# Use of GPUs in Dense Linear Algebra

Enrique S. Quintana-Ortí

Universitat Jaume I
Spain

April, 2008

## Joint work with:

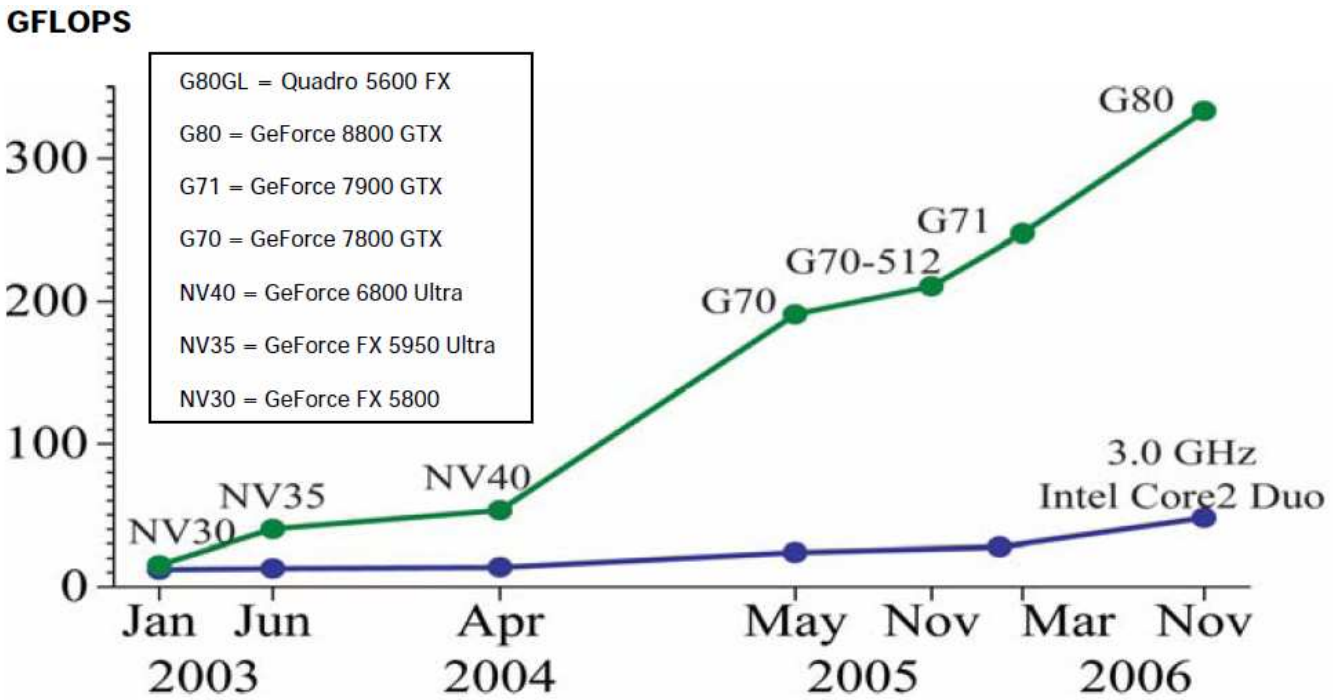
Sergio Barrachina
Maribel Castillo
Francisco D. Igual
Rafael Mayo
Gregorio Quintana-Ortí
Rafael Rubio

Universidad Jaime I (Spain)

Ernie Chan
Robert van de Geijn
Field G. Van Zee

The University of Texas at Austin

# Motivation




The power and versatility of modern GPU have transformed them into the first widely extended HPC platform

# Outline

# Introduction

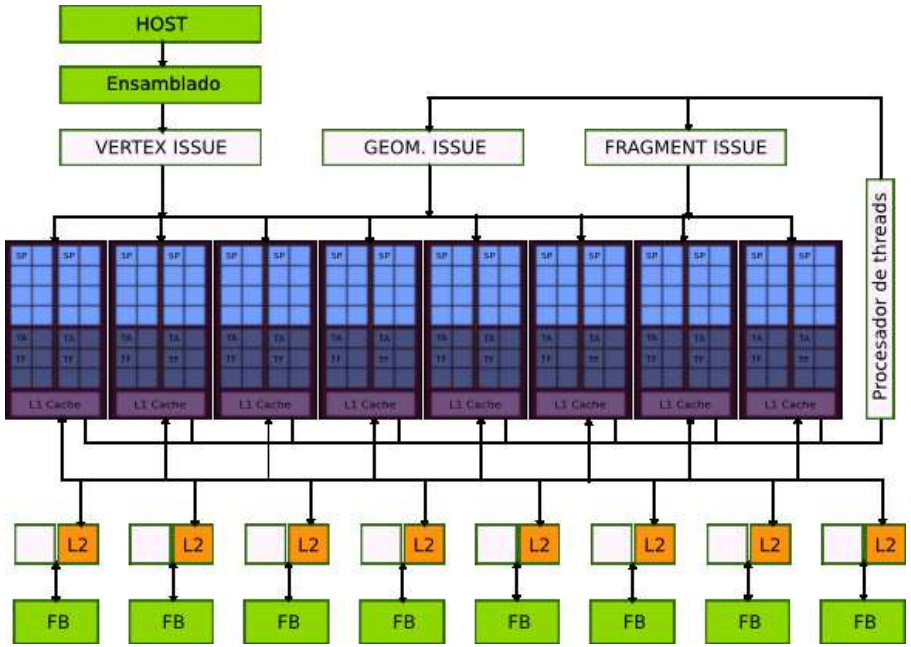
- Graphics processors (GPUs) have emerged as an interesting hardware accelerator for "general-purpose" computations (GPGPU):
  1. High performance
  2. Low cost

$\longrightarrow$

- Applied to physical simulations, real-time image processing, linear algebra, signal processing,...

# CUDA hardware

- A CUDA-enabled device is seen as a coprocessor to the CPU, capable of executing a very high number of threads in parallel
- Example: nVIDIA G80 can be seen as a set of SIMD Multiprocessors with On-Chip Shared Memory




- Up to 128 *Streaming Processors* (SP), grouped in clusters
- SP are SIMD processors
- Small and fast Shared Memory shared per SP cluster
- Local 32-bit registers per processor

# CUDA software

- The CUDA API provides a simple path for writing C programs for execution on the GPU consisting of
  - A minimal set of extensions to the C language
  - A runtime library of routines for controlling the transfers between video and main memory, execution configuration, the execution of device-specific functions, handling multiple GPUs from the host,...

