / `FLA_Finalize()`

Initialization / finalization of the environment

↓

Object creation

`FLAG_Obj_create( FLA_TYPE, int m, int n, FLA_Obj * )`

Allocates space for the object in the GPU memory

↓

Typical structure of a FLAG/C program (I)

Initialization

`FLA_Init()` / `FLA_Finalize()`

Initialization / finalization of the environment

↓

Object creation

`FLAG_Obj_create( FLA_TYPE, int m, int n, FLA_Obj * )`

Allocates space for the object in the GPU memory

↓

Typical structure of a FLAG/C program (II)

↓

CPU -> GPU Transfers

`FLAG_Obj_set_with_buffer( void* buffer, FLA_Obj A )`

Initializes the elements of object A in GPU memory with a copy of the contents of *buffer* in main memory

↓

Matrix Computations

Complete BLAS implementation

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

Typical structure of a FLAG/C program (II)

↓

CPU -> GPU Transfers

```
FLAG_Obj_set_with_buffer( void* buffer, FLA_Obj A )
```

Initializes the elements of object A in GPU memory with a copy of the contents of *buffer* in main memory

↓

Matrix Computations

Complete BLAS implementation

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

Typical FLAG/C program structure (III)

↓

Utils

`FLAG_Obj_show( char *s1, FLAG_Obj A, char * format, char *s2 )`

Shows the contents of object A

↓

GPU -> CPU Transfers

`FLAG_Obj_transfer_to_FLA_Obj( FLAG_Obj A,FLA_Obj B )`

Initializes the elements of an object B with a copy of the contents of A, stored in GPU

Typical FLAG/C program structure (III)

↓

Utils

`FLAG_Obj_show( char *s1, FLAG_Obj A, char * format, char *s2 )`

Shows the contents of object A

↓

GPU -> CPU Transfers

`FLAG_Obj_transfer_to_FLA_Obj( FLAG_Obj A,FLA_Obj B )`

Initializes the elements of an object B with a copy of the contents of A, stored in GPU

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

```
FLAG_Init();

FLAG_Obj AObj, BObj, CObj;

FLAG_Obj_create( 'FLAG_FLOAT', n, n, &AObj );
FLAG_Obj_create( 'FLAG_FLOAT', n, n, &BObj );
FLAG_Obj_create( 'FLAG_FLOAT', n, n, &CObj );

FLAG_Obj_set_with_buffer( (void *) A, &AObj );
FLAG_Obj_set_with_buffer( (void *) B, &BObj );
FLAG_Obj_set_with_buffer( (void *) C, &CObj );

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

FLAG_Obj_show( "C = [", Cobj, "%f", "]" );

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

FLAG_Finalize();
```

Initialization

Object creation

CPU -> GPU Transfers

BLAS operations on GPU

GPU memory free

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

```
FLAG_Init();

FLAG_Obj  AObj,  BObj,  CObj;

FLAG_Obj_create(  'FLAG_FLOAT',  n,  n,  &AObj  );
FLAG_Obj_create(  'FLAG_FLOAT',  n,  n,  &BObj  );
FLAG_Obj_create(  'FLAG_FLOAT',  n,  n,  &CObj  );

FLAG_Obj_set_with_buffer(  (void *) A,  &AObj  );
FLAG_Obj_set_with_buffer(  (void *) B,  &BObj  );
FLAG_Obj_set_with_buffer(  (void *) C,  &CObj  );

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

FLAG_Obj_show(  "C = [",  Cobj,  "%f",  "]"  );

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

FLAG_Finalize();
```

Initialization

Object creation

CPU -> GPU Transfers

BLAS operations on GPU

GPU memory free

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

```
FLAG_Init();

FLAG_Obj AObj, BObj, CObj;

FLAG_Obj_create( 'FLAG_FLOAT', n, n, &AObj );
FLAG_Obj_create( 'FLAG_FLOAT', n, n, &BObj );
FLAG_Obj_create( 'FLAG_FLOAT', n, n, &CObj );

FLAG_Obj_set_with_buffer( (void *) A, &AObj );
FLAG_Obj_set_with_buffer( (void *) B, &BObj );
FLAG_Obj_set_with_buffer( (void *) C, &CObj );

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

FLAG_Obj_show( "C = [", Cobj, "%f", "]" );

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

FLAG_Finalize();
```

- Initialization
- Object creation
- CPU -> GPU Transfers
- BLAS operations on GPU
- GPU memory free

