Matrix Computations on GPUs, multiple GPUs and clusters of GPUs

Francisco D. Igual

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

Motivation

Evolution of GPU architectures through the years

Mono-GPU systems

Software

- CUDA
- OpenCL
- OpenGL, Cg, . . .
- CUBLAS.

Multi-GPU systems

Software

- GPUSs
- Star-PU
- MAGMA??
- FLAME SuperMatrix

Clusters of GPUs

Software

- ??
- Our approach: CUPLAPACK

Evolution of GPU architectures through the years

Mono-GPU systems

Software

- CUDA
- OpenCL
- OpenGL, Cg, . . .
- CUBLAS.

Multi-GPU systems

Software

- GPUSs
- Star-PU
- MAGMA??
- FLAME SuperMatrix

Clusters of GPUs

Software

- ??
- Our approach: CUPLAPACK

Evolution of GPU architectures through the years

Mono-GPU systems

Software

- CUDA
- OpenCL
- OpenGL, Cg, . . .
- CUBLAS.

Multi-GPU systems

Software

- GPUSs
- Star-PU
- MAGMA??
- FLAME SuperMatrix

Clusters of GPUs

Software

- ??
- Our approach: CUPLAPACK

Evolution of GPU architectures through the years

Mono-GPU systems

Multi-GPU systems

Clusters of GPUs

Software

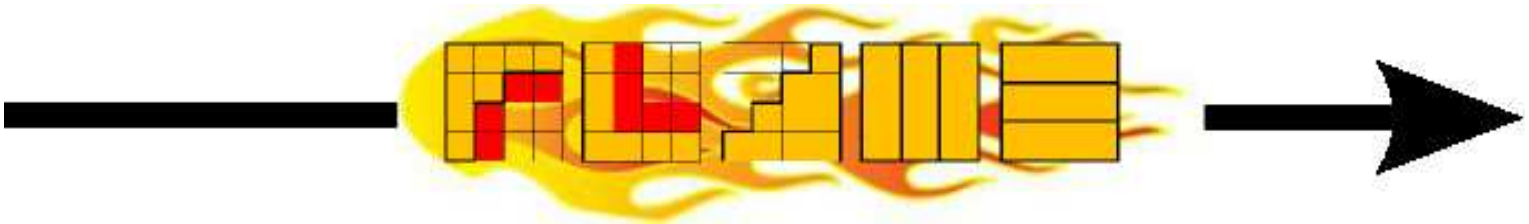
- CUDA
- OpenCL
- OpenGL, Cg, . . .
- CUBLAS.

Software

- GPUSs
- Star-PU
- MAGMA??
- FLAME SuperMatrix

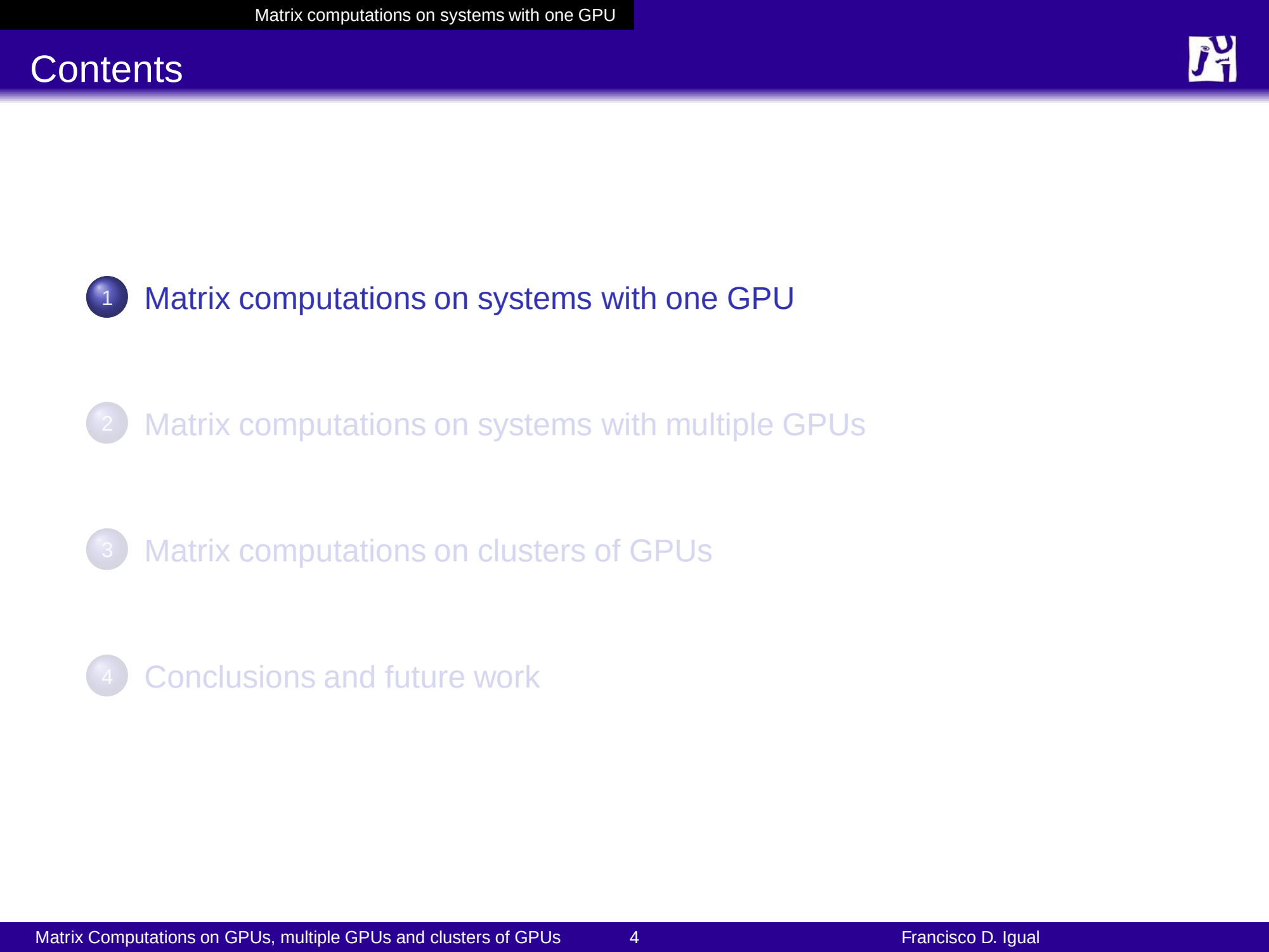
Software

- ??
- Our approach: CUPLAPACK



Outline

Contents



Motivation. CUBLAS evaluation (I)

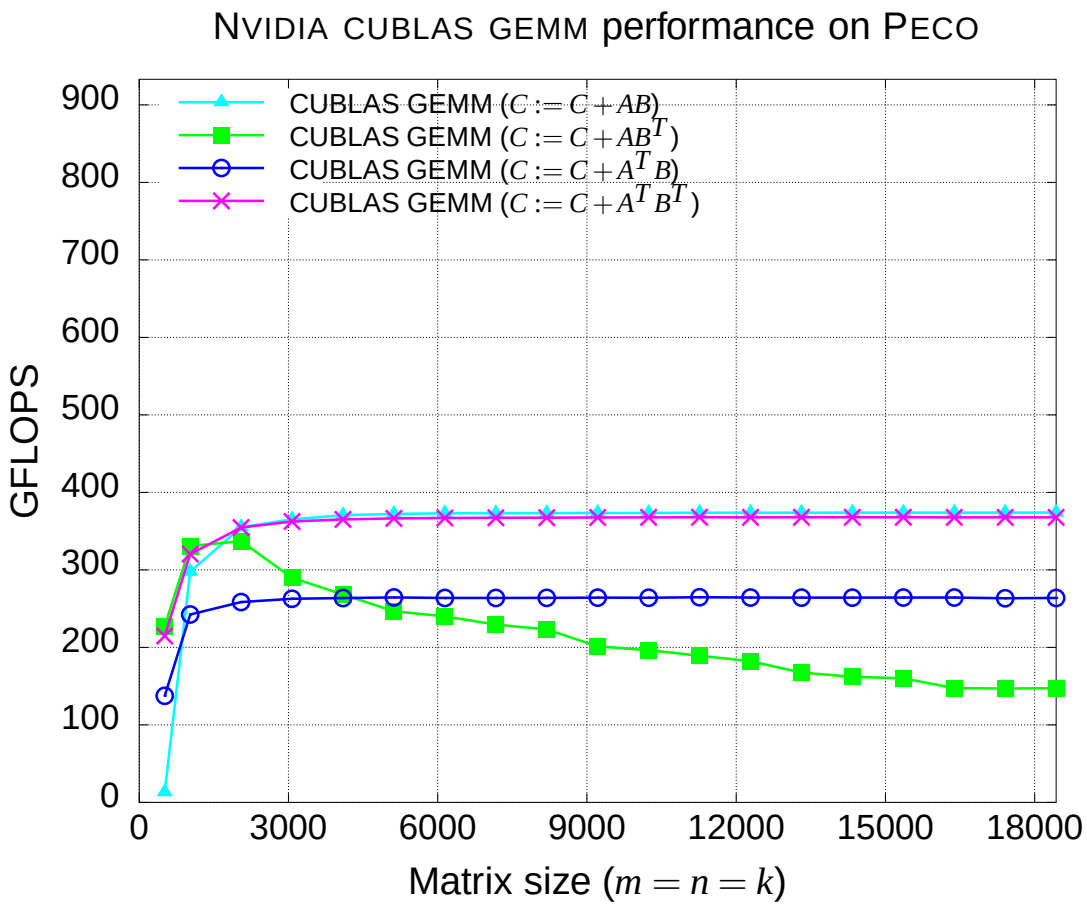
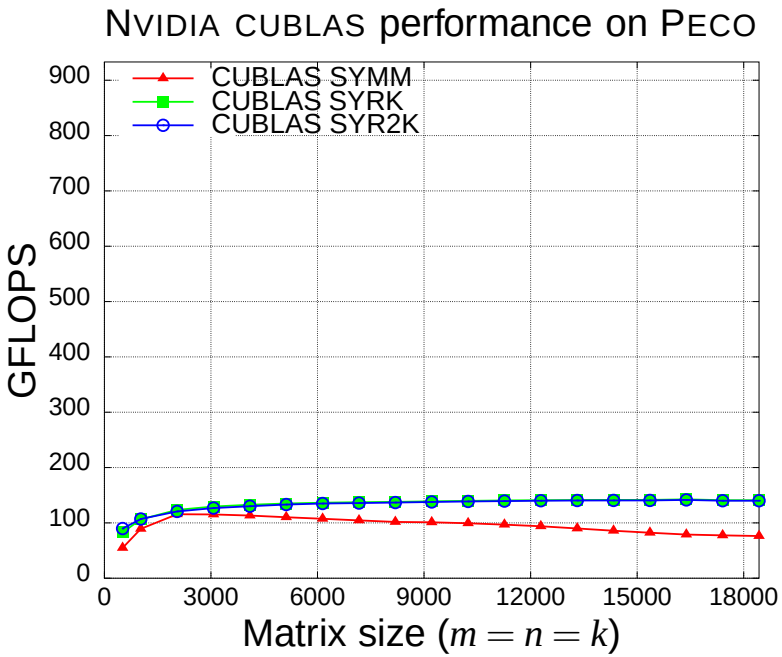


Figure: Performance of the GEMM routine of NVIDIA CUBLAS for square matrices.

Motivation. CUBLAS evaluation (II)



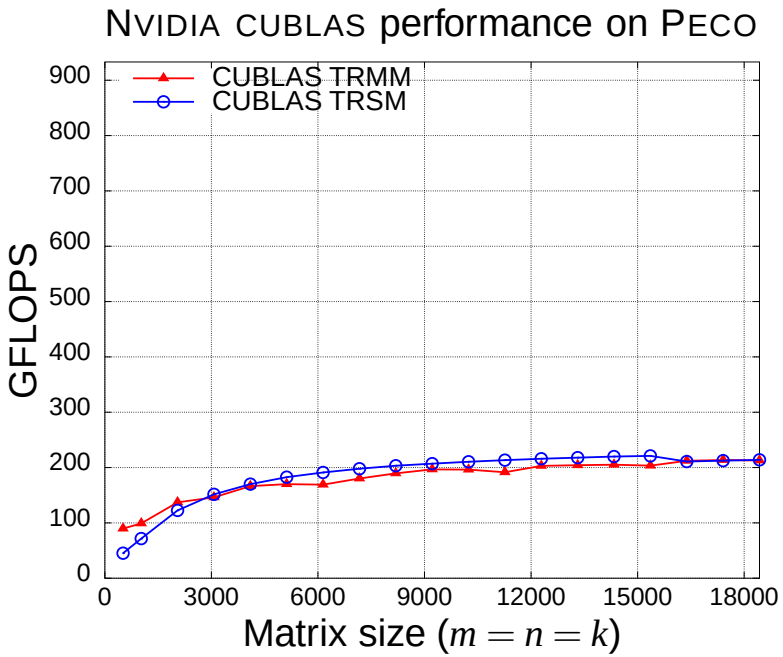


Figure: Performance of the Level-3 routines in NVIDIA CUBLAS for square matrices. Left: SYMM, SYRK and SYR2K. Right: TRMM and TRSM.

Motivation

The problem

Routines in CUBLAS are far from being equally optimized

The idea

- Assembly code is not easy $\longrightarrow$ Efficient CUDA is not easy.
- Can we tune CUBLAS without writing a line of CUDA code?