## CUDA libraries

On top of CUDA, nVIDIA provides two optimized libraries: CUFFT and CUBLAS

# CUBLAS. Example

```
int main( void ){
  ...
  float* h_vector, * d_vector;

  h_vector = (float*)malloc(M*sizeof(float));

  ...
  cublasAlloc(M, sizeof(float),
              (void**) &d_vector);

  cublasSetVector(M, sizeof(float), h_vector,
                  d_vector, 1);
  cublasSscal(M, ALPHA, d_vector, 1);
  cublasGetVector(M, sizeof(float), d_vector,
                  h_vector, 1);

  cublasFree(d_vector);
  ...
}
```

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

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

# Experimental setup

The nVIDIA 8800 Ultra is a CUDA architecture

| | CPU | GPU |
|---|---|---|
| Processor | Intel Core 2 Duo | NVIDIA 8800 Ultra |
| Codename | Crusoe E6320 | G80 |
| Clock frequency | 1.86 GHz | 575 MHz |
| Memory speed | $2 \times 333$ MHz | $2 \times 900$ 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) | |

Use of MKL on the CPU, and CUDA and CUBLAS 1.0 on the GPU

# Evaluation and tuning of Level 3 (CU)BLAS

## GEMM

$$C := \beta \cdot C + \alpha \cdot op(A) \cdot op(B)$$

## SYRK

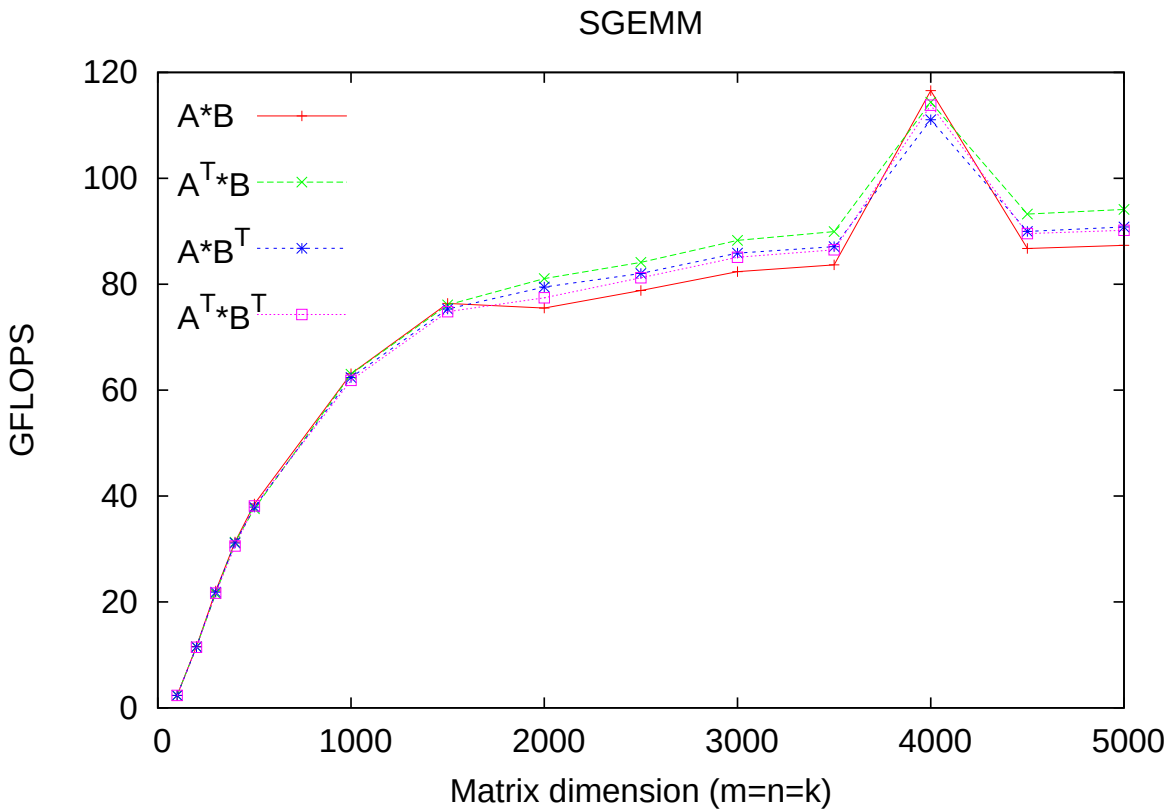
$$C := \beta \cdot C + \alpha \cdot A \cdot A^T \text{ or}$$
$$C := \beta \cdot C + \alpha \cdot A^T \cdot A$$

where $op(X) = X$ or $X^T$

## Others

Similar results observed for TRSM, and are to be expected for TRMM and SYMM

# GEMM evaluation. Experimental results



# GEMM evaluation. Main remarks

## Main remarks

- Peak performance is $\sim$116 Gflops for GEMM

# GEMM evaluation. Main remarks

## Main remarks

- Peak performance is $\sim$116 Gflops for GEMM
- Results attained for big matrices are much better than those for small-medium sized matrices