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

```
FLAG_Init();

FLAG_Obj AObj, BObj, CObj;

FLAG_Obj_create( 'FLAG_FLOAT', n, n, &AObj );
FLAG_Obj_create( 'FLAG_FLOAT', n, n, &BObj );
FLAG_Obj_create( 'FLAG_FLOAT', n, n, &CObj );

FLAG_Obj_set_with_buffer( (void *) A, &AObj );
FLAG_Obj_set_with_buffer( (void *) B, &BObj );
FLAG_Obj_set_with_buffer( (void *) C, &CObj );

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

FLAG_Obj_show( "C = [", Cobj, "%f", "]" );

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

FLAG_Finalize();
```

Initialization

Object creation

CPU -> GPU Transfers

BLAS operations on GPU

GPU memory free

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

```
FLAG_Init();

FLAG_Obj AObj, BObj, CObj;

FLAG_Obj_create( 'FLAG_FLOAT', n, n, &AObj );
FLAG_Obj_create( 'FLAG_FLOAT', n, n, &BObj );
FLAG_Obj_create( 'FLAG_FLOAT', n, n, &CObj );

FLAG_Obj_set_with_buffer( (void *) A, &AObj );
FLAG_Obj_set_with_buffer( (void *) B, &BObj );
FLAG_Obj_set_with_buffer( (void *) C, &CObj );

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

FLAG_Obj_show( "C = [", Cobj, "%f", "]" );

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

FLAG_Finalize();
```

Initialization

Object creation

CPU -> GPU Transfers

BLAS operations on GPU

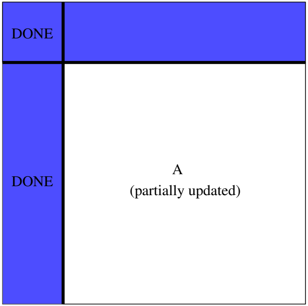
GPU memory free

FLAME

FLAME

- FLAME: fast development of robust and efficient libraries with a high abstraction level
- Focuses on dense and banded linear algebra
- High level of abstraction: hides implementation details
- It is not just a library:
  - Notation to express dense linear algebra algorithms
  - Methodology for systematic algorithm derivation
  - Programming interfaces (APIs) to represent algorithms in code almost effortlessly
  - Tools to ease the algorithmic derivation process
- Example: Cholesky factorization

Cholesky Factorization with FLAME



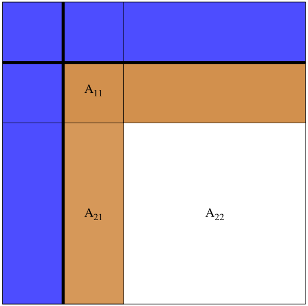
Definition

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

$$A = LL^T$$

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

Cholesky factorization with FLAME

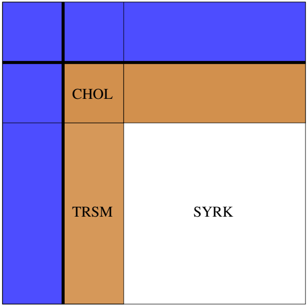


Repartition

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

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

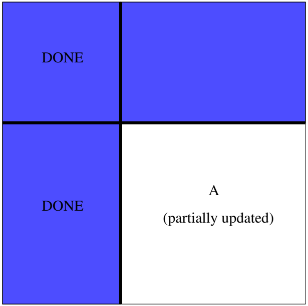
Cholesky Factorization with FLAME



Operations per iteration:

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

Cholesky Factorization with FLAME

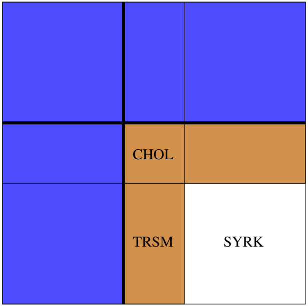


Continue with

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

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

Cholesky Factorization with FLAME



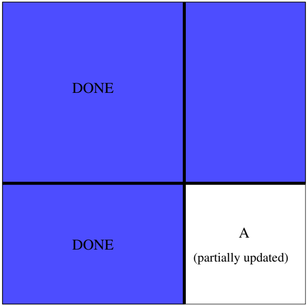
Repartition

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

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

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

Cholesky Factorization with FLAME



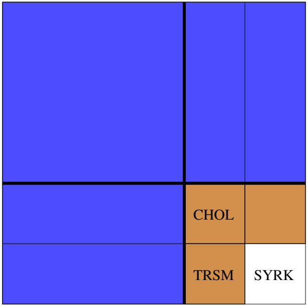
Continue with

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

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

```
FLA_Cont_with_3x3_to_2x2(
  &ATL, &ATR,      A00, A01, A02,
                   A10, A11, A12,
  &ABL, &ABR,      A20, A21, A22,
                   FLA_TL );
```