The solution

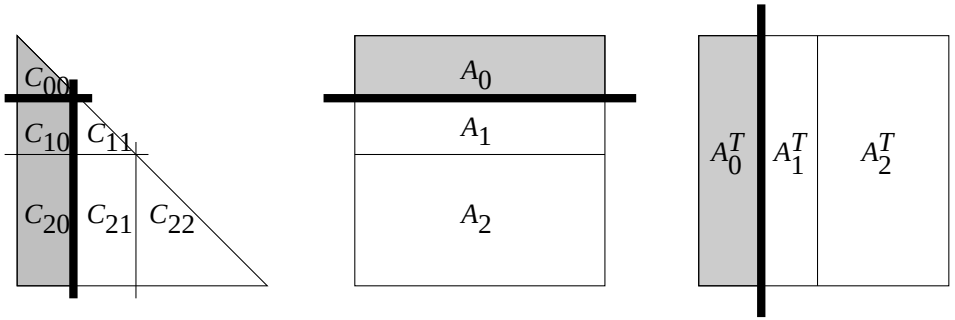
- Revisit the work:

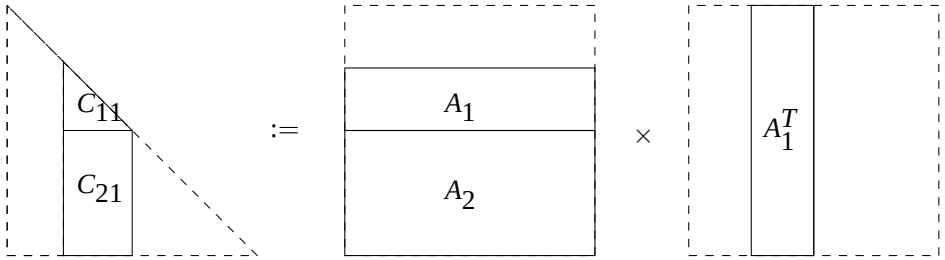
  KÅGSTRÖM, B., LING, P., AND LOAN, C. V.
  GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark.
  ACM Trans. Math. Softw. 24, 3 (1998), 268–302.
- Combine it with the FLAME methodology:
  - Systematic evaluation of algorithmic variants and block sizes

GEMM-based implementations for CUBLAS

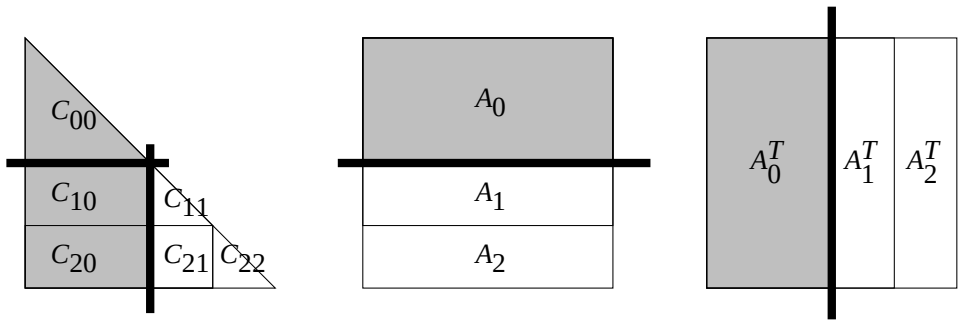
Step 1: Partitioning before iteration



Step 2: Computation in iteration



Step 3: Advancement of partitioning for next iteration



GEMM-based implementations for CUBLAS

Algorithm: SYRK_MP_PM(C, A)

Partition

$$C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right), A \rightarrow \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right)$$

where C_{TL} is 0×0, A_T has 0 rows

while $m(C_{TL}) < m(C)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \rightarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

where C_{11} is $b \times b$, A_1 has b rows

Syrk_mp

$C_{11} := C_{11} - A_1 A_1^T$ (SYRK)

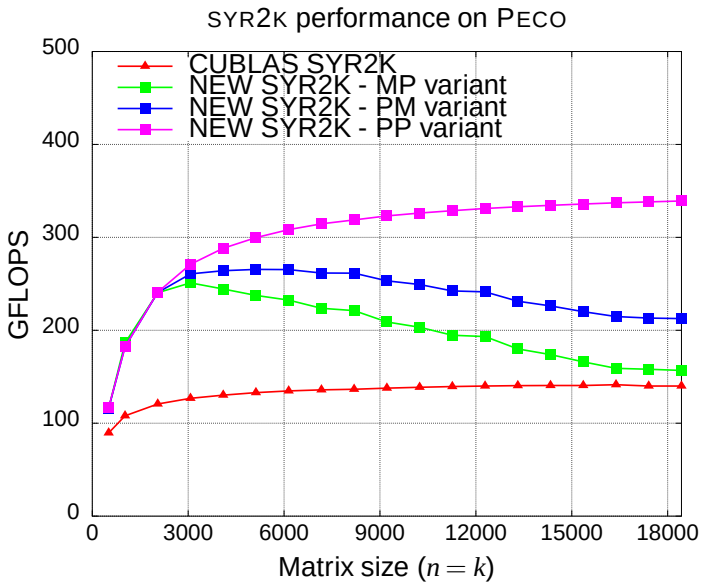
$C_{21} := C_{21} - A_2 A_1^T$ (GEMM)

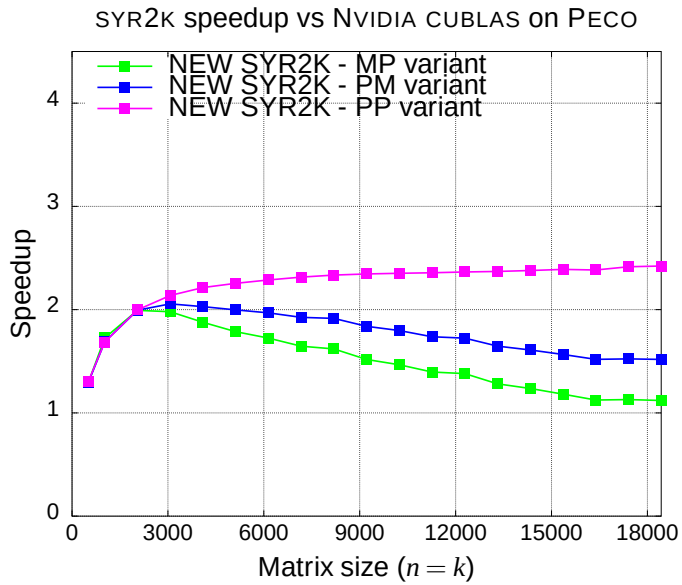
Continue with

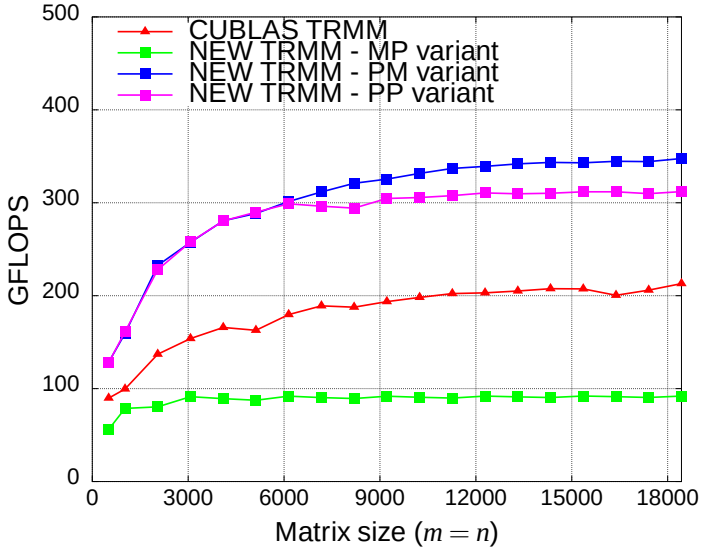
$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right), \left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) \leftarrow \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right)$$

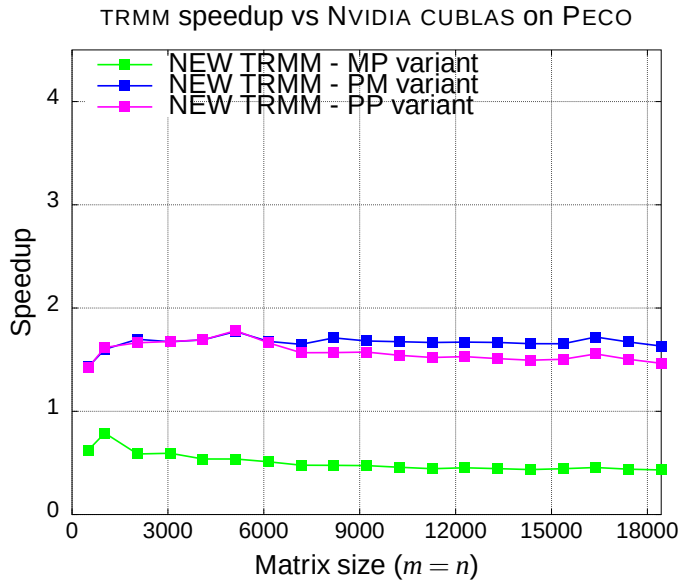
endwhile

Some experimental results

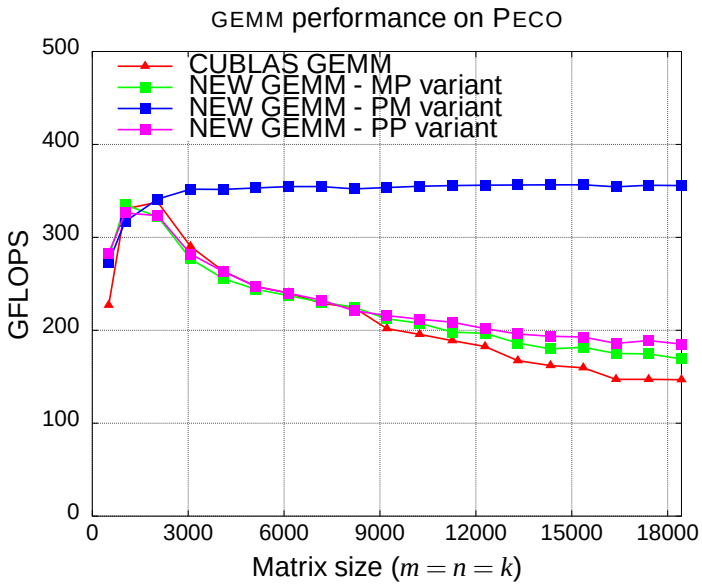


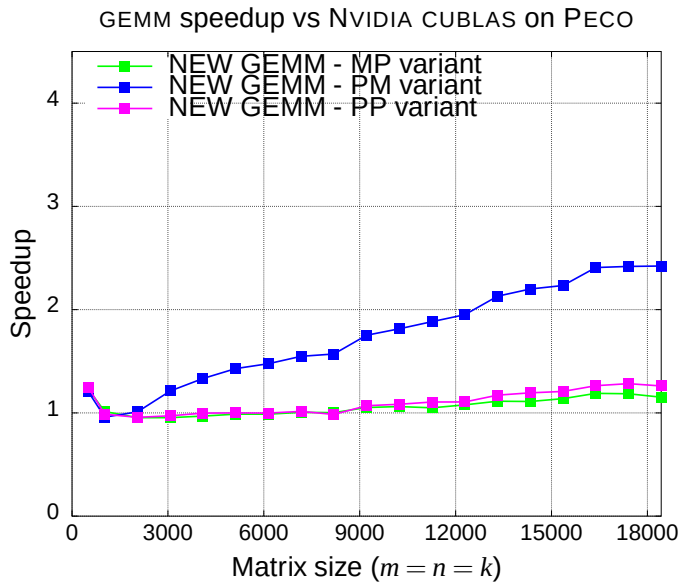


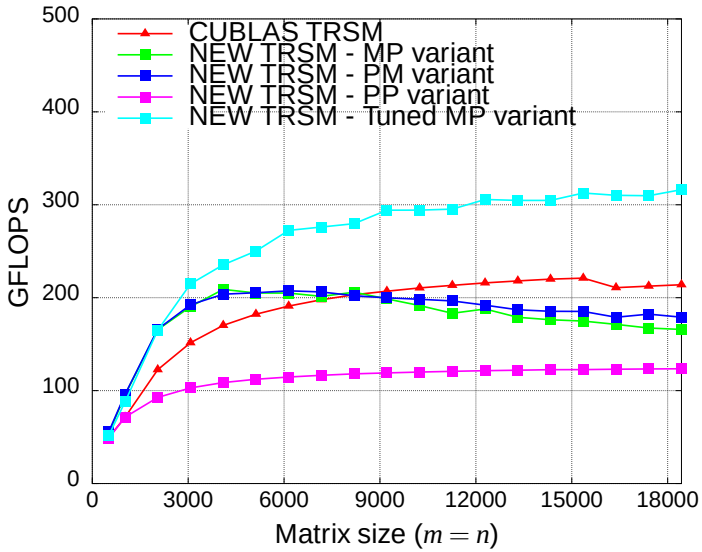


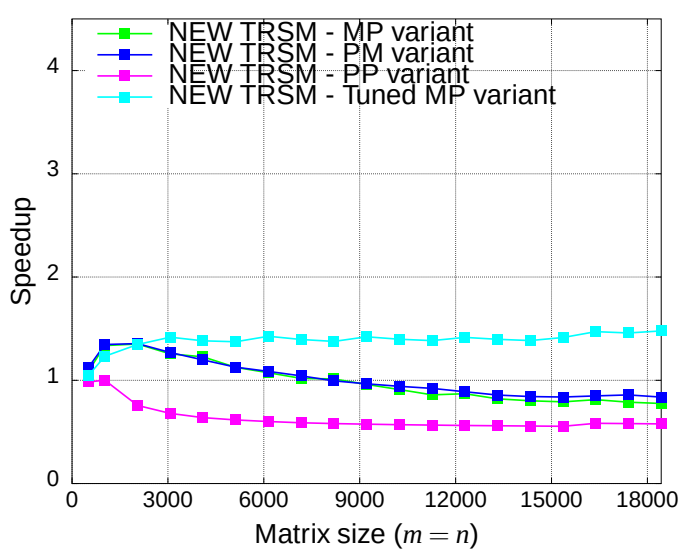


Some experimental results









Advantages of the high-level approach

- Our insights (optimal block size, best algorithmic variant, . . .) can be directly translated to CUDA code
- Straightforward implementation to future architectures
- We just need an optimized CUBLAS GEMM to adapt it to future architectures