$$\Downarrow$$

**Stream-oriented architecture**

# GEMM evaluation. Main remarks

## Main remarks

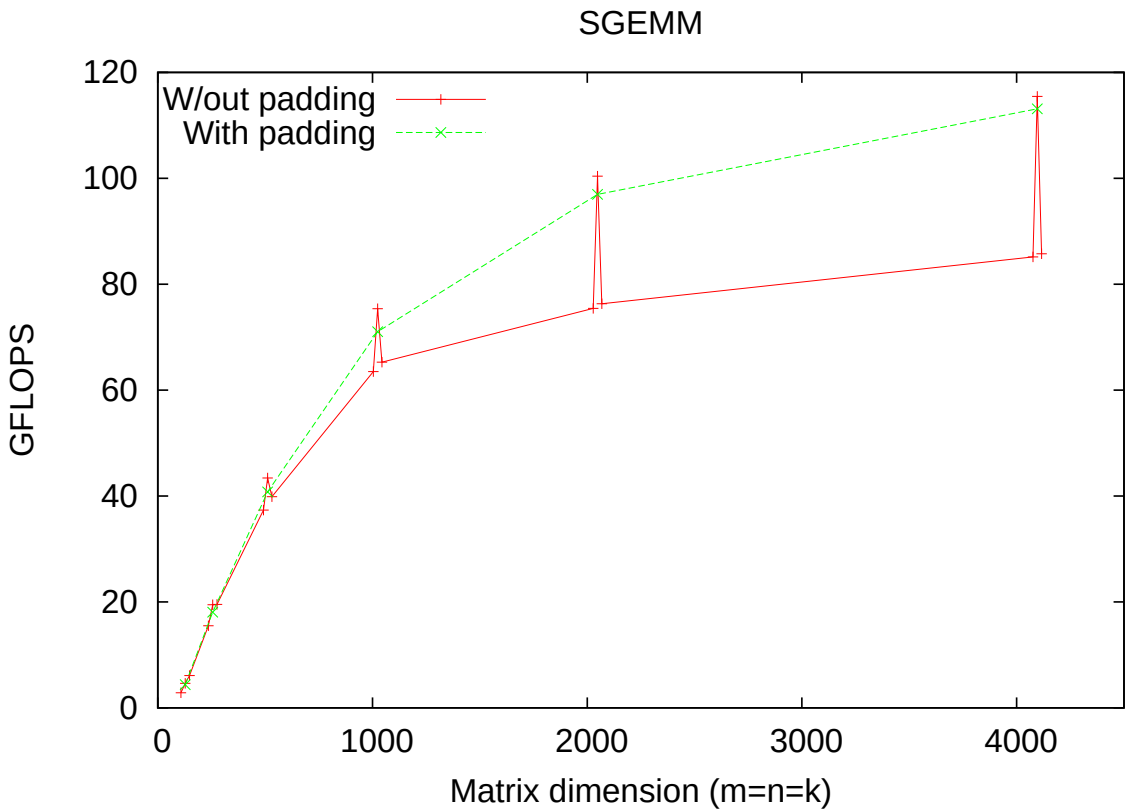
- Peak performance is ∼116 Gflops for GEMM
- Results attained for big matrices are much better than those for small-medium sized matrices

$$\Downarrow$$

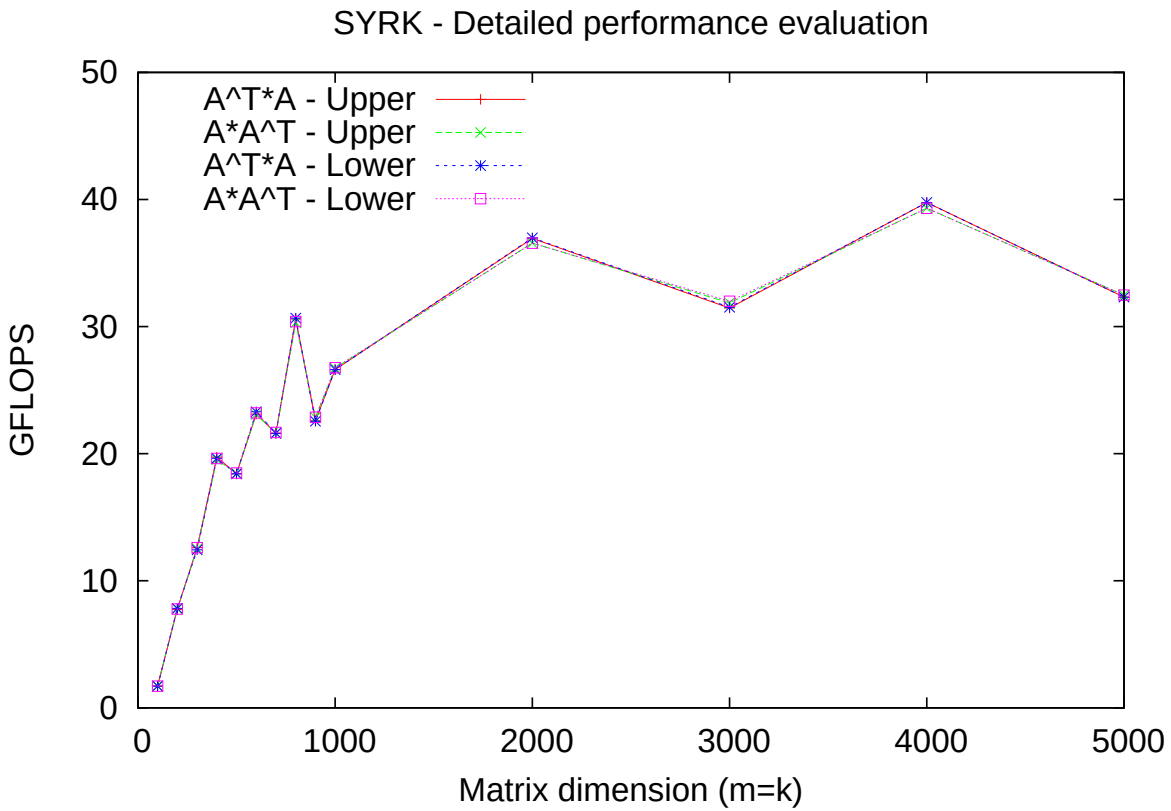
**Stream-oriented architecture**

- The peak observed for $m = 4000$ is also observed for all dimensions multiple of 32
- Proposal: apply padding!

# GEMM padding. Experimental results



# SYRK evaluation. Experimental results



# SYRK evaluation. Main remarks

## Main remarks

- Peak performance is ∼40 Gflops for SYRK ⇒ Suboptimal implementation

# SYRK evaluation. Main remarks

## Main remarks

- Peak performance is ∼40 Gflops for SYRK ⇒ Suboptimal implementation
- As for GEMM, results attained for big matrices are better

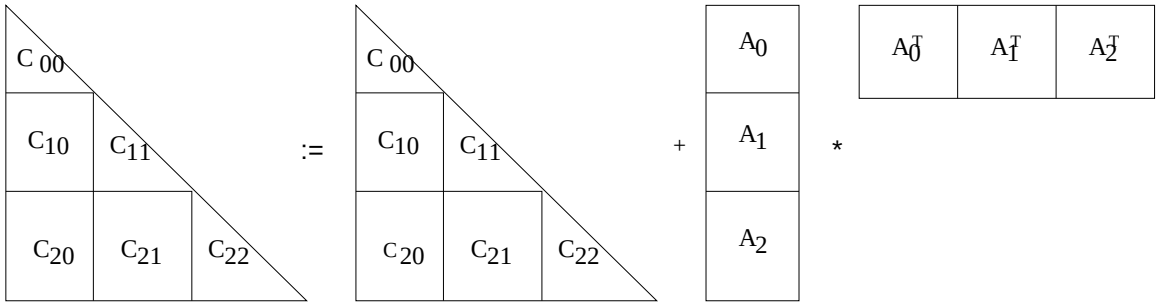
# SYRK evaluation. Main remarks

## Main remarks

- Peak performance is ∼40 Gflops for SYRK ⇒ Suboptimal implementation
- As for GEMM, results attained for big matrices are better
- Proposal: build SYRK on top of GEMM and use padding

