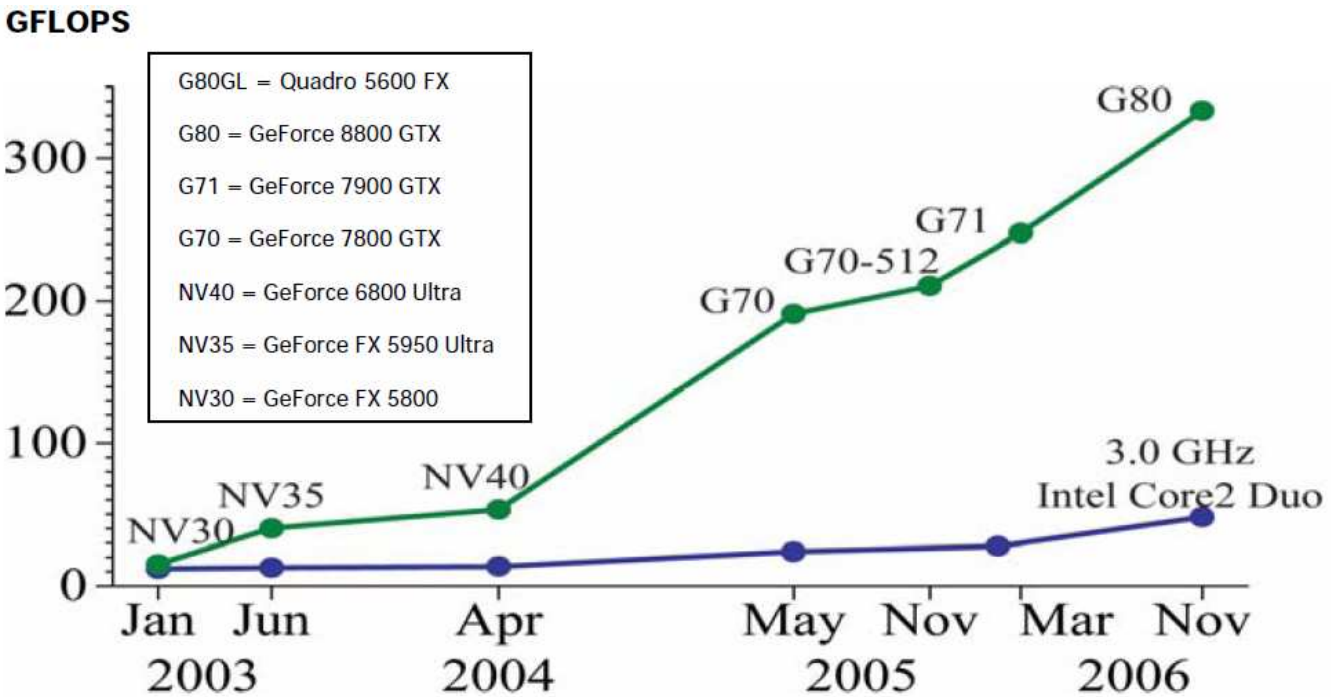
Solving Dense Linear Systems on Graphics Processors

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

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

Motivation (I)



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

Motivation (II)

- The solution of dense linear systems arises in a wide variety of fields
- How does the new generation of GPUs adapt to this type of problems?
- What optimization techniques can be applied to boost performance?
- Is really single precision support a problem?

Outline

Introduction

- Dense linear algebra has been a pioneering area to explore the performance of new architectures
- This fact continues with the advent of
  - Multicore processors
  - Hardware accelerators (GPUs, Cell B.E., ClearSpeed boards,...)
- The increase in performance, functionality and programmability of the current GPUs has renewed the interest in this class of hardware

Related work

- Galoppo et al. (2005): *"Efficient algorithms for solving dense linear systems on graphics hardware"*
  - Based on older GPUs (not *unified* architectures)
  - Based on graphics APIs (Cg and OpenGL)
- Volkov and Demmel (2008): *"LU, QR and Cholesky factorization using vector capabilities of GPUs"*
  - Using CUBLAS 2.0
  - Just one variant of each procedure
- Barrachina et al. (2008): *"Evaluation and tuning of the level 3 CUBLAS for graphics processors"*
  - Evaluating and tuning version 1.1 of CUBLAS
  - Gave us ideas to find the best variants of the factorizations