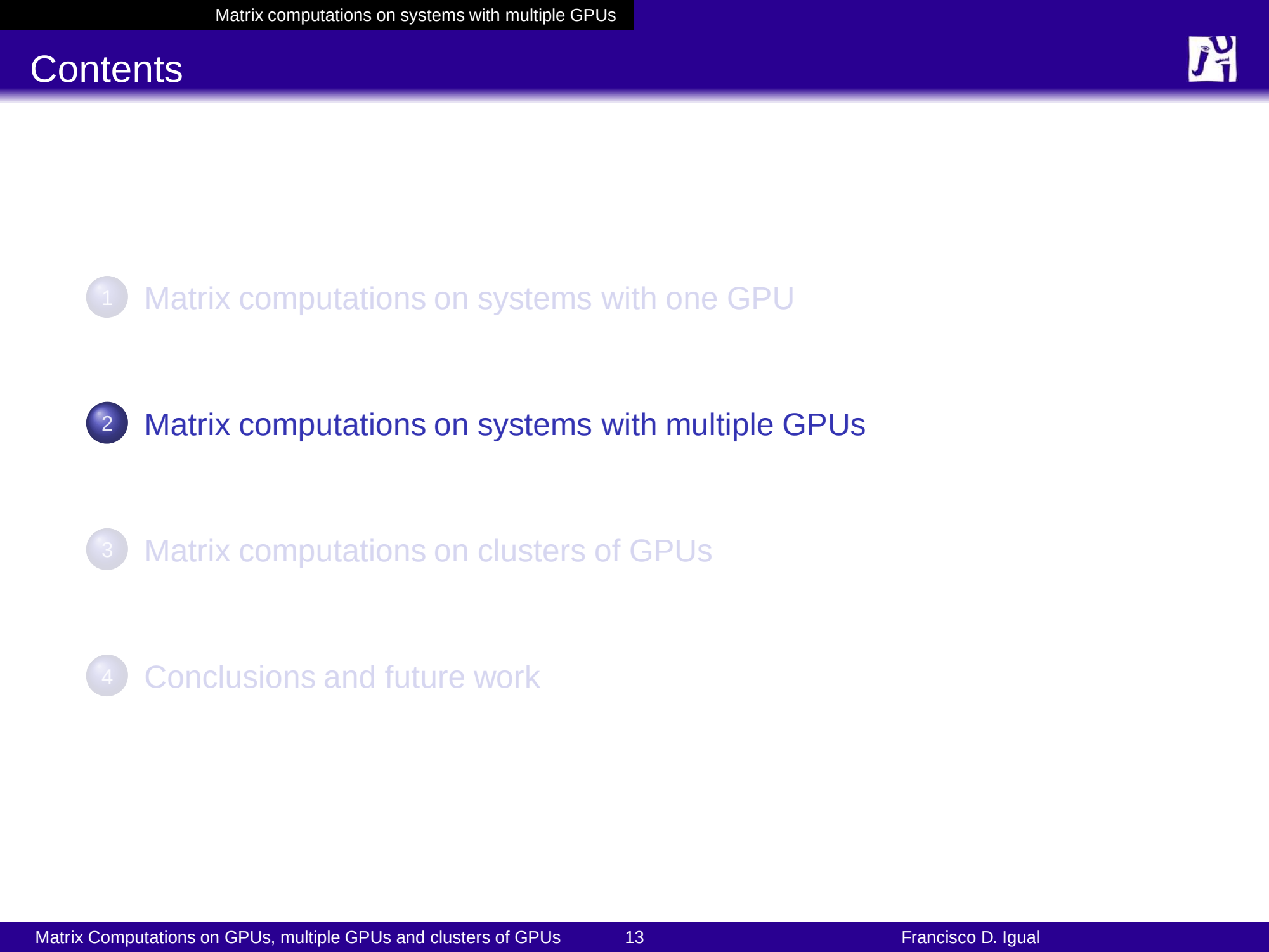
Example

Future CUBLAS 3.2 will be shipped with a (re)tuned GEMM:

RAJIB NATH, STANIMIRE TOMOV, AND JACK DONGARRA
An Improved MAGMA GEMM for Fermi GPUs.
LAWN #227, July 29, 2010.

Thank you!!
We now have a full BLAS tuned for Fermi

Contents



Motivation

- First work with runtime support on multi-GPU systems (2008)

  QUINTANA, G., IGUAL, F., QUINTANA, E. S., AND VAN DE GEIJN, R.
  Solving dense linear systems on platforms with multiple hardware accelerators.
  In *PPoPP'09*.

- Related works on multi-core: SuperMatrix, CellSs, SMPSs, PLASMA.
- MAGMA is still not ready for multi-GPU support.
- Task-level parallelism.
- Particularities of multi-GPU systems:
  - Goal: Reduction of data transfers.
  - Goal: Transparent data transfers for the programmer.
- Optimizations:
  - Cache systems + memory coherence policies.
  - Scheduling policies.

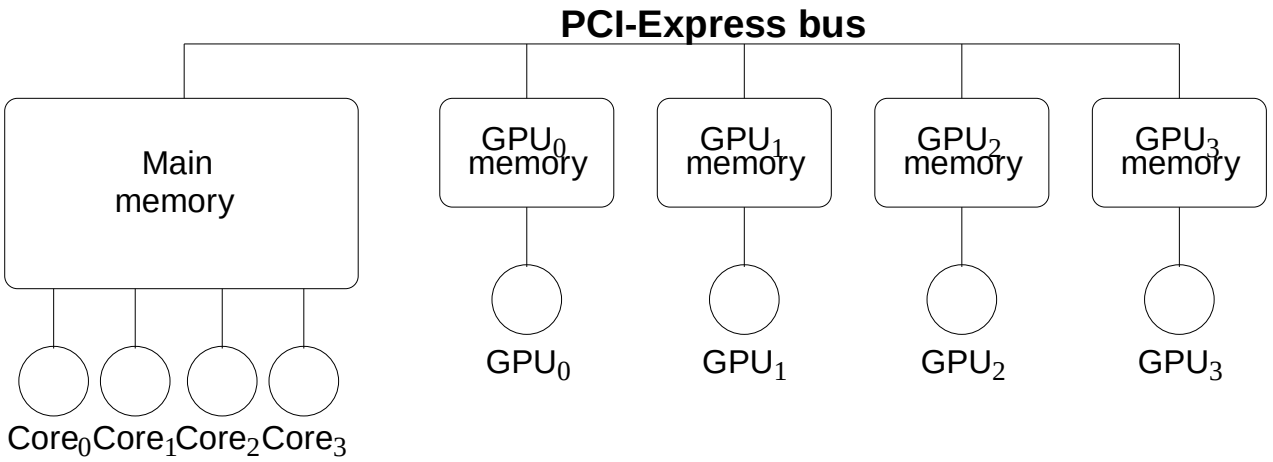
Multi-core vs. Multi-GPU. Analogies

	Multi-core	**Cell B.E.**	**Multi-GPU**
Execution Unit	1 core	1 SPU	**1 GPU**
Tasks	Serial BLAS calls	Serial BLAS calls	**CUBLAS**
Paradigm	Shared memory	Distributed memory	**Distributed memory**
Example	SuperMatrix, SMPSs	CellSs	GPUSs, **SuperMatrix**

Modifications to SuperMatrix

- Each CPU thread controls one GPU
- (Possibly) other threads to control additional cores
- Data transfers must be managed by the runtime

Distributed-memory approach



Problems

- Separate memory spaces
- No direct GPU-GPU transfers. Communication through RAM
- PCI-Express is the main bottleneck

Our solutions

- Memory coherence protocols
- Better scheduling policies

Memory coherence protocols (I)

Version 1: Basic approach

1. Task issued to a GPU
2. Input blocks are transferred to GPU
3. Output blocks are transferred back to RAM and destroyed

☒Unnecessary transfers

Version 2: 2D data affinity

1. Concepts of *own* and *alien* blocks. 2D affinity. Owner computes.
2. Own blocks are transferred back to RAM
3. Own blocks are kept in GPU memory after execution

☑Reduction in number of writes (*own* blocks)
☒Still unnecessary writes (*alien* blocks)

Memory coherence protocols (II)

Version 3: Software cache of alien blocks + Write-invalidate

1. Each GPU keeps a cache of *alien* blocks
2. Blocks in other caches are invalidated in a write
3. Own blocks are kept in GPU memory after execution
4. Own blocks are transferred inmediately to RAM (write-through)

☑Reduction in number of writes (*alien* blocks)
☒Still unnecesary reads (*own* blocks)

Version 4: Software cache of alien blocks + Write-back

1. Each GPU keeps a cache of *alien* blocks
2. Blocks in other caches are invalidated in a write
3. Own blocks are kept in GPU memory after execution
4. Own blocks are transferred to RAM when needed (write-back)

☑Reduction in number of reads (*own* blocks)

Reduction in the number of transfers. I - Writes

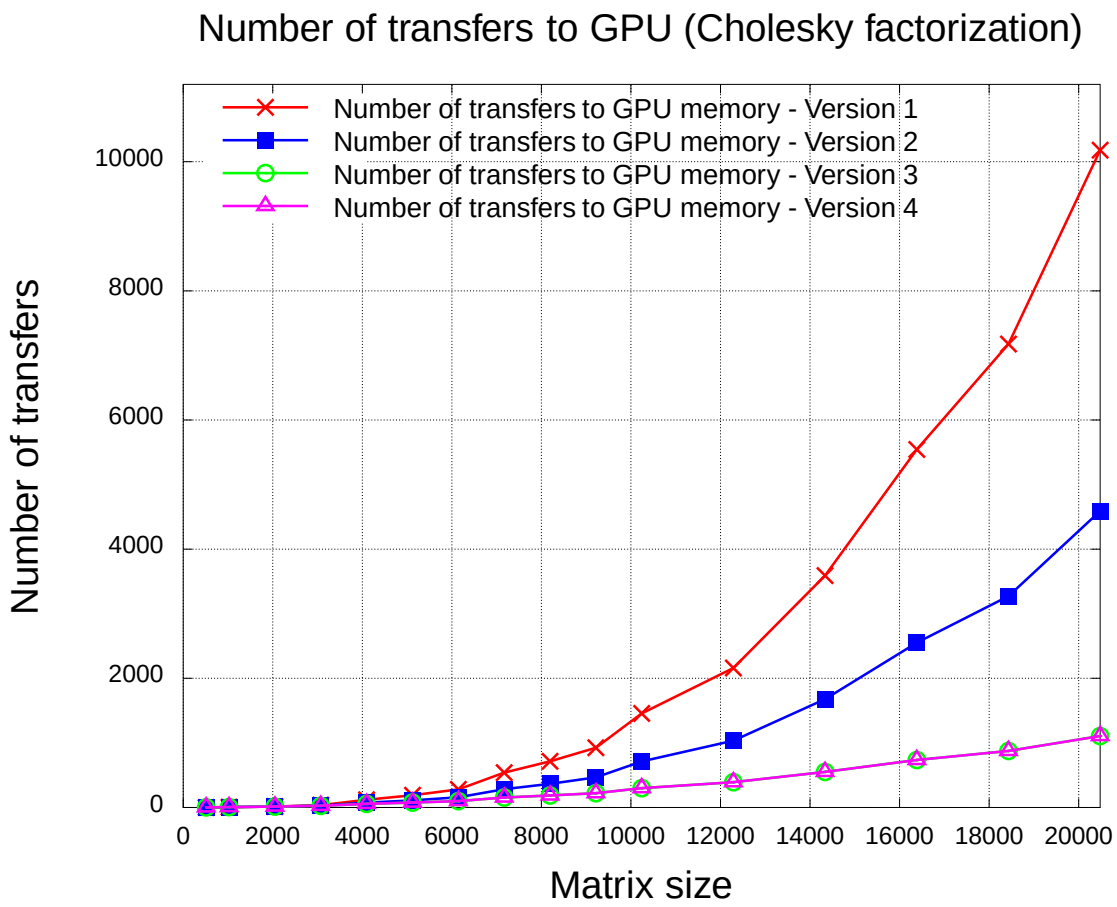


Figure: Number of data writes for the Cholesky factorization using 4 GPUs on TESLA. Block size is $b = 768$.

Reduction in the number of transfers. II - Reads

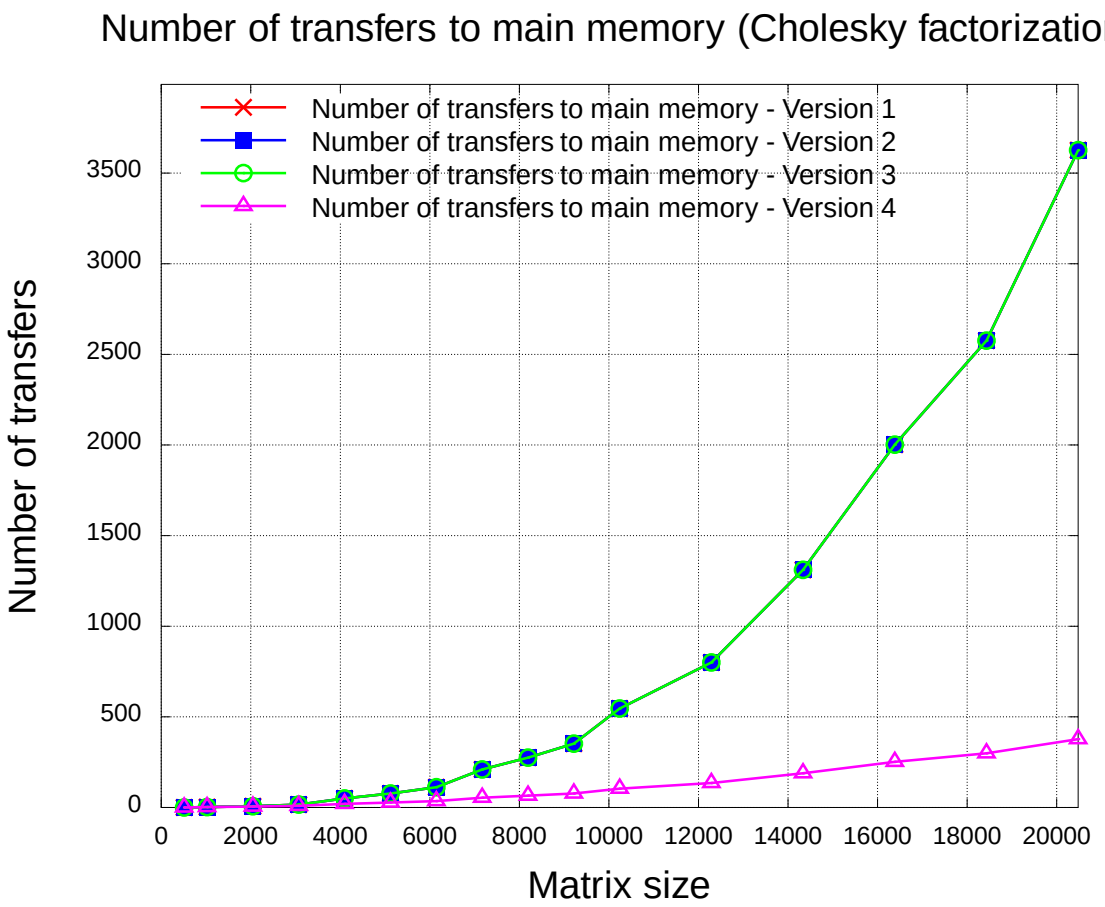


Figure: Number of data reads for the Cholesky factorization using 4 GPUs on TESLA. Block size is $b = 768$.