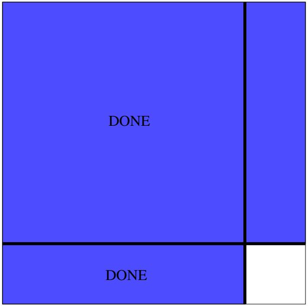
Cholesky Factorization with FLAME



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 and FLAME/C Notation

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 $n_b \times n_b$

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

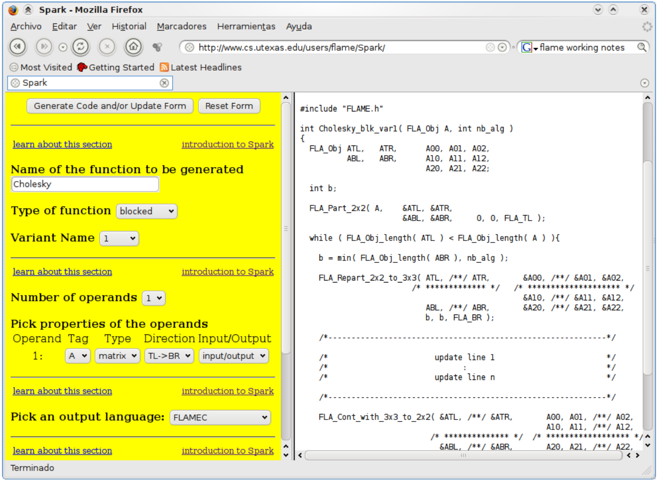
Continue with

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

endwhile

```
int FLA_Chol_l_blk_var3_imp1( FLAG_Obj A, int nb_alg )
{
  FLA_Part_2x2( A,   &ATL, &ATR,
                     &ABL,  &ABR,
                     0, 0, FLAG_TL );

  while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) ){
    b = min( min( FLA_Obj_length( ABR ),
                  FLA_Obj_width( 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_Chol_unb( A11 );

    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              FLA_ONE, A11, A21 );

    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A21, FLA_ONE, A22 );
    /* ***************************************** */

    FLA_Cont_with_3x3_to_2x2(
        &ATL, &ATR,       A00, A01, A02,
                          A10, A11, A12,
        &ABL, &ABR,       A20, A21, A22,
                          FLA_TL );
  }
}
```

Getting the FLAME/C code: Spark



The Spark tool eases the development process by automatically generating code skeletons

FLAME/C - FLAG/C Comparison

```
int FLAG_Chol_l_blk_var3_imp1( FLAG_Obj A, int nb_alg )
{
  FLAG_Part_2x2( A,   &ATL, &ATR,
                      &ABL,  &ABR,
                      0, 0, FLAG_TL );

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

    FLAG_Repart_2x2_to_3x3(
        ATL, ATR,        &A00, &A01, &A02,
                         &A10, &A11, &A12,
        ABL, ABR,        &A20, &A21, &A22,
                         b,   b, FLAG_BR );

    /* ************************************* */
    FLAG_Chol_unb( A11 );

    FLAG_Trsm( FLAG_RIGHT, FLAG_LOWER_TRIANGULAR,
               FLAG_TRANSPOSE, FLAG_NONUNIT_DIAG,
               FLA_ONE, A11, A21 );

    FLAG_Syrk( FLAG_LOWER_TRIANGULAR, FLAG_NO_TRANSPOSE,
               FLA_MINUS_ONE, A21, FLA_ONE, A22 );
    /* ************************************* */

    FLAG_Cont_with_3x3_to_2x2(
        &ATL, &ATR,        A00, A01, A02,
                           A10, A11, A12,
        &ABL, &ABR,        A20, A21, A22,
                           FLA_TL );
  }
}
```

Main differences:

1. BLAS routines provided by FLAG/C
2. Matrices must have been transferred to the GPU

FLAG/C evaluation

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**

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 1×1

Variant 1:
$a_{10}^T := a_{10}^T \mathrm{TRIL}(A_{00})^{-T}$
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{10}$
$\alpha_{11} := \sqrt{\alpha_{11}}$

Variant 2:
$\alpha_{11} := \alpha_{11} - a_{10}^T a_{10}$
$\alpha_{11} := \sqrt{\alpha_{11}}$
$a_{21} := a_{21} - A_{20} a_{10}$
$a_{21} := a_{21} / \alpha_{11}$