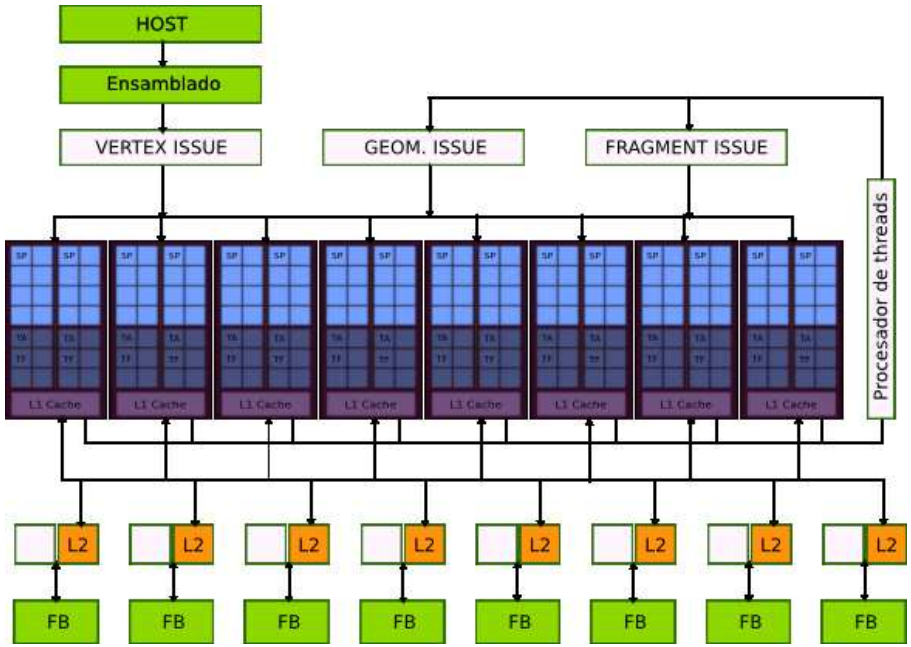
Goals

Our paper makes the following contributions:

- We propose a full set of variants for both the Cholesky and the LU factorization
- Variants are evaluated taking into account the underlying BLAS implementation (CUBLAS)
- All implementations are evaluated on modern GPUs (G80 processor)
- Several optimization techniques are described and successfully implemented:
  - Padding
  - Hybrid implementation
  - Recursive implementation
  - Iterative refinement

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 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 framework for writing C programs for execution on the GPU
- Consists of:
  - A minimal set of extensions to the C language
  - A runtime library of routines for controlling the transfers between video and main memory, run-time configuration, execution of device-specific functions, handling multiple GPUs,...

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 BLAS kernel
3. Transfer of results back to main memory

Our codes have been developed using FORTRAN, and the FORTRAN wrappers provided by NVIDIA

Cholesky and LU factorizations

Cholesky factorization

Given a s.p.d. matrix A,

$$A = LL^T, \tag{1}$$

where L is a lower triangular matrix known as the *Cholesky factor* of A.

LU factorization with partial pivoting

Given a matrix A, the LU factorization with partial pivoting decomposes this matrix into two matrices, L and U, such that

$$PA = LU, \tag{2}$$

where P is a permutation matrix, L is a unit lower triangular matrix, and U is an upper triangular matrix.

Variants for the Cholesky and LU factorization

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

Why different variants?

- All variants perform exactly the same operations
- However, their performance depends on the specific BLAS implementation employed
- CUBLAS is far from being fully optimized (Barrachina et al.)
- The CUBLAS routines used by each variant will determine their performance

Algorithmic variants. Cholesky factorization

Variant 1:

$$A_{11} := \text{CHOL\_UNB}(A_{11})$$
$$A_{21} := A_{21}\text{TRIL}\left(A_{11}\right)^{-\text{T}} \quad (\text{TRSM})$$
$$A_{22} := A_{22} - A_{21}A_{21}^{\text{T}} \quad (\text{SYRK})$$

Variant 2:

$$A_{10} := A_{10}\text{TRIL}\left(A_{00}\right)^{-\text{T}} \quad (\text{TRSM})$$
$$A_{11} := A_{11} - A_{10}A_{10}^{\text{T}} \quad (\text{SYRK})$$
$$A_{11} := \text{CHOL\_UNB}(A_{11})$$

Variant 3:

$$A_{11} := A_{11} - A_{10}A_{10}^{\text{T}} \quad (\text{SYRK})$$
$$A_{11} := \text{CHOL\_UNB}(A_{11})$$
$$A_{21} := A_{21} - A_{20}A_{10}^{\text{T}} \quad (\text{GEMM})$$
$$A_{21} := A_{21}\text{TRIL}\left(A_{11}\right)^{-\text{T}} \quad (\text{TRSM})$$