Experimental results. Cholesky on 4 GPUs

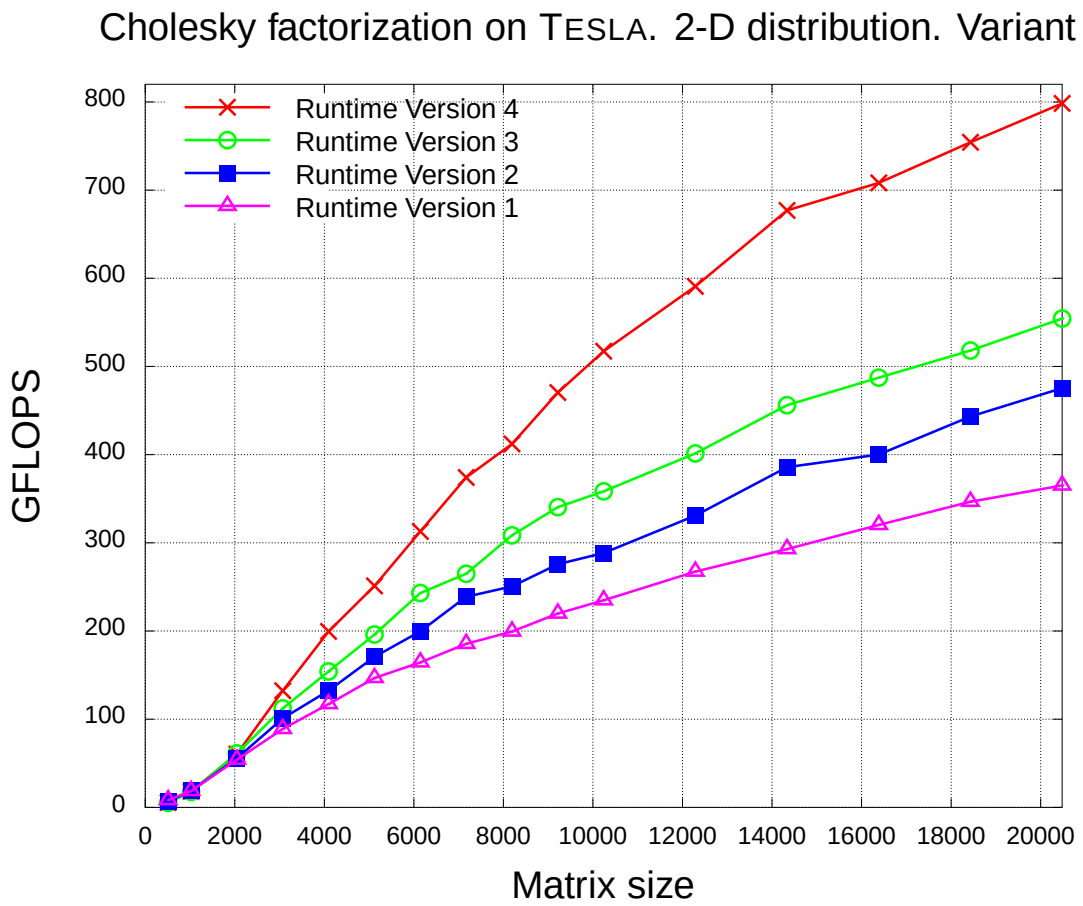


Figure: Performance of the Cholesky algorithmic variants on TESLA. 2-D distribution

Scheduling policies

2-D data affinity

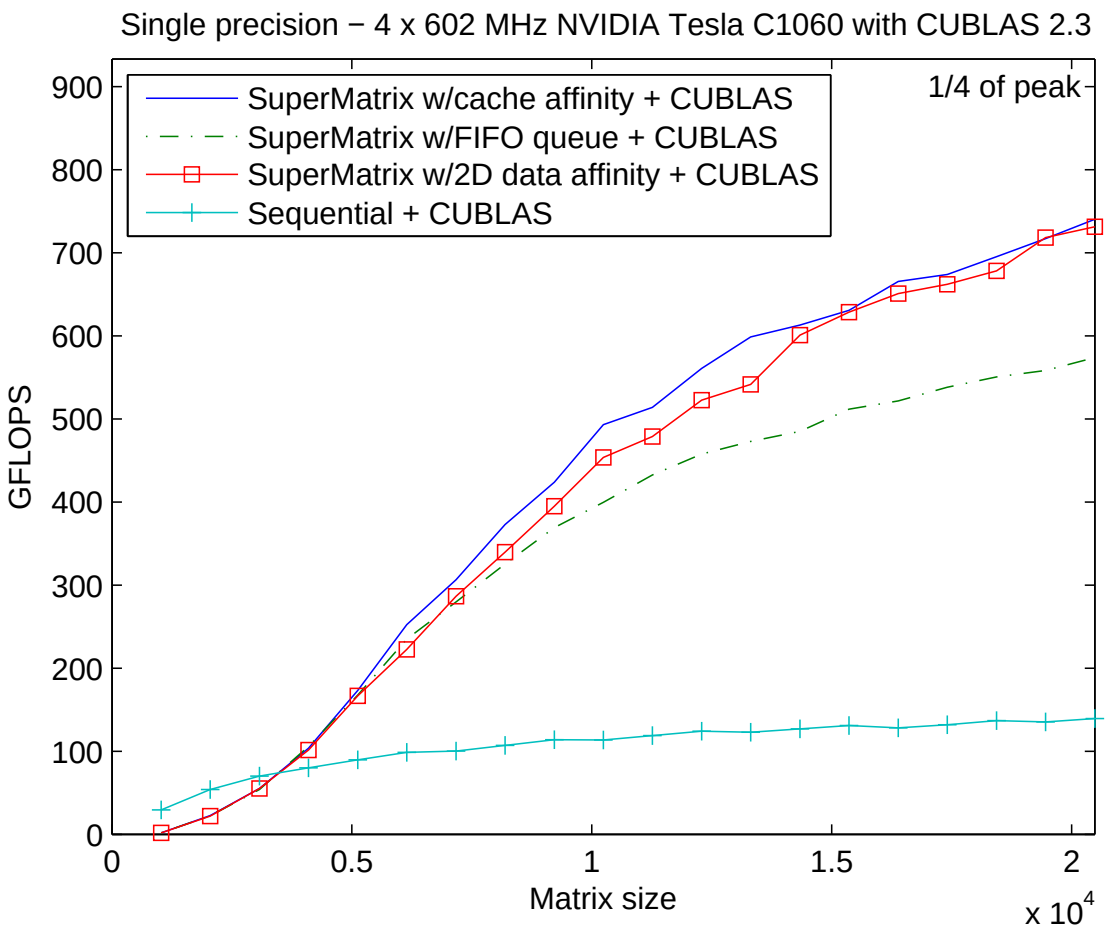
- Advantage: better data locality
- Disadvantage: worse load balancing

Solution: cache affinity

- Same approach as used in SuperMatrix
- Tasks are issued to the most *affine* GPU
- One common priority queue
- Gain data locality and load balancing

CHAN, E., IGUAL, F. D.
Runtime Data Flow Graph Scheduling of Matrix Computations with Multiple Hardware Accelerators.
FLAME Working Note #XX, 2010 .

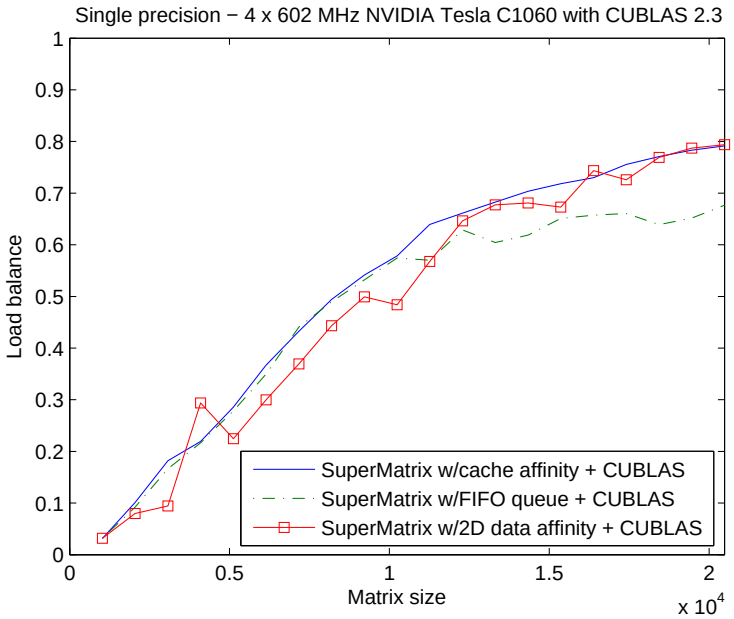
Performance



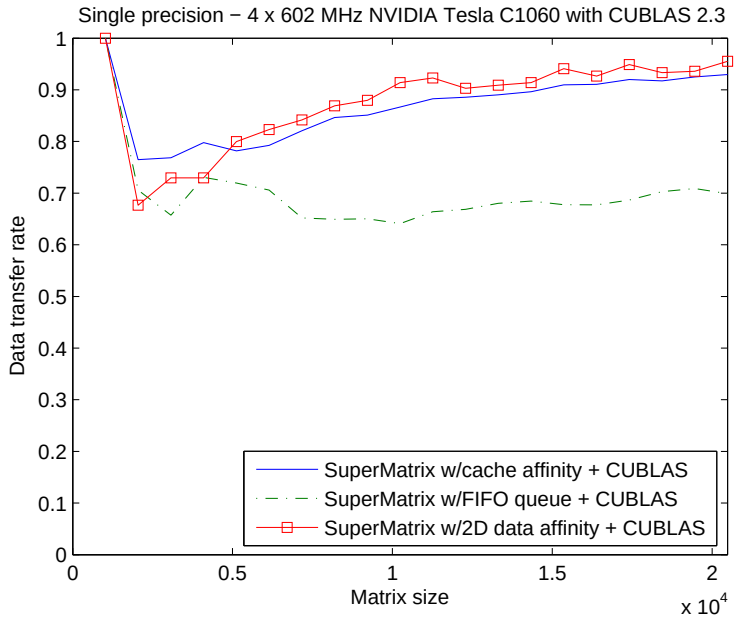
Reduction from a symmetric definite generalized eigenproblem to standard form:

$$A \leftarrow L^T A L$$

Load balance and data affinity



Load balance



Data affinity

Contents

Motivation

Future of GPU computing

Clusters with GPUs attached to each node.
Why? $\longrightarrow$ PCI-Express is the main bottleneck.

Future is today...

China's new Nebulae Supercomputer is No. 2 at Top500 (June 2010)
Powered with Tesla C2050 GPUs

- CUPLAPACK:
  - Retarget of PLAPACK to clusters with graphics processors.
- Why a PLAPACK-based approach?:
  - Modular design of PLAPACK. Independent layers.
  - FLAME-like methodology. Object-based approach.

Using PLAPACK as a library. Cholesky code

From code to accelerated code

```
1   int PLA_Chol( int b, PLA_Obj A )
2   {
3     PLA_Obj  ABR = NULL,
4              A11 = NULL,      A21 = NULL;
5     /* ... */
6
7     /* View ABR = A */
8     PLA_Obj_view_all( A, &ABR );
9
10    while ( TRUE ) {
11      /* Partition  ABR = / A11  ||   *  \
12       *                 | =====  ===== |
13       *                 \ A21  || ABR  /
14       * where A11 is b x b             */
15      PLA_Obj_split_4( ABR, b, b,
16                       &A11, PLA_DUMMY,
17                       &A21, &ABR );
18
19      /* A11 := L11 = Cholesky Factor( A11 ) */
20      PLA_Local_chol( PLA_LOWER_TRIANGULAR, A11 );
21
22      /* Update A21 := L21 = A21 * inv( L11' ) */
23      PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOWER_TRIANGULAR,
24                PLA_TRANSPOSE,  PLA_NONUNIT_DIAG,
25                one, A11,
26                     A21 );
27
28      /* Update A22 := A22 – L21 * L21' */
29      PLA_Syrk( PLA_LOWER_TRIANGULAR, PLA_NO_TRANS,
30                minus_one, A21,
31                one,       ABR );
32    }
33    /* ... */
34  }
```

```
int CUPLA_Chol( int b, PLA_Obj A )
{
  PLA_Obj  ABR = NULL,
           A11 = NULL,      A21 = NULL;
  /* ... */

  /* View ABR = A */
  PLA_Obj_view_all( A, &ABR );

  while ( TRUE ) {
    /* Partition  ABR = / A11  ||   *  \
     *                 | =====  ===== |
     *                 \ A21  || ABR  /
     * where A11 is b x b             */
    PLA_Obj_split_4( ABR, b, b,
                     &A11, PLA_DUMMY,
                     &A21, &ABR );

    /* A11 := L11 = Cholesky Factor( A11 ) */
    CUPLA_Local_chol( PLA_LOWER_TRIANGULAR, A11 );

    /* Update A21 := L21 = A21 * inv( L11' ) */
    CUPLA_Trsm( PLA_SIDE_RIGHT, PLA_LOWER_TRIANGULAR,
                PLA_TRANSPOSE,  PLA_NONUNIT_DIAG,
                one, A11,
                     A21 );

    /* Update A22 := A22 – L21 * L21' */
    CUPLA_Syrk( PLA_LOWER_TRIANGULAR, PLA_NO_TRANS,
                minus_one, A21,
                one,       ABR );
  }
  /* ... */
}
```

Using PLAPACK as a library. Cholesky code

From code to accelerated code

```
1   int PLA_Chol( int b, PLA_Obj A )
2   {
3     PLA_Obj  ABR = NULL,
4              A11 = NULL,      A21 = NULL;
5     /* ... */
6
7     /* View ABR = A */
8     PLA_Obj_view_all( A, &ABR );
9
10    while ( TRUE ) {
11      /* Partition  ABR = / A11  ||   *  \
12       *                 | =====  ===== |
13       *                 \ A21  || ABR  /
14       * where A11 is b x b              */
15      PLA_Obj_split_4( ABR, b, b,
16                       &A11, PLA_DUMMY,
17                       &A21, &ABR );
18
19      /* A11 := L11 = Cholesky Factor( A11 ) */
20      PLA_Local_chol( PLA_LOWER_TRIANGULAR, A11 );
21
22      /* Update A21 := L21 = A21 * inv( L11' ) */
23      PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOWER_TRIANGULAR,
24                PLA_TRANSPOSE,  PLA_NONUNIT_DIAG,
25                one, A11,
26                     A21 );
27
28      /* Update A22 := A22 – L21 * L21' */
29      PLA_Syrk( PLA_LOWER_TRIANGULAR, PLA_NO_TRANS,
30                minus_one, A21,
31                one,       ABR );
32    }
33    /* ... */
34  }
```

```
int CUPLA_Chol( int b, PLA_Obj A )
{
  PLA_Obj  ABR = NULL,
           A11 = NULL,      A21 = NULL;
  /* ... */

  /* View ABR = A */
  PLA_Obj_view_all( A, &ABR );

  while ( TRUE ) {
    /* Partition  ABR = / A11  ||   *  \
     *                 | =====  ===== |
     *                 \ A21  || ABR  /
     * where A11 is b x b              */
    PLA_Obj_split_4( ABR, b, b,
                     &A11, PLA_DUMMY,
                     &A21, &ABR );

    /* A11 := L11 = Cholesky Factor( A11 ) */
    CUPLA_Local_chol( PLA_LOWER_TRIANGULAR, A11 );

    /* Update A21 := L21 = A21 * inv( L11' ) */
    CUPLA_Trsm( PLA_SIDE_RIGHT, PLA_LOWER_TRIANGULAR,
                PLA_TRANSPOSE,  PLA_NONUNIT_DIAG,
                one, A11,
                     A21 );

    /* Update A22 := A22 – L21 * L21' */
    CUPLA_Syrk( PLA_LOWER_TRIANGULAR, PLA_NO_TRANS,
                minus_one, A21,
                one,       ABR );
  }
  /* ... */
}
```

GPU acceleration of PLAPACK

How to accelerate PLAPACK?