Variant 3:
$\alpha_{11} := \sqrt{\alpha_{11}}$
$a_{21} := a_{21} / \alpha_{11}$
$A_{22} := A_{22} - a_{21} a_{21}^T$

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

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 $n_b \times n_b$

Variant 1:
$A_{10} := A_{10} \mathrm{TRIL}(A_{00})^{-T}$
$A_{11} := A_{11} - A_{10} A_{10}^T$
$A_{11} := \mathrm{CHOL\_UNB}(A_{11})$

Variant 2:
$A_{11} := A_{11} - A_{10} A_{10}^T$
$A_{11} := \mathrm{CHOL\_UNB}(A_{11})$
$A_{21} := A_{21} - A_{20} A_{10}^T$
$A_{21} := A_{21} \mathrm{TRIL}(A_{11})^{-T}$

Variant 3:
$A_{11} := \mathrm{CHOL\_UNB}(A_{11})$
$A_{21} := A_{21} \mathrm{TRIL}(A_{11})^{-T}$
$A_{22} := A_{22} - A_{21} A_{21}^T$

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

- Square root calculations $\rightarrow$ CPU

FLAG/C evaluation

Implementation family Imp1

- Transfer of diagonal elements
  - Calculations
    - if $n_b \ll n \rightarrow$ lots of calls to level-1 and level-2 CUBLAS with low computational load
    - if n_b is large, the number of operations expressed in terms of level-3 CUBLAS is reduced
  - Communications
    - $T_1(n) = n(2\alpha + 8\beta)$ where $\alpha \gg \beta$
    - $2n$ transfers of 4 bytes each

Implementation family Imp2 (Imp1 generalization)

- Transfer of diagonal blocks
- Calculations: level-3 CUBLAS calls
- Communications
  - $T_2(n, n_b) = \frac{n}{n_b}(2\alpha + 8\beta n_b^2)$
  - $2\frac{n}{n_b}$ transfers of $4n_b^2$ bytes

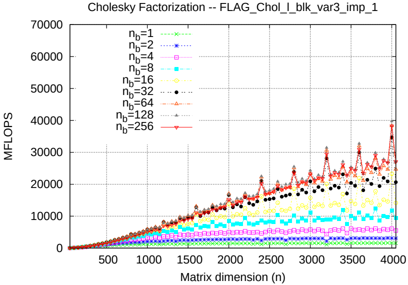
FLAG/C evaluation

- Used metric: $\text{MFLOPs} = \frac{n^3}{3\times 10^6 \times T_{ejec}}$
- T_{ejec} is the time used by the processor to calculate the factorization. It includes:
  - Transfer times of the matrix (CPU $\rightarrow$ GPU)
  - Transfer times of the Cholesky factor (GPU $\rightarrow$ CPU)

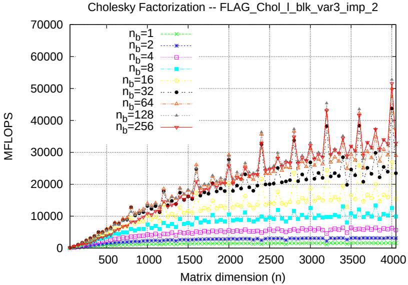
	CPU	GPU
Processor	Intel Core 2 Duo	NVIDIA 8800 Ultra
Model	Crusoe E6320	G80
Clock frequency	1.86 GHz	575 MHz
Memory frequency	2×333 MHz	2×900 MHz
Bus bandwidth	64 bits	384 bits
Max. bandwidth	5.3 GB/s	86.4 GB/s
Memory	1024 MB DDR2	768 MB GDDR3
Bus type	PCI Express x16 (4 GB/s)	
BLAS version	Intel MKL 10	CUBLAS 2.0

FLAG/C evaluation: Imp1 vs. Imp2

Imp1 Var3 CHOL_BLK



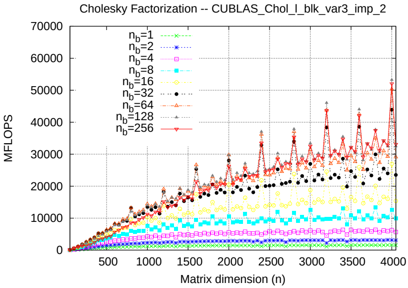
Imp2 Var3 CHOL_BLK



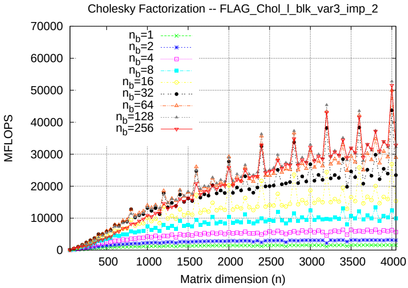
- Important improvement when $n \in [0, 2000]$
- Conclusions:
    - Versatility to get high performance implementations