# SYRK tuning

- Partition the SYRK operands:

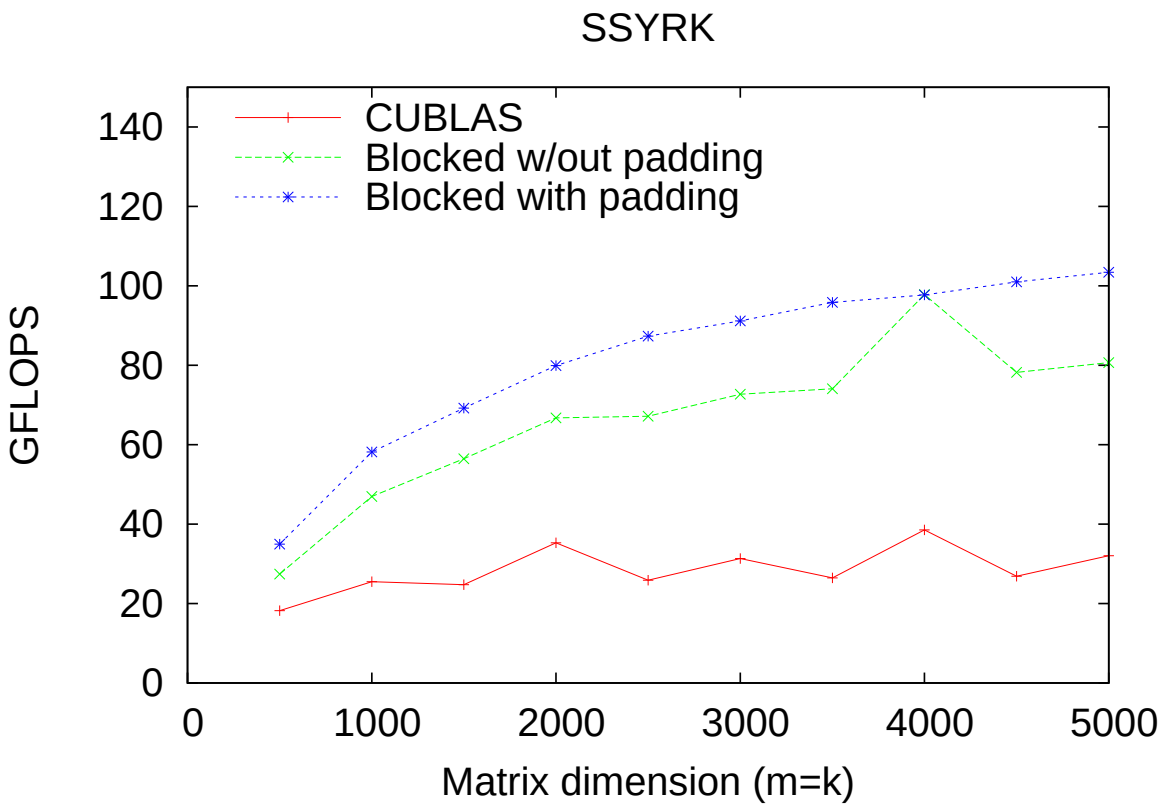



- The second block of columns of $C$ is given by:

$$C_{11} := \beta \cdot C_{11} + \alpha \cdot A_1 \cdot A_1^T$$

$$C_{21} := \beta \cdot C_{21} + \alpha \cdot A_2 \cdot A_1^T$$

# SYRK tuning. Experimental results

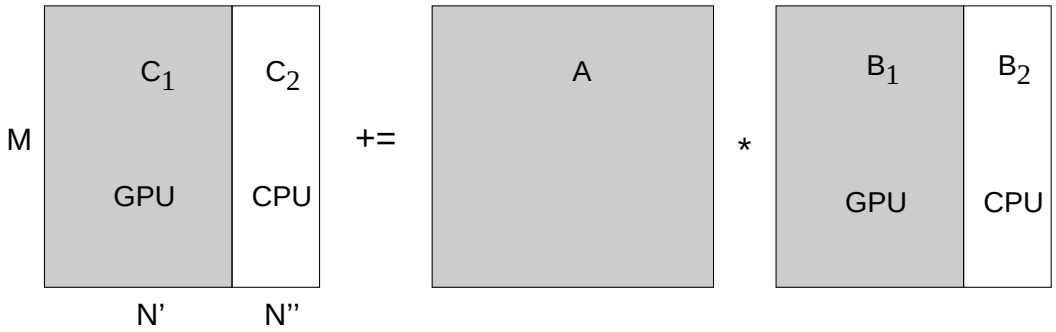


# Partitioning for larger matrices

- Transfer times GPU$\leftrightsquigarrow$CPU are an important bottleneck
- Our blocked version of GEMM allows to overlap:
  - The partial multiplication $A_p$ and $B_p$ and
  - The transference of the next pair of blocks $A_{p+1}$ and $B_{p+1}$, being $A_p$ and $B_p$ blocks of columns of $A$ and $B$, respectively.
- Nor CUDA 1.0 neither G80 allow overlapping of communication and calculation (supported by more recent systems!)
- An orthogonal benefit: enables computation with larger matrices