- *Naive implementation:*
  - Objects created in RAM.
  - GPU $\rightleftarrows$ CPU transfers bound to calculations.
  - Intercept calls to local BLAS and transfer data to/from GPU.
  - Straightforward implementation.
- *Tuned implementation:*
  - Objects created in GPU.
  - GPU $\rightleftarrows$ CPU transfers bound to communications.
  - Only transfer to RAM when necessary (communication).
  - Creation of objects involves allocation on GPUs.
  - Modification of communication layer in PLAPACK.

PLAPACK infrastructure

PLAPACK layered design

<table>
<tr><td colspan="5">Naive User Application</td><td>application layer</td></tr>
<tr><td rowspan="2">Application Program Interface
(PLA_API)</td><td colspan="4">Higher Level Global LA Routines
PLA_Global BLAS</td><td>library layer</td></tr>
<tr><td colspan="2">PLA_Copy/Reduce</td><td>LA object manipulation</td><td>PLA_Local BLAS</td><td>PLAPACK abstraction layer</td></tr>
<tr><td>MMPI</td><td>PLA/MPI interface</td><td>PLA_malloc</td><td>PBMD Templates</td><td>PLA/ BLAS interface</td><td>machine/distribution independent layer</td></tr>
<tr><td colspan="2">Message-Passing Interface</td><td>malloc</td><td>cartesian distributions</td><td>vendor BLAS</td><td>machine/distribution specific layer</td></tr>
</table>

Necessary changes to use GPU

1. PLA_Local_BLAS: NVIDIA CUBLAS.
2. LA object manipulation: management of objects in GPU memory.
3. Communication module: transfers to RAM prior to communications.

LA object manipulation. Cholesky driver using GPU

```
1   int main( void ){
2     /* ... */
3     // Object creation on GPU
4     CUPLA_Matrix_create(MPI_FLOAT, size, size, templ,
5                         PLA_ALIGN_FIRST, PLA_ALIGN_FIRST,
6                         &A_GPU);
7
8     // Object initialization on GPU
9     CUPLA_Obj_set_to_SPD_random( A_GPU );
10
11    /* ... */
12
13    // Cholesky factorization on GPU
14    CUPLA_Chol( nb_alg, A_GPU );
15    /* ... */
16
17    // Object destruction on GPU
18    CUPLA_Obj_free( &A_GPU );
19  }
```

- `CUPLA_Matrix_create` creates a L. A. object on GPU.
- `CUPLA_Obj_free` destroys an object on GPU.
- `CUPLA_Chol` operates with objects created on GPU.
- ...

Communication layer

Communication is restricted to routines `PLA_Copy` and `PLA_Reduce`:

int PLA_Copy (PLA_Obj obj_from, PLA_Obj obj_to)

Purpose: Copy contents between linear algebra objects.

IN	obj_from	Object to be copied
IN/OUT	obj_to	Object into which to copy

int PLA_Reduce (PLA_Obj obj_from, MPI_OP op, PLA_Obj obj_to)

Purpose: Reduce using `op` the contents in the duplicated object given by `obj_from` and overwrite `obj_to` with the result.

IN	obj_from	Object to be reduced
IN	op	Reduce operator to be used
IN/OUT	obj_to	Object into which to reduce result

Modifications to PLA_Copy (I)

```
1   /* PLAPACK PLA_Copy between column aligned matrices */
2
3   while ( TRUE ){
4     PLA_Obj_split_size( from_cur, PLA_SIDE_TOP, &size_from, &owner_from );
5     PLA_Obj_split_size( to_cur,   PLA_SIDE_TOP, &size_to,   &owner_to);
6
7     if ( 0 == ( size = min( size_from, size_to ) ) ) break;
8
9     PLA_Obj_horz_split_2( from_cur, size, &from_1, &from_cur );
10    PLA_Obj_horz_split_2( to_cur,   size, &to_1,   &to_cur );
11
12    if ( myrow == owner_from && owner_from == owner_to ){
13      PLA_Local_copy( from_1, to_1 );
14    }
15    else{
16      if ( myrow == owner_from ) {
17        PLA_Obj_get_local_contents( from_1, PLA_NO_TRANS, &dummy, &dummy,
18                                    buffer_temp, size, 1 );
19
20        MPI_Send( BF( buffer_temp ), size * local_width, datatype,
21                  owner_to, 0, comm_col );
22      }
23      if ( myrow == owner_to ) {
24        MPI_Recv( BF( buffer_temp ), size * local_width, datatype,
25                  owner_from, MPI_ANY_TAG, comm_col, &status );
26
27        PLA_Obj_set_local_contents( PLA_NO_TRANS, size, local_width,
28                                    buffer_temp, size, 1, to_1 );
29      }
30    }
31  }
32  PLA_free( buffer_temp );
```

Modifications to PLA_Copy (II)

```
1  /* PLAPACK PLA_Obj_get_local_contents */
2
3  int PLA_Obj_get_local_contents    (
4          PLA_Obj    obj,
5          int        trans,
6          int        *rows_in_buf,
7          int        *cols_in_buf,
8          void       *buf,
9          int        ldim_buf,
10         int        stride_buf)
11
12   /****************************/
13
14   for ( j=0; j<n; j++ ){
15     tmp_local = buf_local + j*ldim_buf*typesize;
16     tmp_obj   = buf_obj   + j*ldim*typesize;
17     memcpy( tmp_local, tmp_obj, m*typesize );
18   }
19
20   /****************************/
```

```
/* CUPLAPACK PLA_Obj_get_local_contents */

int CUPLA_Obj_get_local_contents    (
        PLA_Obj    obj,
        int        trans,
        int        *rows_in_buf,
        int        *cols_in_buf,
        void       *buf,
        int        ldim_buf,
        int        stride_buf)

  /****************************/

  cublasGetMatrix( m, n, typesize,
                   buf_obj,   ldim,
                   buf_local, ldim_buf );

  /****************************/
```

Necessary GPU $\rightleftarrows$ CPU transfers

Data is transferred to RAM prior to a communication.
Data is transferred to GPU after each communication.

The LONGHORN visualization cluster

	Per Node	Per System (256 nodes)
Number of cores	8	2,048
CPU	Intel Xeon Nehalem @ 2.53 GHz	
Available memory	48 Gbytes	13.5 TBytes
Interconnection network	QDR Infiniband	
Graphics system	128 NVIDIA Quadro Plex S4s	
GPU	2 x NVIDIA Quadro FX5800	512 x NVIDIA Quadro FX5800
Interconnection bus	PCIExpress 2.0 (8x)	
Available video memory	8 Gbytes DDR3	2 TBytes DDR3
Peak performance (SP)	161.6 GFLOPS	41.40 TFLOPS
Peak performance (DP)	80.8 GFLOPS	20.70 TFLOPS

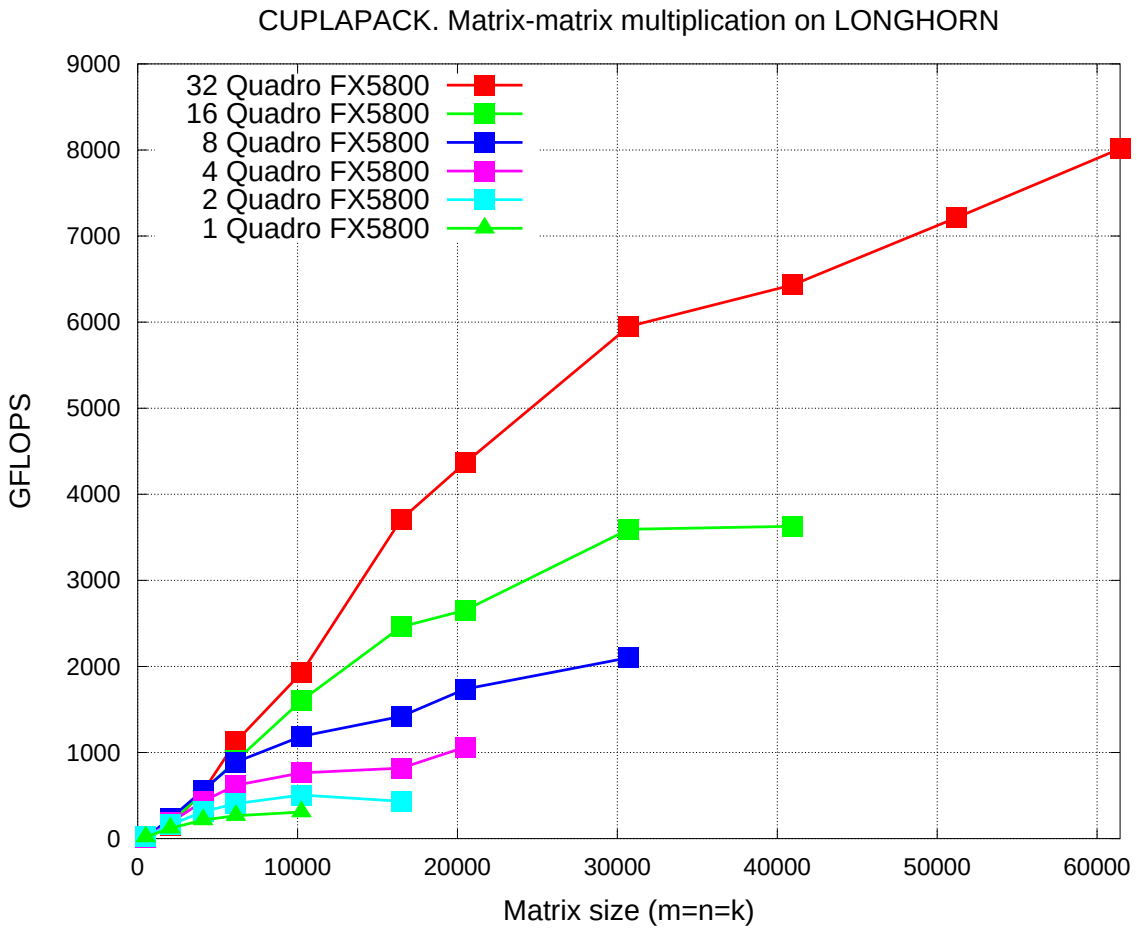
PLAPACK: version R32.

BLAS: NVIDIA CUBLAS 2.2, MKL 10.1. Single precision.

MPI: MVAPICH 1.4.

Experiments on 16 nodes of LONGHORN (16 or 32 GPUs).
Optimal distribution and algorithmic block sizes.

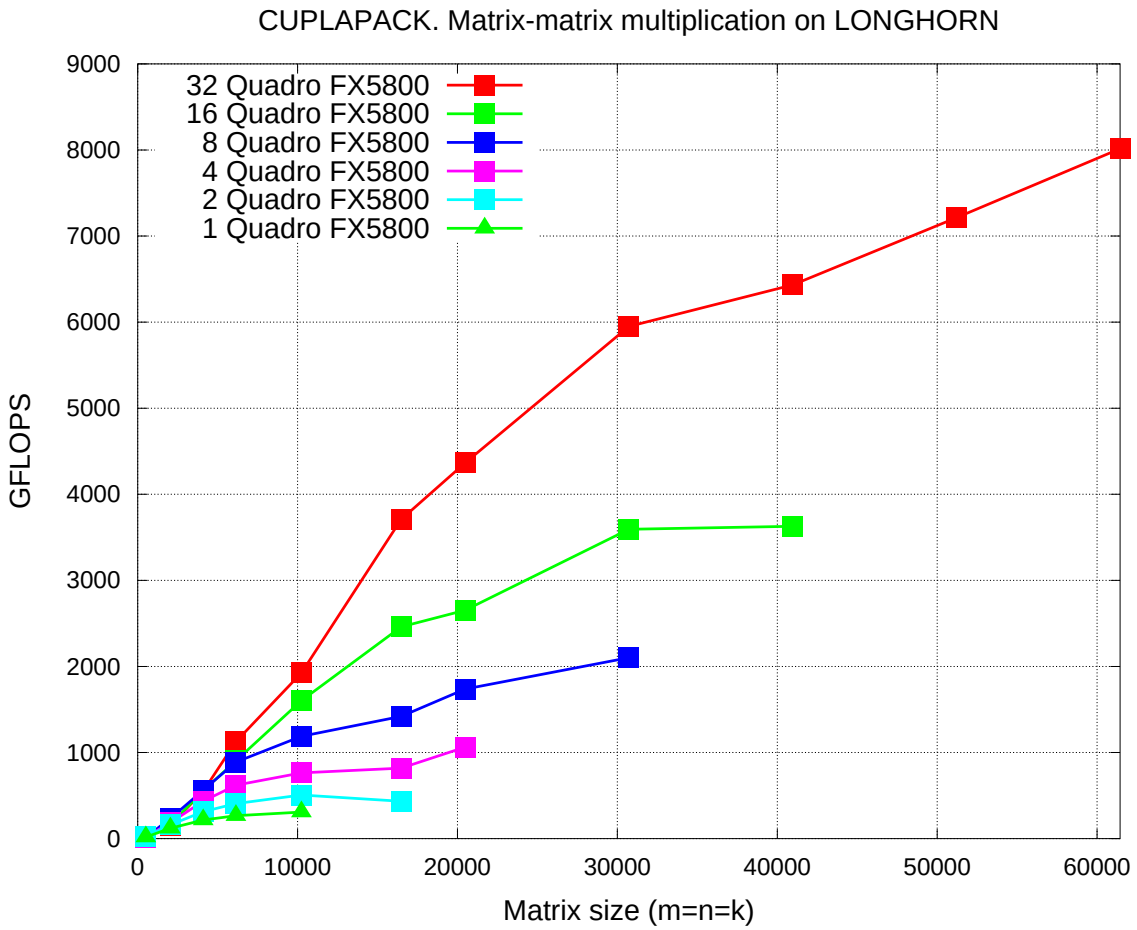
GEMM: performance



CUPLAPACK GEMM on 16 nodes.

Peak performance: 8 TFLOPS for GEMM using 32 GPUs.