FLAG/C evaluation: overhead

CUBLAS
Imp2 Var3 CHOL_BLK



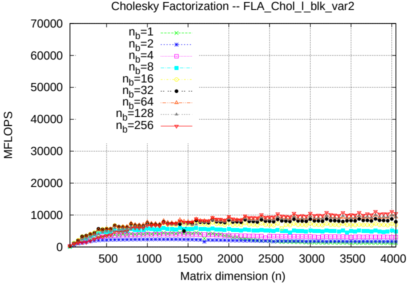
FLAG/C
Imp2 Var3 CHOL_BLK



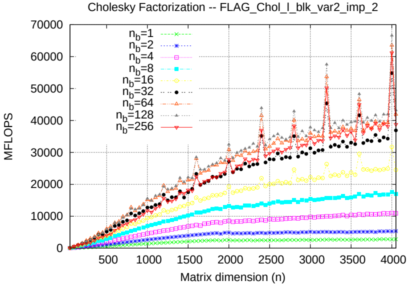
- The overhead introduced by FLAG/C is small

FLAG/C evaluation: FLAME/C vs. FLAG/C

FLAME/C
Var2 CHOL_BLK



FLAG/C
Imp2 Var2 CHOL_BLK



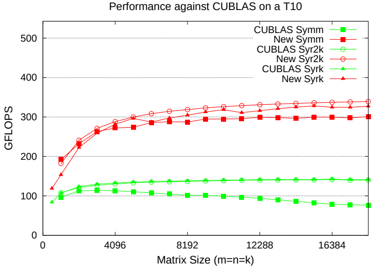
- The GPU can accelerate the calculation significantly ($7x$ in our example)

The programmability problem (I)

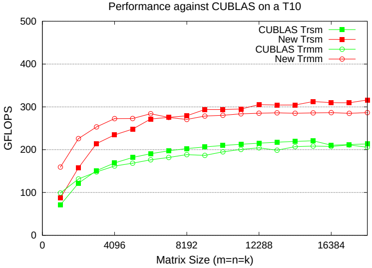
- Programmability: difficult to measure
- Sucess story: alternative level-3 (CU)BLAS implementation using FLAME
- GEMM in CUBLAS is the only fully tuned routine
- Our goal: develop different GEMM-based variants for each BLAS-3 routine in CUBLAS

The programmability problem (II)

Our SSYRK, SSYMM, SSYR2K vs. CUBLAS



Our STRSM, STRMM vs. CUBLAS



- It took few hours to develop and test all the routines, algorithmic variants and block sizes.
- More information: FLAME Working Note #37.

Conclusion

- FLAG/C is easy to use:
  - The learning curve of FLAG/C is minimal
  - High abstraction level for the development of linear algebra codes
- FLAG/C is efficient:
  - Allows the exploitation of the massive multithread parallelism available on modern GPUs
  - Can attain up to $7x$ speedup compared with the CPU (for the shown Cholesky factorization)
- FLAG/C (FLAME) is flexible:
  - The interface can be adapted to any type of hardware accelerator

Conclusion

- FLAG/C is easy to use:
  - The learning curve of FLAG/C is minimal
  - High abstraction level for the development of linear algebra codes
- FLAG/C is efficient:
  - Allows the exploitation of the massive multithread parallelism available on modern GPUs
  - Can attain up to $7x$ speedup compared with the CPU (for the shown Cholesky factorization)
- FLAG/C (FLAME) is flexible:
  - The interface can be adapted to any type of hardware accelerator

Conclusion

- FLAG/C is easy to use:
  - The learning curve of FLAG/C is minimal
  - High abstraction level for the development of linear algebra codes
- FLAG/C is efficient:
  - Allows the exploitation of the massive multithread parallelism available on modern GPUs
  - Can attain up to $7x$ speedup compared with the CPU (for the shown Cholesky factorization)
- FLAG/C (FLAME) is flexible:
  - The interface can be adapted to any type of hardware accelerator

Thank you...

More information. . .

http://www3.uji.es/˜figual
http://www.hpca.uji.es
http://www.cs.utexas.edu/˜flame