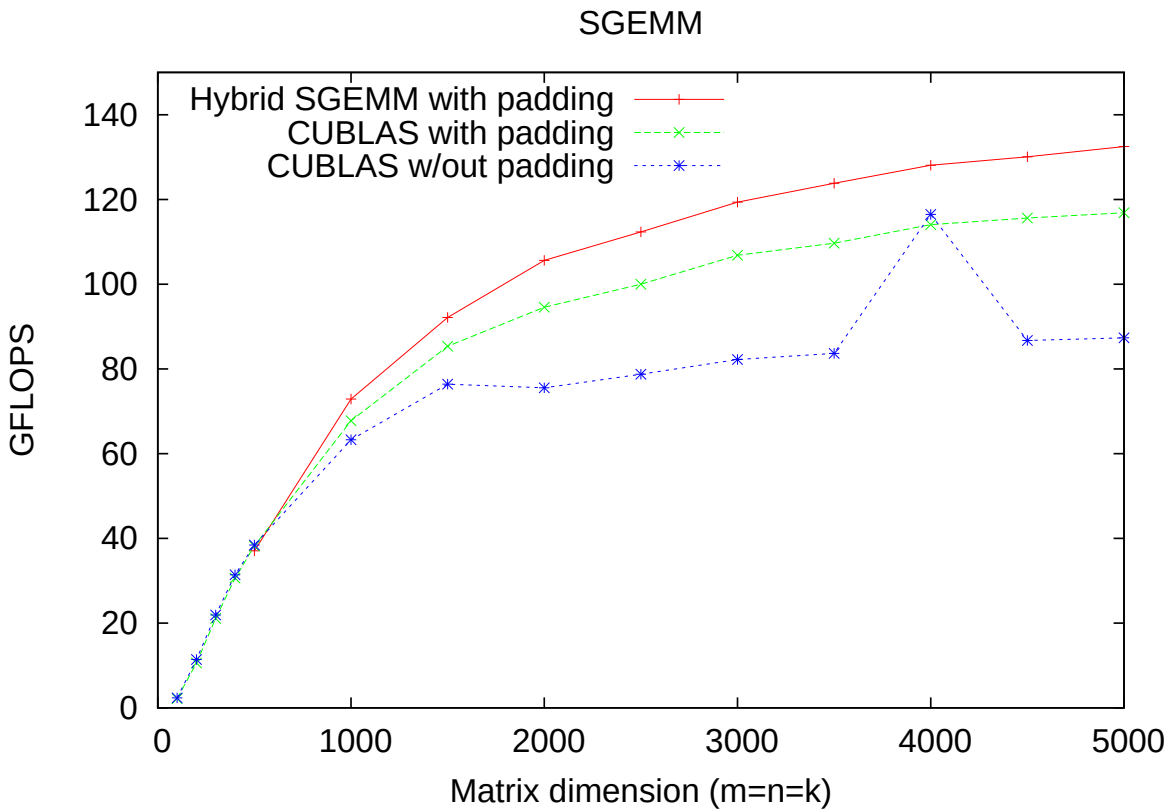
# Hybrid computation

- While GPU is performing a calculation, CPU can also be executing part of the computation.
- We implement a hybrid GEMM implementation following the decomposition:




- Values for $N'$ and $N''$ must be selected carefully to balance the computation load.

# Hybrid computation. Experimental results



# Design and implementation of LAPACK: linear systems

## Cholesky factorization

$$A = LL^T \quad (\text{or } A = U^T U)$$

with $L$ lower triangular (or $U$ upper triangular)

## Others

Similar results obtained for the LU factorization with partial pivoting, and are to be expected for the QR factorization (linear least-squares problems)

# Cholesky factorization. Blocked variants

**Algorithm:** $A := \text{CHOL\_BLK}(A)$

**Partition** ...
**where** ...
**while** $m(A_{TL}) < m(A)$ **do**
**Determine block size** $b$
**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 $b \times b$

Variant 1:

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

Variant 2:

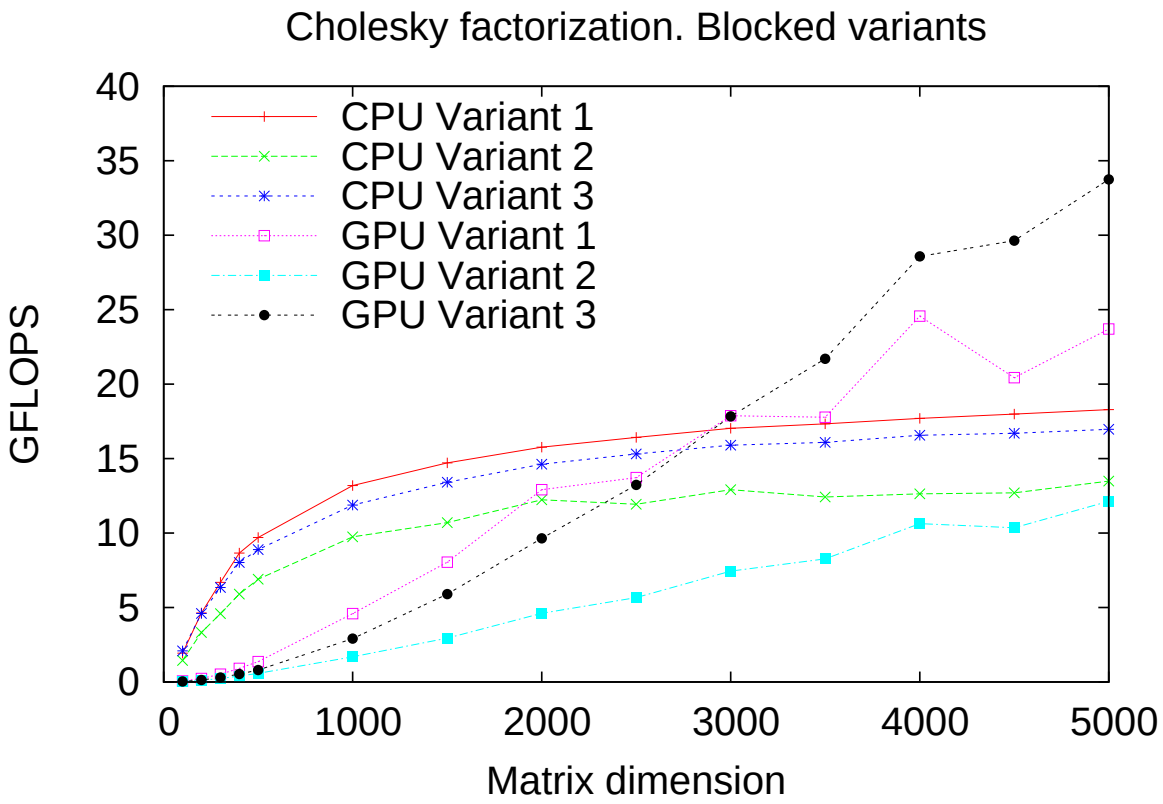
$$A_{10} := A_{10}\,\text{TRIL}\,(A_{00})^{-\text{T}}$$
$$A_{11} := A_{11} - A_{10}A_{10}^{\text{T}}$$
$$A_{11} := \text{CHOL\_UNB}(A_{11})$$

Variant 3:

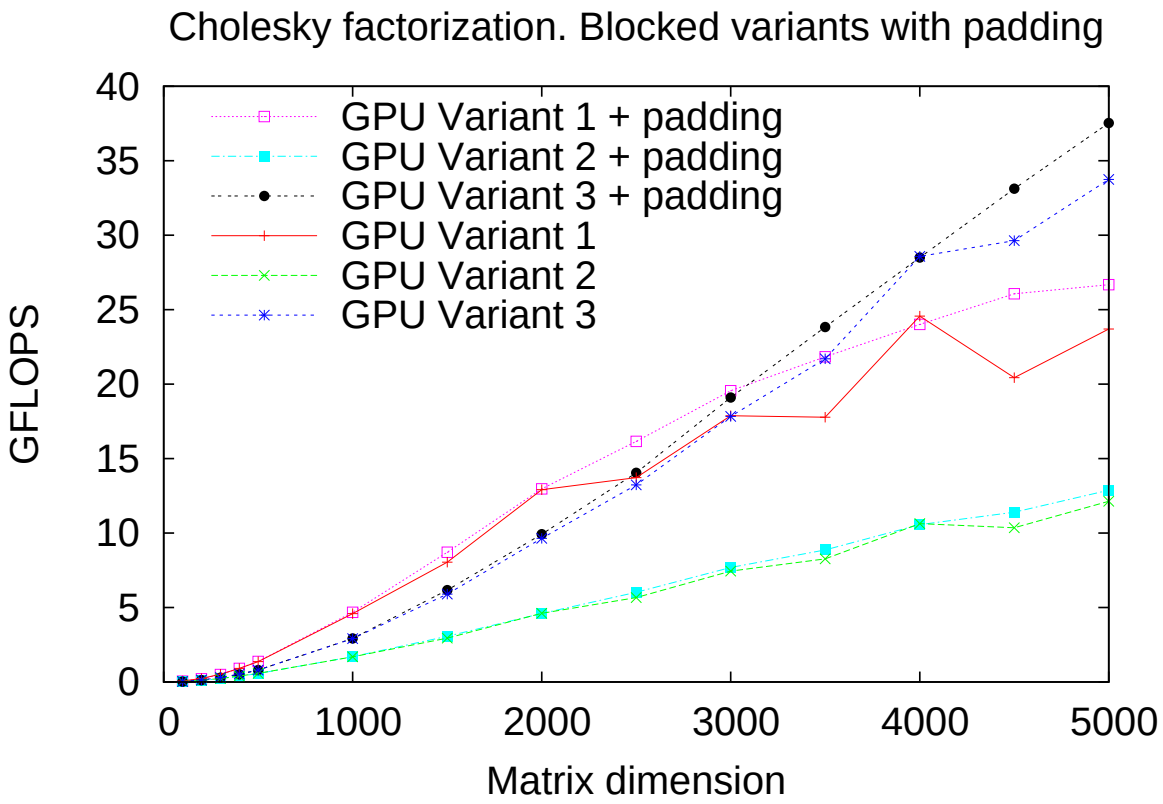
$$A_{11} := A_{11} - A_{10}A_{10}^{\text{T}}$$
$$A_{11} := \text{CHOL\_UNB}(A_{11})$$
$$A_{21} := A_{21} - A_{20}A_{10}^{\text{T}}$$
$$A_{21} := A_{21}\,\text{TRIL}\,(A_{11})^{-\text{T}}$$

**Continue with**
...
**endwhile**

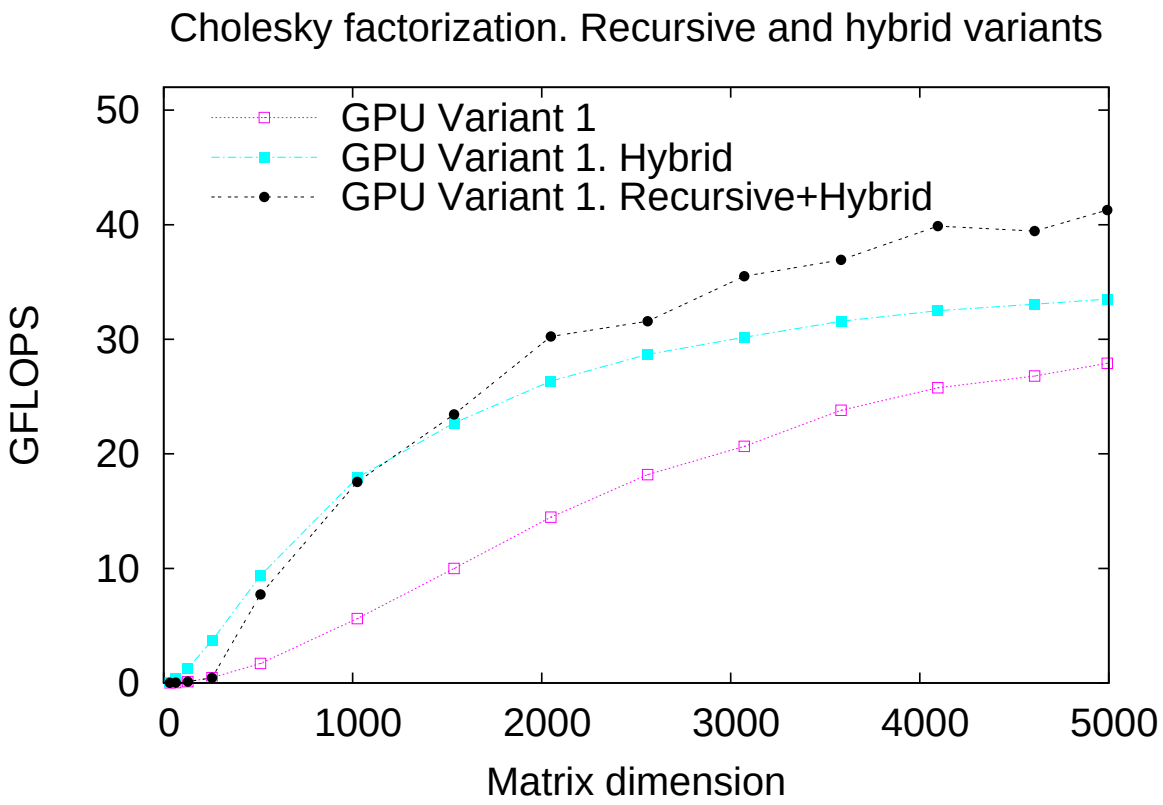
# Cholesky factorization. Experimental results