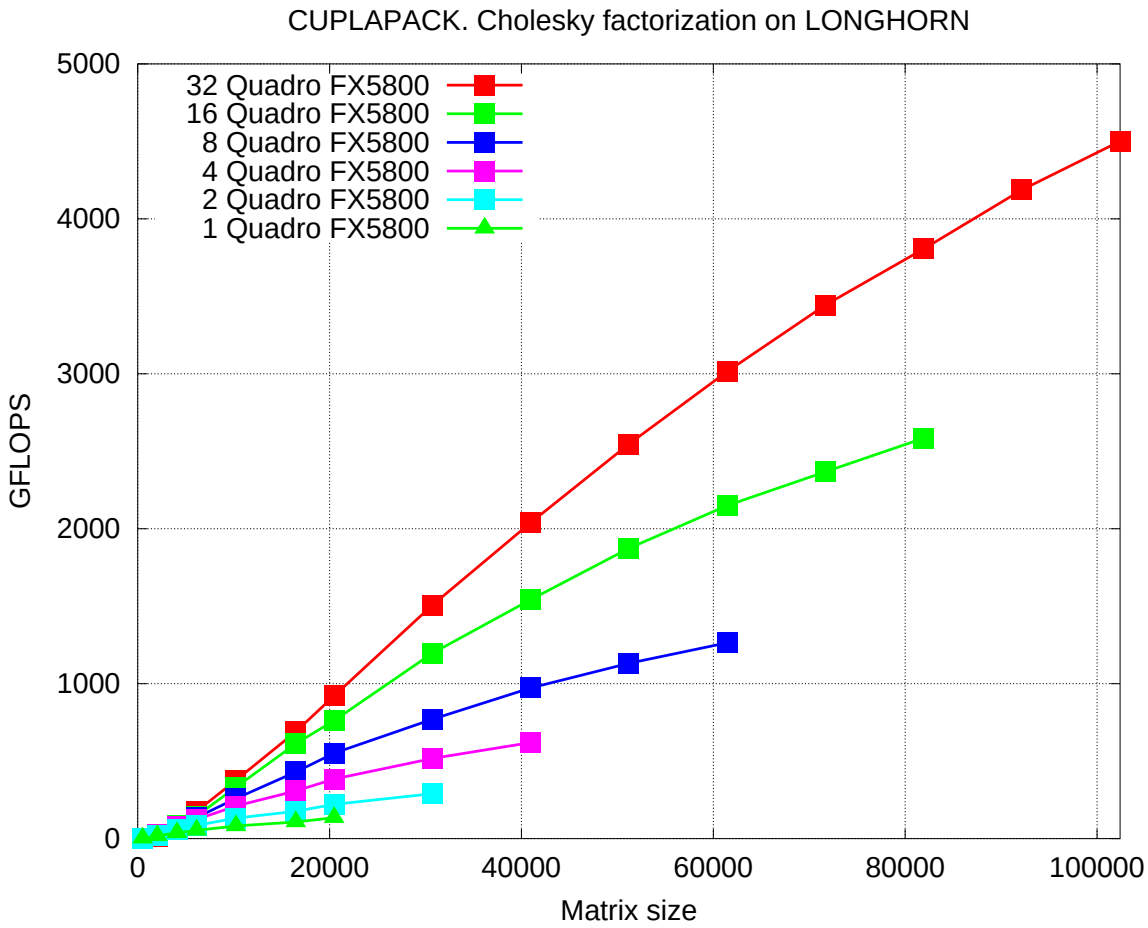
GEMM: performance



CUPLAPACK GEMM on 16 nodes.

Peak performance: 8 TFLOPS for GEMM using 32 GPUs.

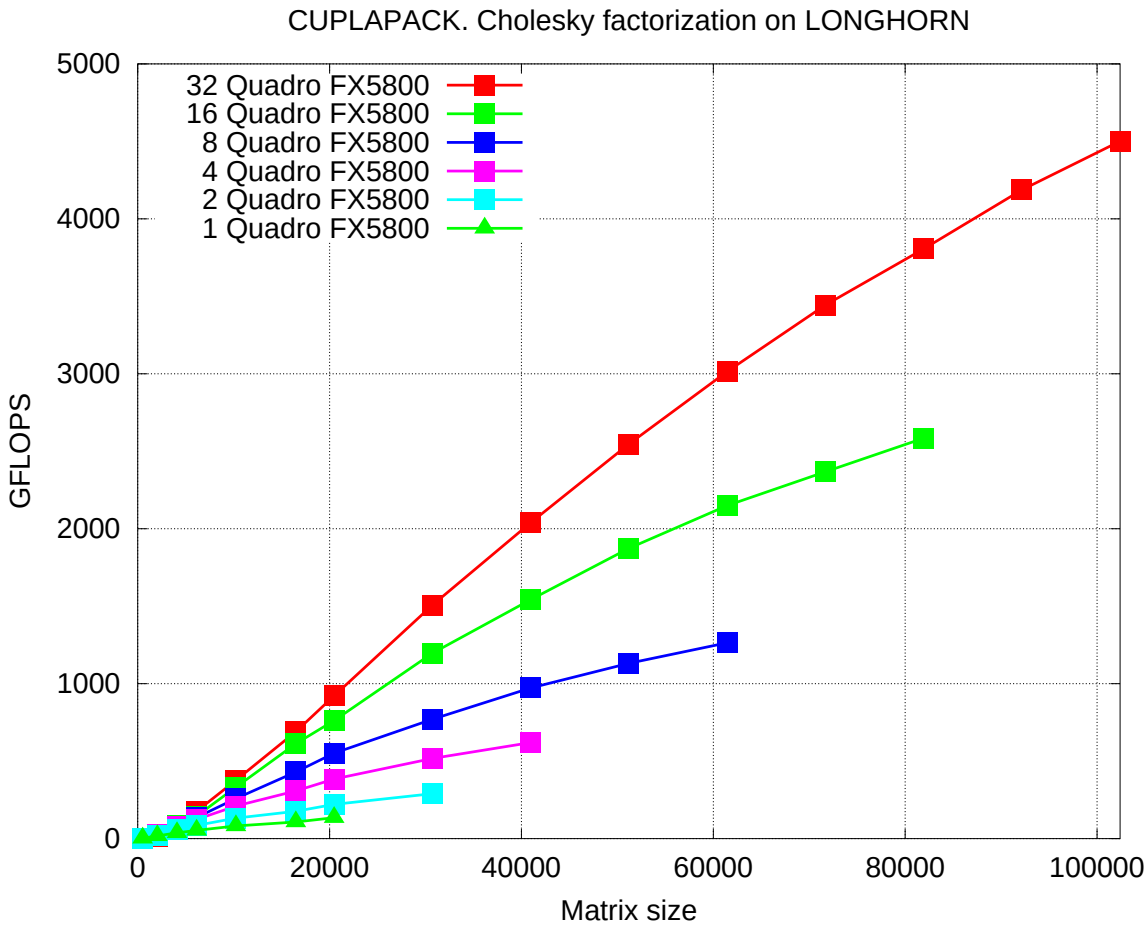
Cholesky factorization: performance



CUPLAPACK Cholesky on 16 nodes.

Peak performance: 4.5 TFLOPS for Cholesky using 32 GPUs.

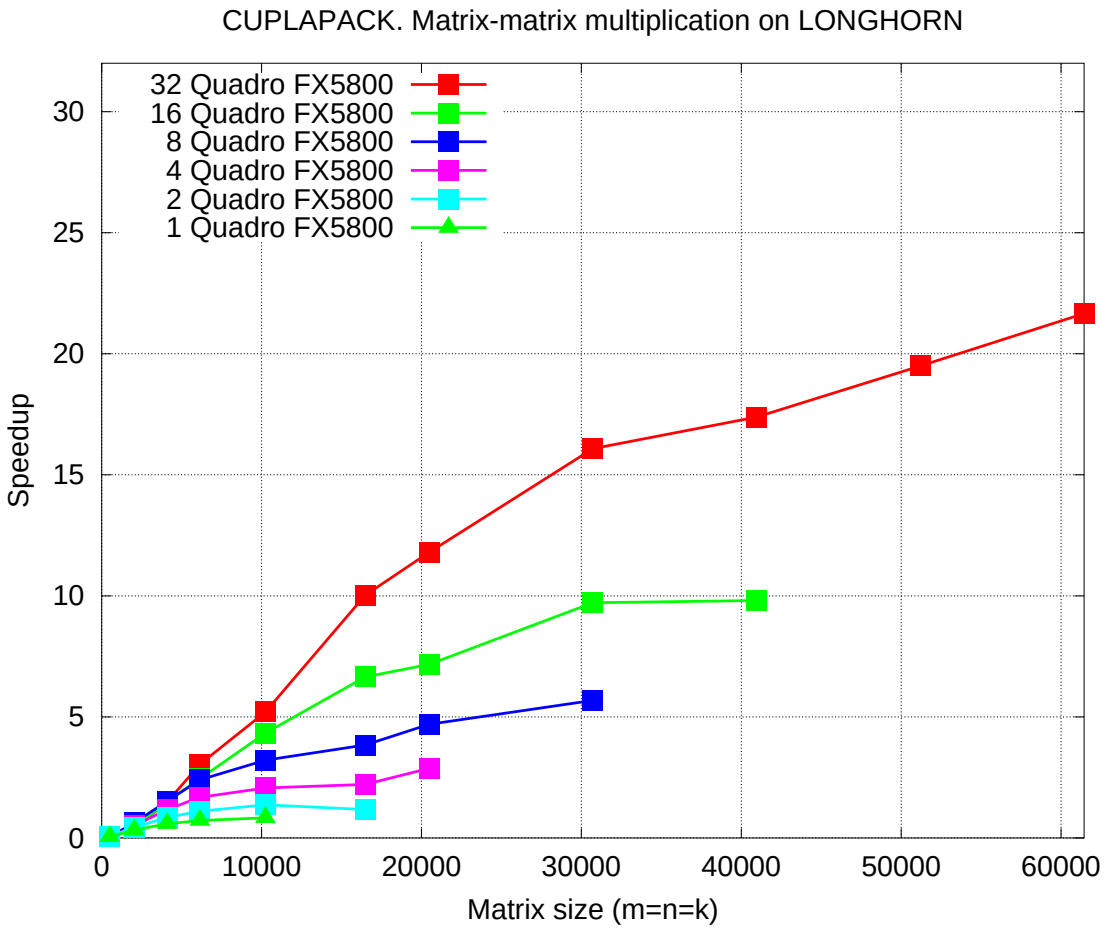
Cholesky factorization: performance



CUPLAPACK Cholesky on 16 nodes.

Peak performance: 4.5 TFLOPS for Cholesky using 32 GPUs.

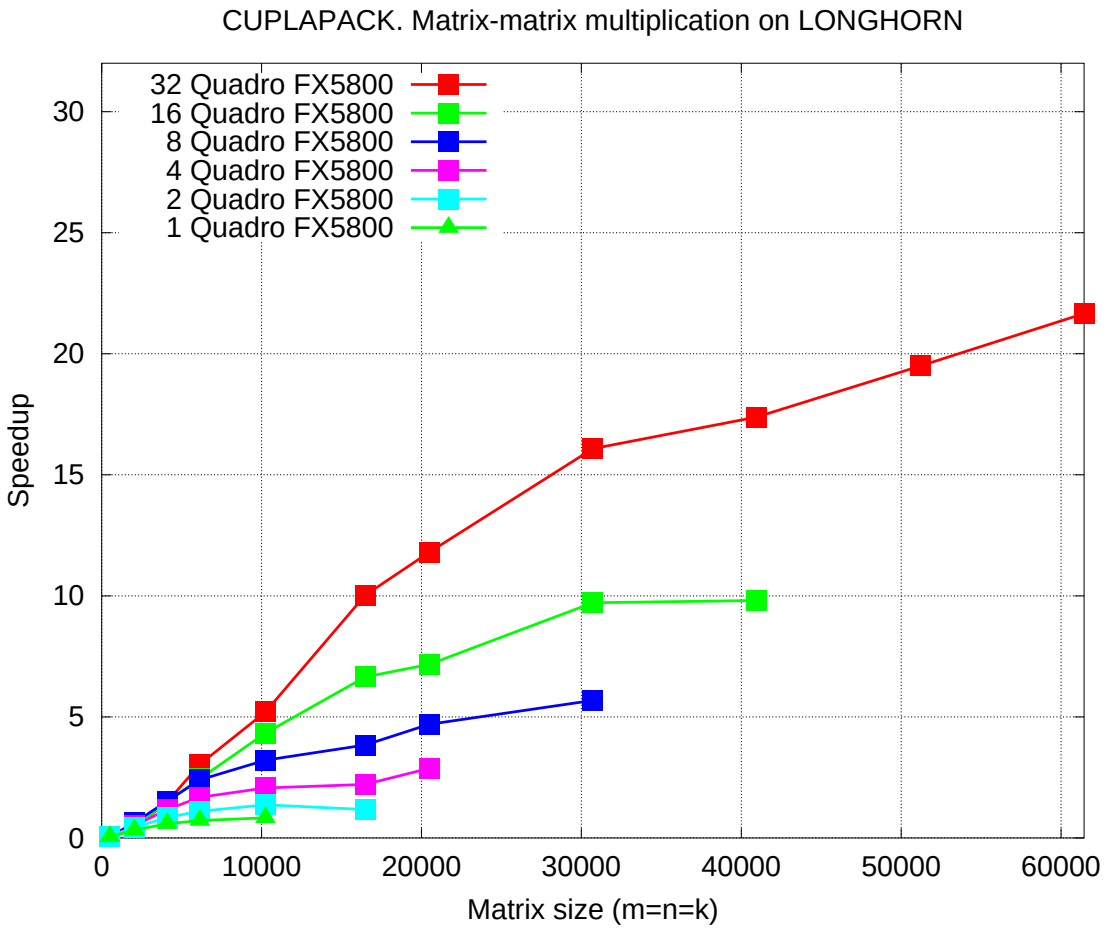
GEMM: scalability



Scalability of GEMM. Reference is CUBLAS SGEMM on one GPU.

No significant bottlenecks.