# Cholesky padding. Experimental results



# Cholesky hybrid and recursive. Experimental results



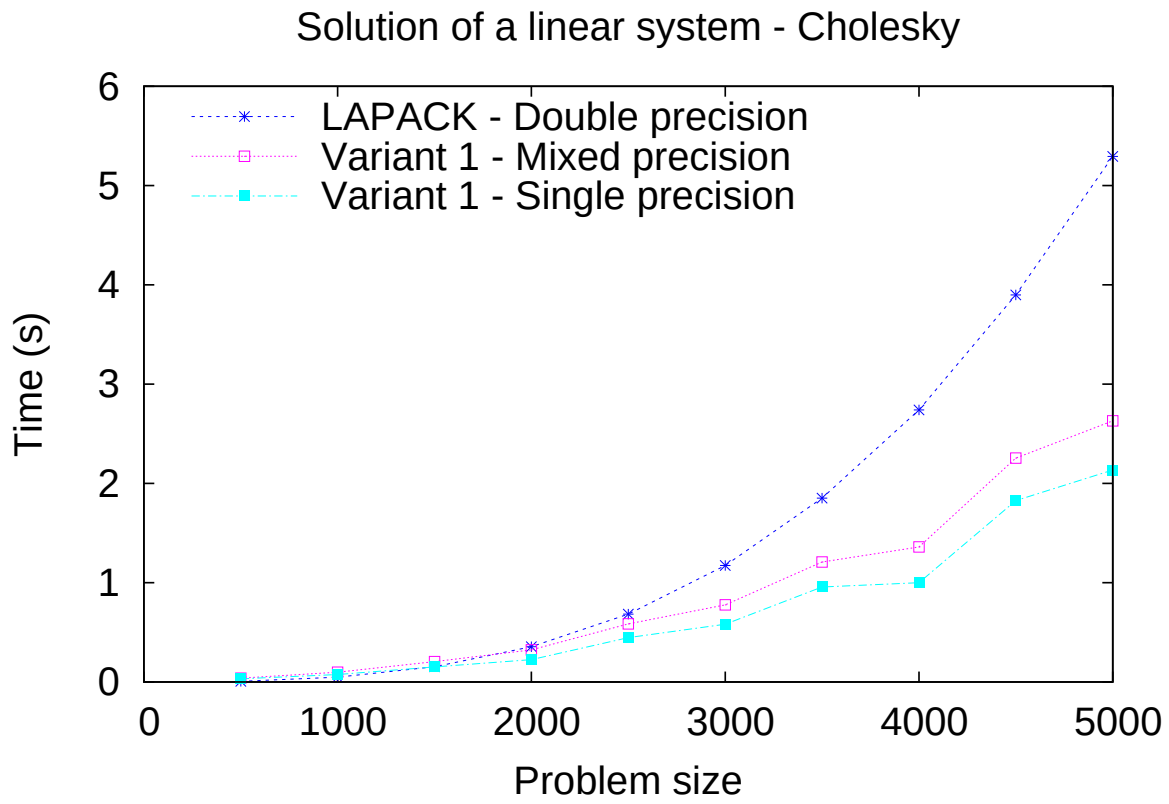
# Iterative refinement for extended precision

Compute the Cholesky factorization $A = LL^T$ and solve $(LL^T)x = b$ in the GPU

$\longrightarrow$ 32 bits of accuracy!

$i \leftarrow 1$
repeat
$r^{(i)} \leftarrow b - A \cdot x^{(i)}$
$r^{(i)}_{(32)} \leftarrow r^{(i)}$
$z^{(i)}_{(32)} \leftarrow L^{-T}_{(32)}(L^{-1}_{(32)} r^{(i)}_{(32)})$
$z^{(i)} \leftarrow z^{(i)}_{(32)}$
$x^{(i+1)} \leftarrow x^{(i)} + z^{(i)}$
$i \leftarrow i + 1$
until $\|r^{(i)}\| < \sqrt{\varepsilon}\|x^{(i)}\|$

# Iterative refinement. Experimental results



# What if multiple GPUs are available?

## Already here:

- Multiple ClearSpeed boards
- Multiple NVIDIA cards
- nVIDIA Tesla series

# What if multiple GPUs are available?

**Already here:**

- Multiple ClearSpeed boards
- Multiple NVIDIA cards
- nVIDIA Tesla series

How are we going to program these?

# Experimental setup

| | CPU | GPU |
|---|---|---|
| Processor | 16 x Intel Itanium2 | NVIDIA Tesla s870 (4 G80) |
| Clock frequency | 1.5 GHz | 575 MHz |

Use of MKL on the Itanium vs. CUDA and CUBLAS 1.1 on the Tesla

# SuperMatrix for multithreaded architectures

- Traditional (and pipelined) parallelizations are limited by the control dependencies dictated by the code
- The parallelism should be limited only by the data dependencies between operations!
- In dense linear algebra, imitate a superscalar processor: dynamic detection of data dependencies

# SuperMatrix for multithreaded architectures

```
int FLA_Chol_blk( FLA_Obj A )
{
  /* ... FLA_Part_2x2( ); ... */

  while ( FLA_Obj_width( ATL ) < FLA_Obj_width( A ) ){

    /* FLA_Repart_2x2_to_3x3( ); ... */
    /*-----------------------------------------------------------*/
    FLA_Chol( A11 );                    /* Chol( A11 )               */
    FLA_Trsm_rltn( FLA_ONE, A11,
                            A21 );      /* A21 := A21 * TRIL(A11)^-T */
    FLA_Syrk_ln( FLA_MINUS_ONE, A21,
                 FLA_ONE,       A22 );  /* A22 := A22 - A21 * A21^T  */
    /*-----------------------------------------------------------*/
    /* FLA_Cont_with_3x3_to_2x2( ); ... */
  }
}
```

The *FLAME runtime system* "pre-executes" the code:

- Whenever a routine is encountered, a pending task is annotated in a global task queue

# SuperMatrix for multithreaded architectures

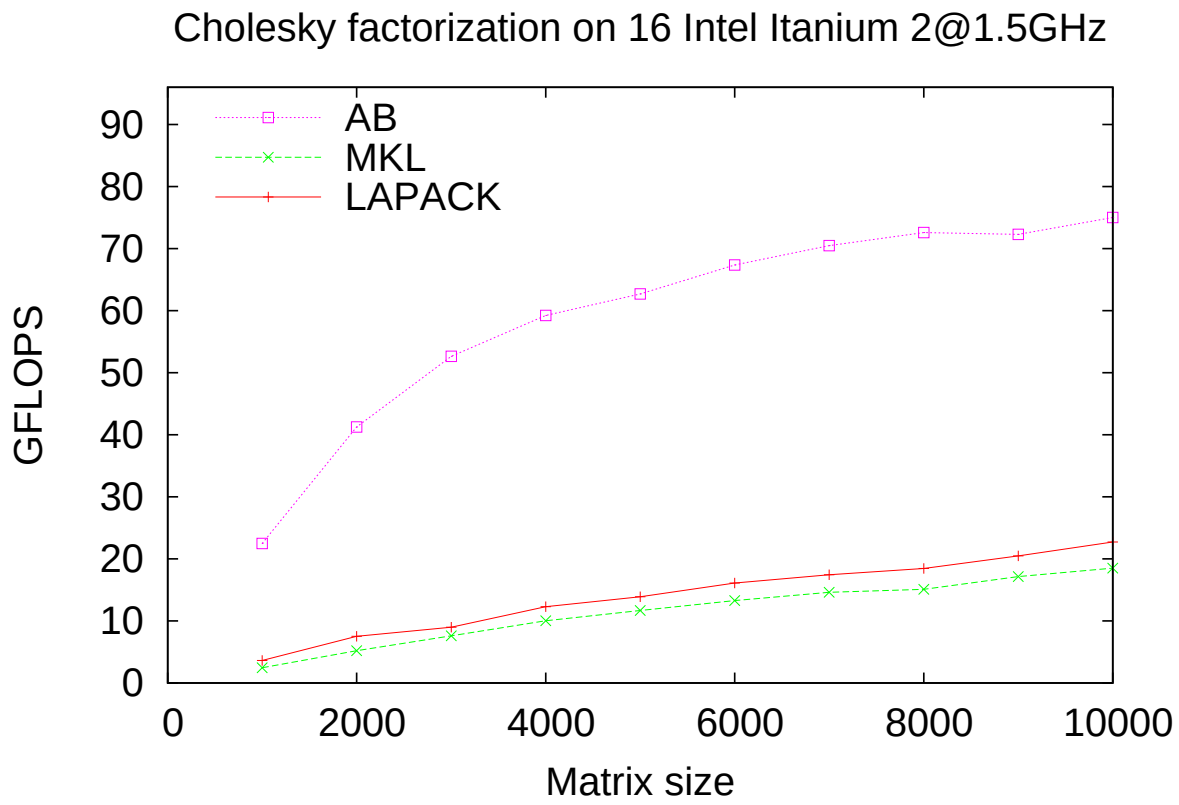
$$\left(\begin{array}{c|c|c} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} \\ \hline \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} \\ \hline \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} \end{array}\right) \quad \text{Runtime} \quad \rightarrow$$

1. $\bar{A}_{00} := \text{CHOL}\left(\bar{A}_{00}\right)$
2. $\bar{A}_{10} := \bar{A}_{10}\,\text{TRIL}\left(\bar{A}_{00}\right)^{-T}$
3. $\bar{A}_{20} := \bar{A}_{20}\,\text{TRIL}\left(\bar{A}_{00}\right)^{-T}$
4. $\bar{A}_{11} := \bar{A}_{11} - \bar{A}_{10}\bar{A}_{01}$
5. $\ldots$

## SuperMatrix

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

# SuperMatrix for multithreaded architectures

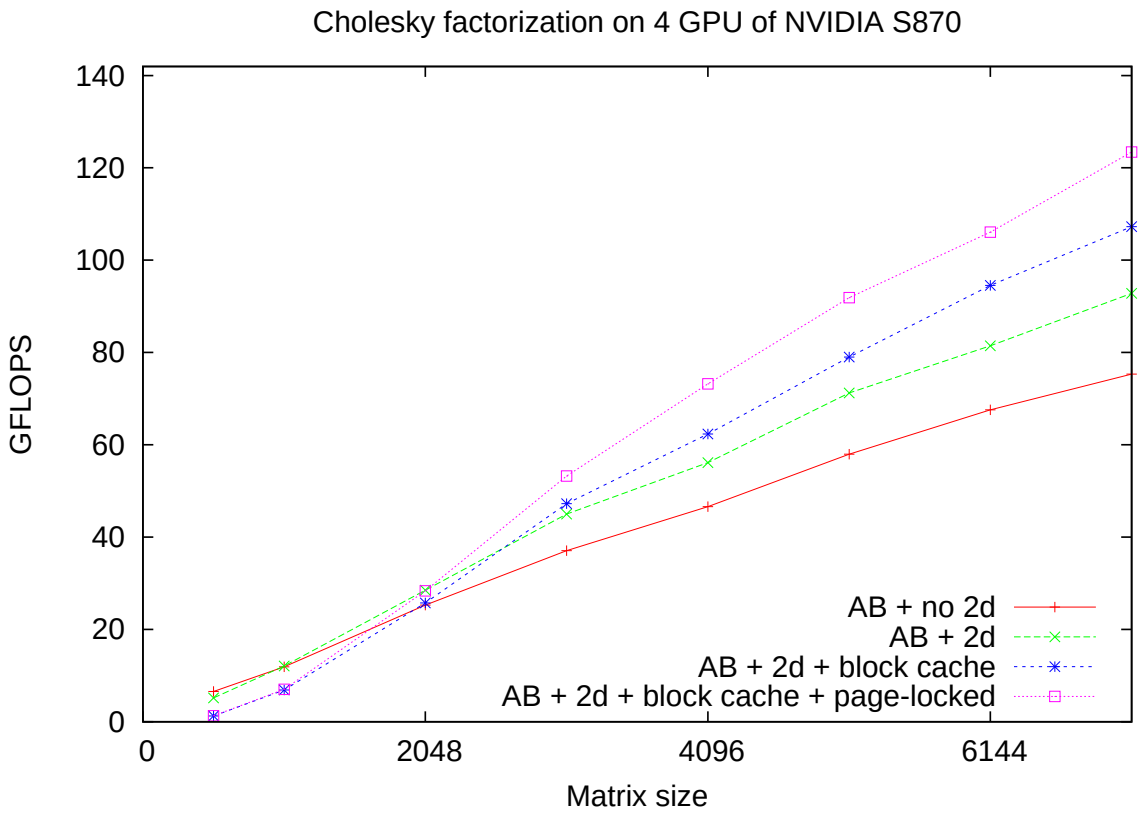


# Porting SuperMatrix to multiple GPUs

- Run-time system (scheduling), storage, and code are independent
- No significative modification to the FLAME codes: Interfacing to CUBLAS

A software effort of two hours!

# Porting SuperMatrix. Experimental results



Hope you enjoyed

Thanks!