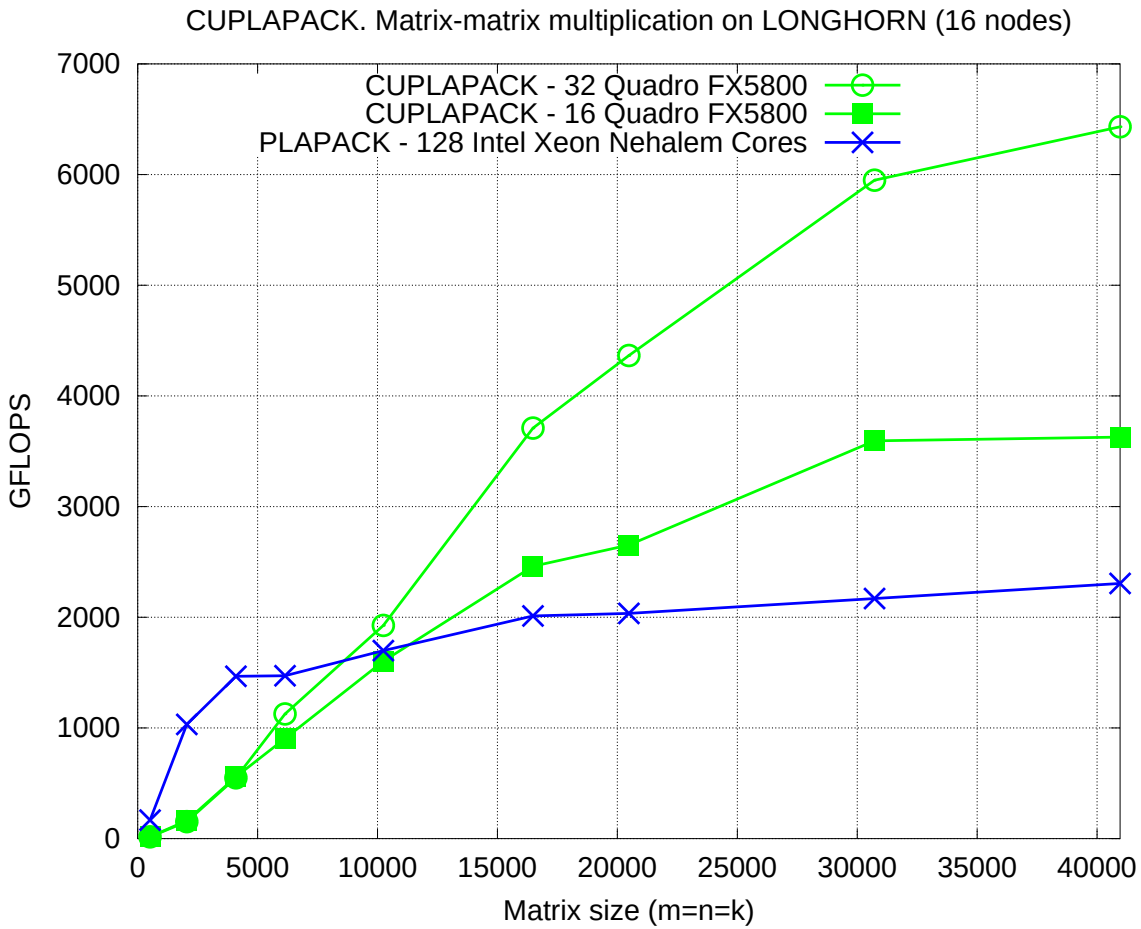
GEMM: scalability



Scalability of GEMM. Reference is CUBLAS SGEMM on one GPU.

No significant bottlenecks.

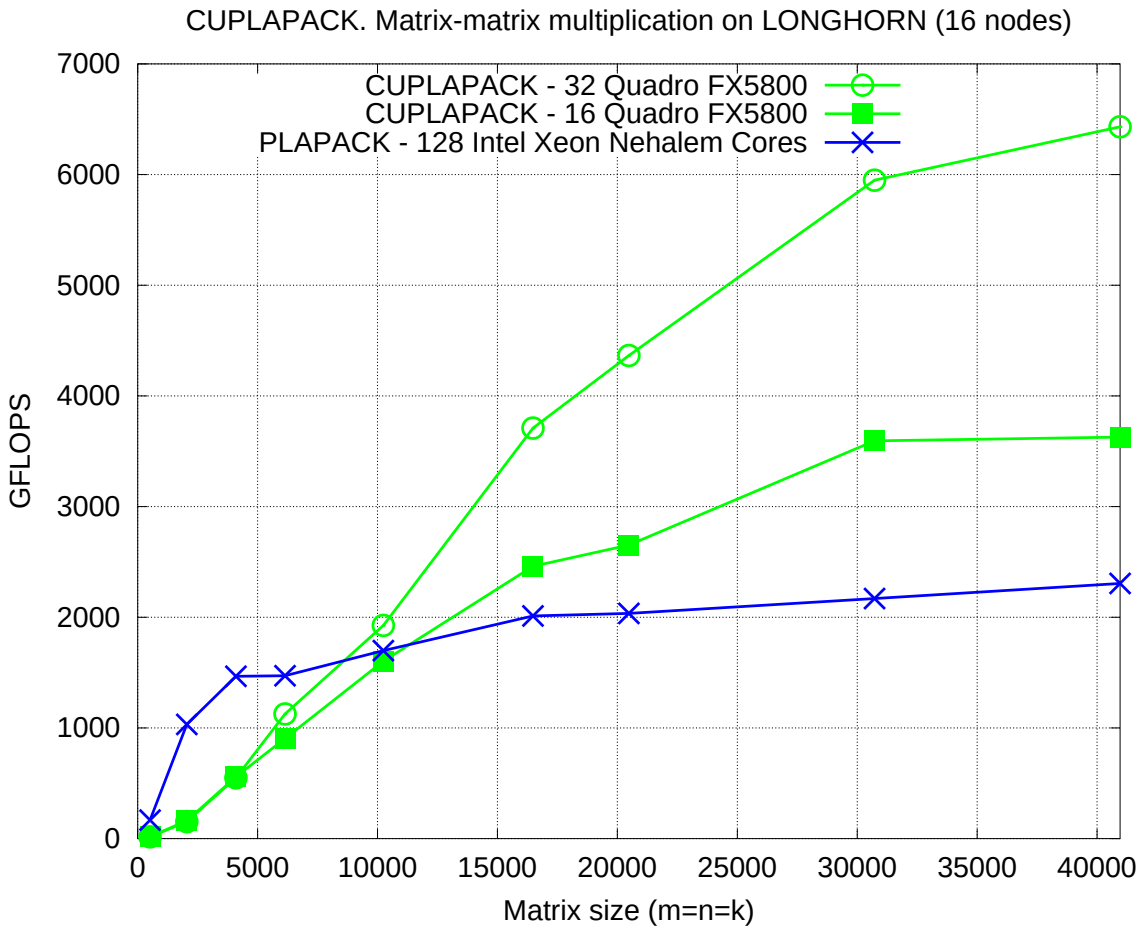
GEMM: PLAPACK vs. CUPLAPACK



GEMM on 16 nodes. PLAPACK vs CUPLAPACK.

2.7x using 32 GPUs. Improvement for bigger matrices.

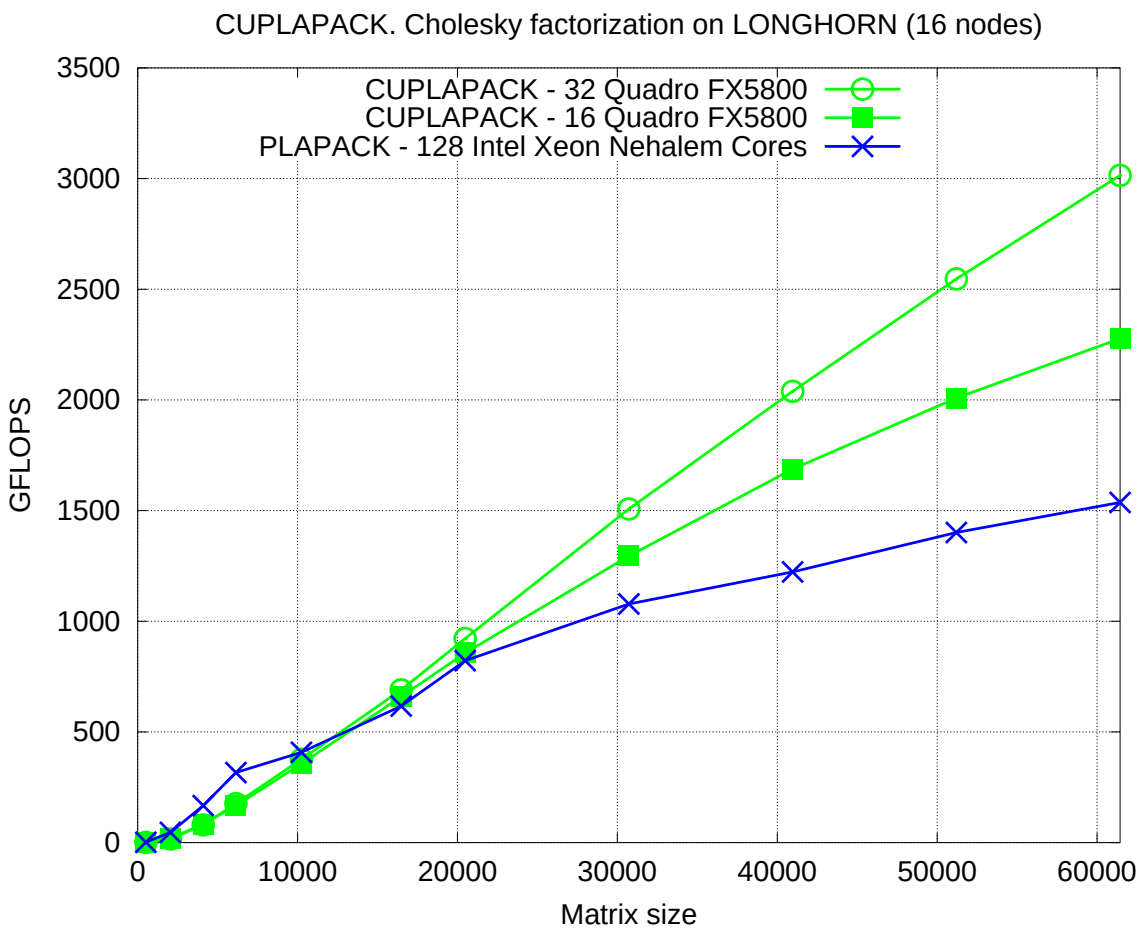
GEMM: PLAPACK vs. CUPLAPACK



GEMM on 16 nodes. PLAPACK vs CUPLAPACK.

2.7x using 32 GPUs. Improvement for bigger matrices.

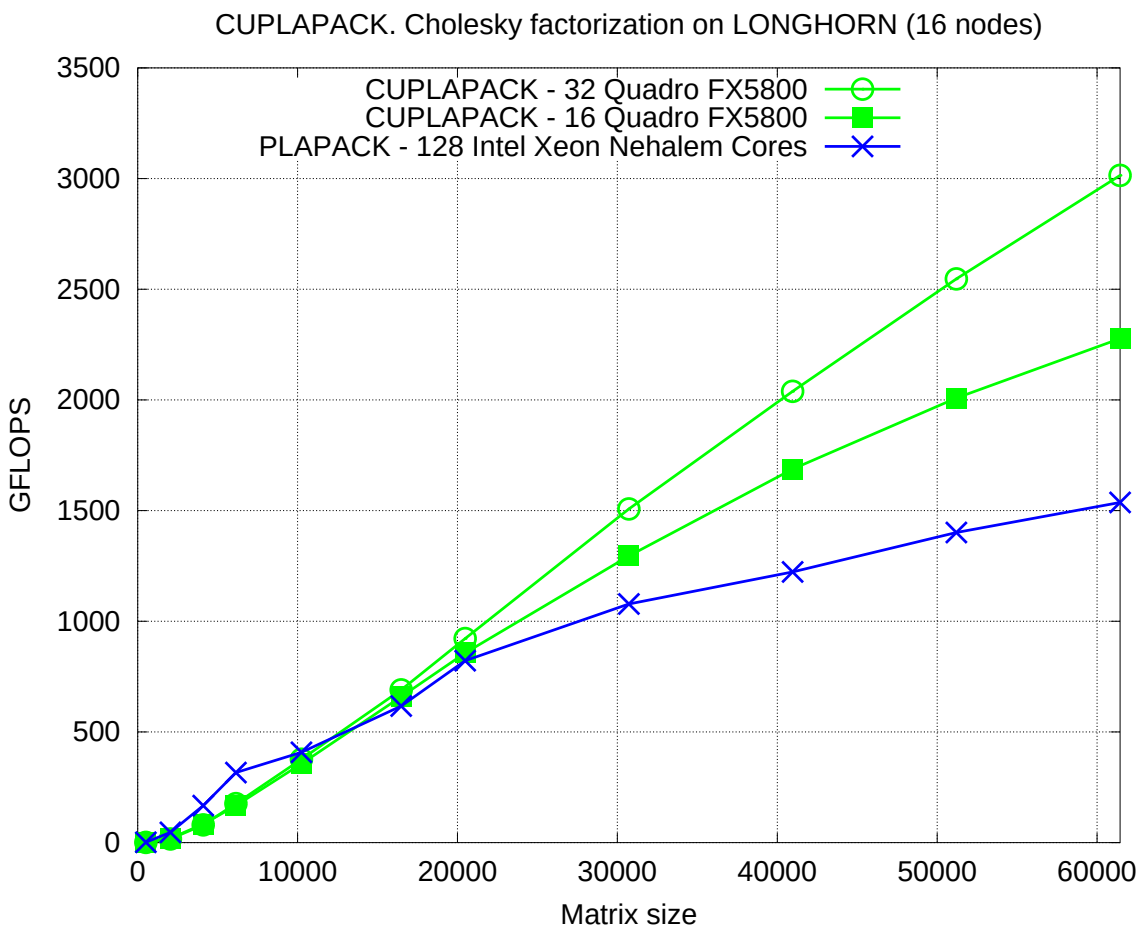
Cholesky factorization: PLAPACK vs. CUPLAPACK



Cholesky factorization on 16 nodes. PLAPACK vs CUPLAPACK.

2x using 32 GPUs. Improvement for bigger matrices.

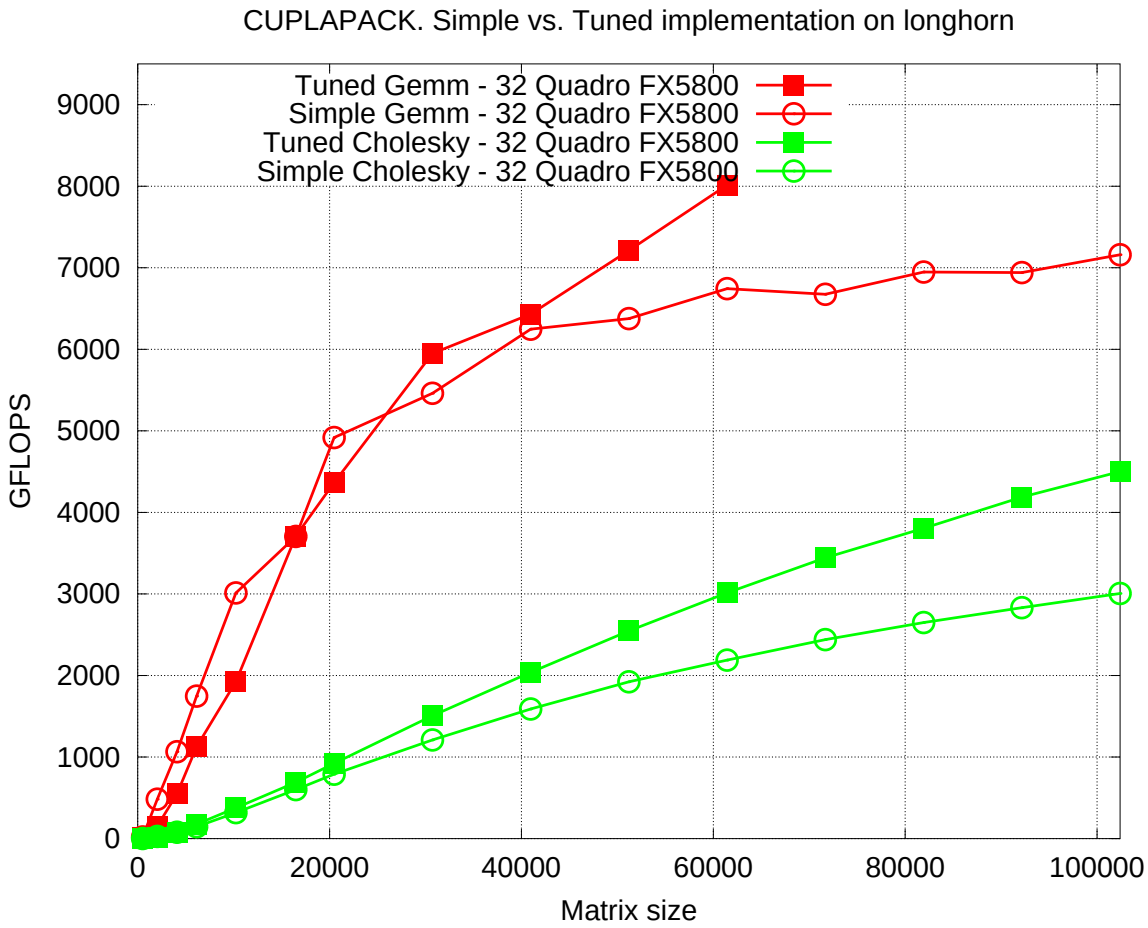
Cholesky factorization: PLAPACK vs. CUPLAPACK



Cholesky factorization on 16 nodes. PLAPACK vs CUPLAPACK.

2x using 32 GPUs. Improvement for bigger matrices.

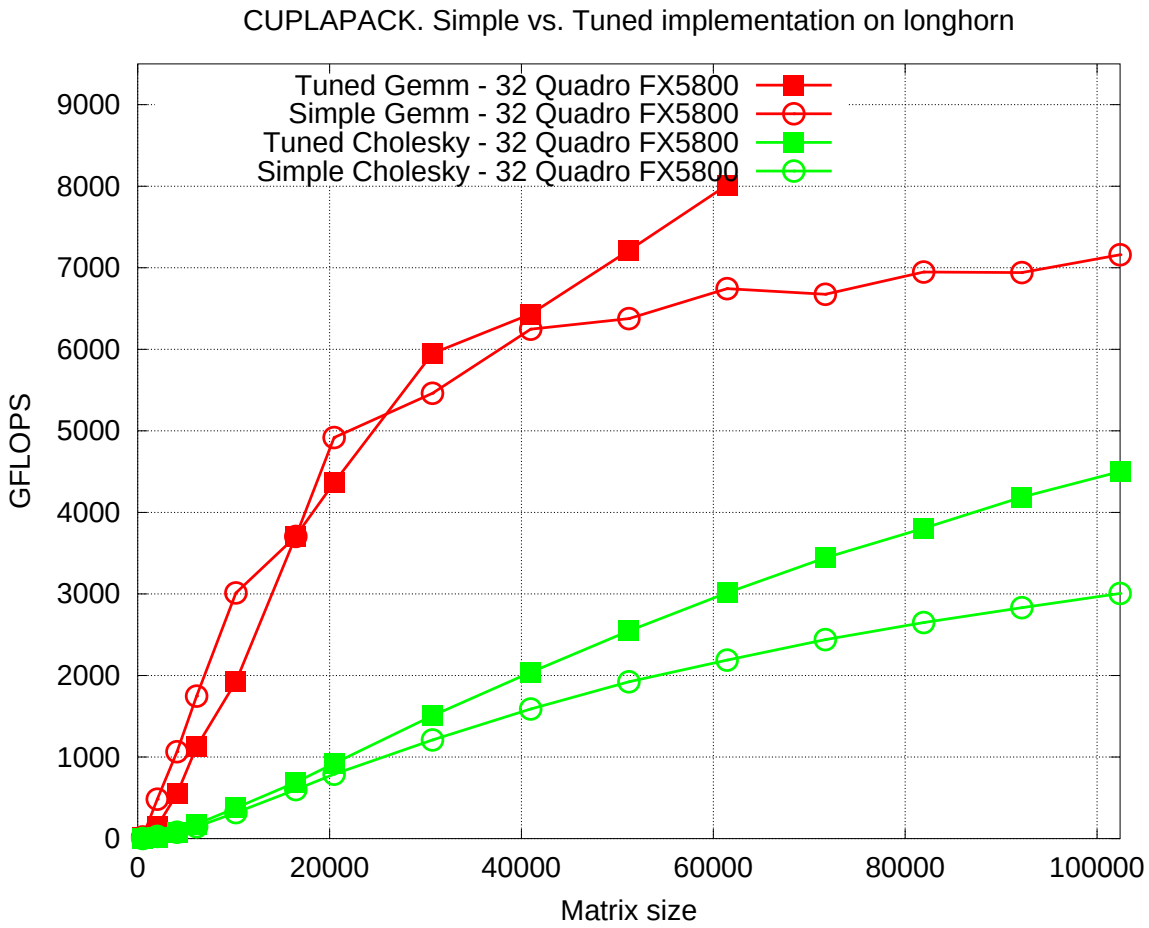
Naïve vs Tuned implementation



Comparison between *naive* and *tuned* implementations.

Improvements of the *tuned* for large matrices.

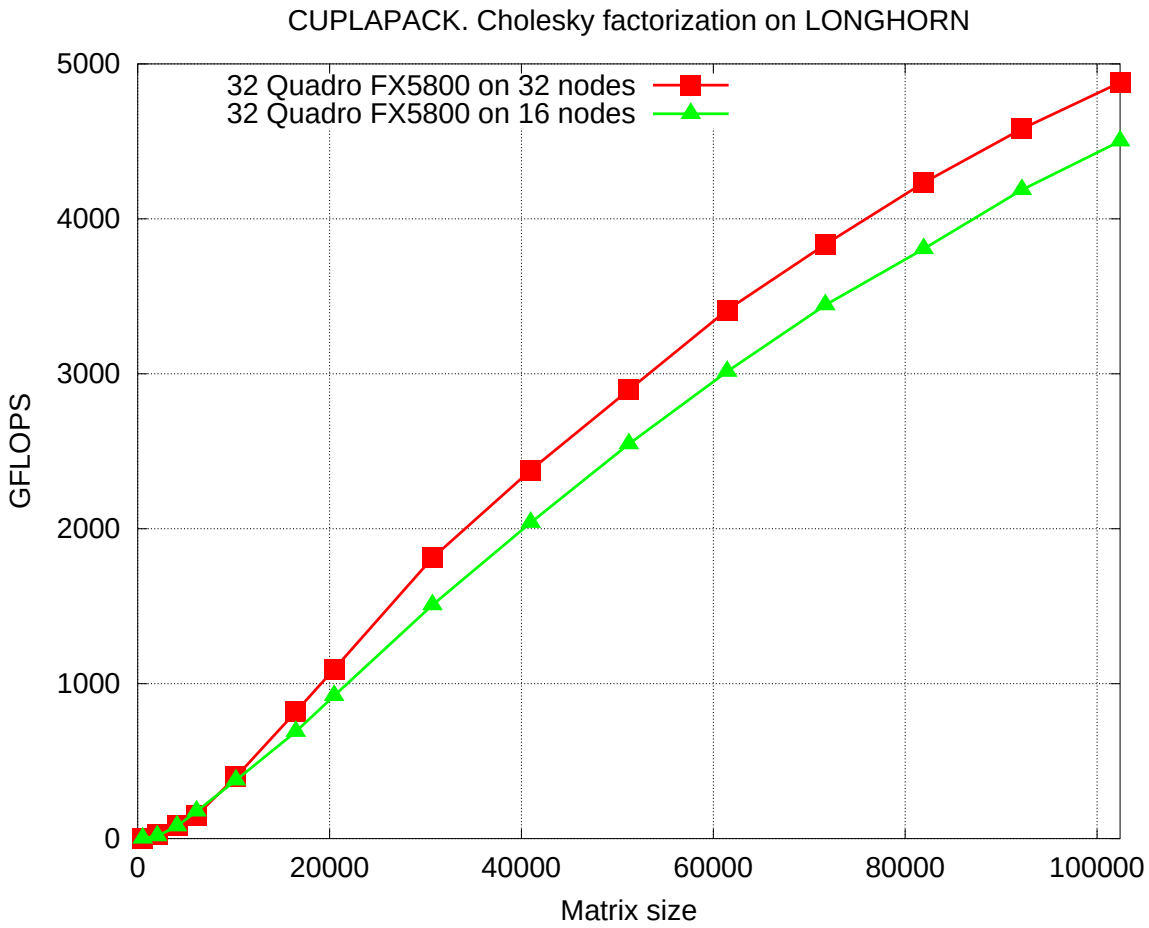
Naïve vs Tuned implementation



Comparison between *naive* and *tuned* implementations.

Improvements of the *tuned* for large matrices.

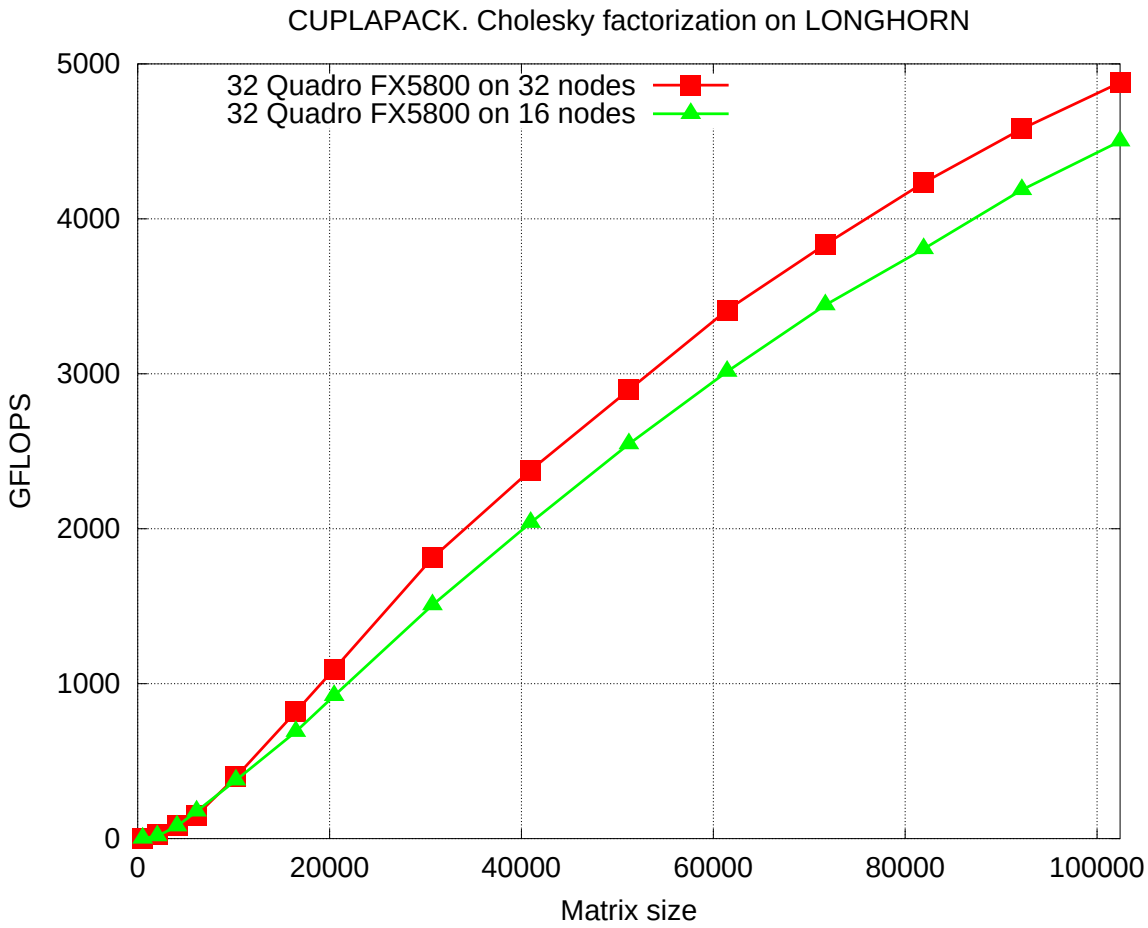
Multiple GPUs per node vs. One GPU per node



Cholesky factorization on 32 GPUs (**one** or **two** GPUs per node).

Penalty introduced due to the shared PCI-Express bus.

Multiple GPUs per node vs. One GPU per node



Cholesky factorization on 32 GPUs (**one** or **two** GPUs per node).

Penalty introduced due to the shared PCI-Express bus.

Contents

Conclusions and future work

Conclusions

- Ability of FLAME to adapt to exotic architectures
- Good performance results without loosing programmability

Future work

- Integrate and release as part of `libflame`...
- Use SuperMatrix + GPU for multi-GPU per node.
- Forget PLAPACK and port Elemental to clusters of GPUs.
- Mellanox support for direct GPU-GPU communication through InfiniBand in clusters of GPUs.

Other research lines

- Out-of-core + GPU support to deal with even larger problems.
- Energy-aware GPU computing.
- `rCUDA`: GPU virtualization.

Acknowledgements

The team. . .

- Maribel Castillo
- Manuel Fogué
- Francisco Igual
- Rafael Mayo
- Antonio Peña
- Enrique Quintana-Ortí
- Gregorio Quintana-Ortí
- . . . and Ernie Chan.

More information. . .

`http://www.hpca.uji.es`
`http://www3.uji.es/~figual`

Supported by. . .

GOBIERNO DE ESPAÑA
MINISTERIO DE CIENCIA E INNOVACIÓN
NVIDIA
Microsoft

We thank the `Texas Advanced Computing Center` for granting access to LONGHORN.

TACC TEXAS ADVANCED COMPUTING CENTER
Powering Discoveries That Change The World
THE UNIVERSITY OF TEXAS AT AUSTIN