Algorithmic variants. LU factorization

Variant 1:

$$\left(\begin{array}{c} \frac{A_{01}}{A_{11}} \\ \hline A_{21} \end{array}\right) := P(p_0) \left(\begin{array}{c} \frac{A_{01}}{A_{11}} \\ \hline A_{21} \end{array}\right)$$

$$A_{01} := \text{TRILU}(A_{00})^{-1} A_{01} \qquad (\text{TRSM})$$

$$A_{11} := A_{11} - A_{10} A_{01} \qquad (\text{GEMM})$$

$$A_{21} := A_{21} - A_{20} A_{01} \qquad (\text{GEMM})$$

$$\left[\left(\frac{A_{11}}{A_{21}}\right), p_1\right] := \text{LUP\_UNB}\left(\frac{A_{11}}{A_{21}}\right)$$

$$\left(\frac{A_{10}}{A_{20}}\right) := P(p_1) \left(\frac{A_{10}}{A_{20}}\right)$$

Algorithmic variants. LU factorization

Variant 2:

$$A_{11} := A_{11} - A_{10}A_{01} \quad (\text{GEMM})$$

$$A_{21} := A_{21} - A_{20}A_{01} \quad (\text{GEMM})$$

$$\left[\begin{pmatrix} A_{11} \\ \hline A_{21} \end{pmatrix}, p_1 \right] := \text{LUP\_UNB} \left(\begin{pmatrix} A_{11} \\ \hline A_{21} \end{pmatrix} \right)$$

$$\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array} \right) := P(p_1) \left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array} \right)$$

$$A_{12} := A_{12} - A_{10}A_{02} \quad (\text{GEMM})$$

$$A_{12} := \text{TRILU}(A_{11})^{-1}A_{12} \quad (\text{TRSM})$$

Algorithmic variants. LU factorization

Variant 3:

$$\left[\left(\frac{A_{11}}{A_{21}}\right), p_1\right] := \mathrm{LUP\_UNB}\left(\frac{A_{11}}{A_{21}}\right)$$

$$\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right) := P(p_1)\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right)$$

$$A_{12} := \mathrm{TRILU}(A_{11})^{-1} A_{12} \qquad (\mathrm{TRSM})$$

$$A_{22} := A_{22} - A_{21} A_{12} \qquad (\mathrm{GEMM})$$

Padding

- Barrachina et al. have shown that Level 3 CUBLAS is far from optimized
- Kernels deliver better performance for certain matrix dimensions (multiple of 32) $\Rightarrow$ memory alignment issues
- It is possible to take benefit for the Cholesky (and LU) factorizations:
  - Starting from a block size n_b that is multiple of 32, we pad the $n \times n$ matrix A:

$$\bar{A} = \begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix} = \begin{pmatrix} L & 0 \\ 0 & I_k \end{pmatrix} \begin{pmatrix} L & 0 \\ 0 & I_k \end{pmatrix}^T,$$

  - I_k denotes the identity matrix of order k,
  - k is the difference between the matrix size n and the nearest integer multiple of n_b larger than n
  - all BLAS-3 calls operate on submatrices of dimensions that are a multiple of 32

Hybrid algorithm

- Goal: Exploit the different abilities of each processor to deal with specific operations
- Two main advantages of the CPU:
  1. Higher performance with small matrices
  2. Higher performance for some fine-grained operations (square root)
- Hybrid algorithm (Cholesky):
  1. Sends the diagonal block from video memory to main memory
  2. Factorizes this block on the CPU
  3. Transfers back the results to video memory
  4. The factorization continues on GPU

Recursive implementation

- Recursive implementation: partitions the matrix into 2×2 square blocks
- Factorizes the upper-left block using the same blocked algorithm
- The procedure is then repeated recursively at each deeper level

Iterative refinement (I)

- The G80 only provides single precision
- Iterative refinement can be used to regain full precision
- Mixed precision approach introduced by Buttari et al. for the Cell B.E. processor
- Can be used with any type of accelerator using single precision
- Procedure:
  1. Factorization of matrix A is performed on the GPU (single precision)
  2. A first solution is computed from the factors
  3. Iteratively, the solution is refined on CPU to double-precision arithmetic

Iterative refinement (II)

Solution of a S.P.D. using mixed precision and iterative refinement:

$A_{(32)}, b_{(32)} \leftarrow A, b$
$L_{(32)} \leftarrow \text{GPU\_CHOL\_BLK}(A_{(32)})$
$x^{(1)}_{(32)} \leftarrow L^{-T}_{(32)}(L^{-1}_{(32)} b_{(32)})$
$x^{(1)} \leftarrow x^{(1)}_{(32)}$
$i \leftarrow 0$
repeat
  $i \leftarrow i + 1$
  $r^{(i)} \leftarrow b - A \cdot x^{(i)}$
  $r^{(i)}_{(32)} \leftarrow r^{(i)}$
  $z^{(i)}_{(32)} \leftarrow L^{-T}_{(32)}(L^{-1}_{(32)} r^{(i)}_{(32)})$
  $z^{(i)} \leftarrow z^{(i)}_{(32)}$
  $x^{(i+1)} \leftarrow x^{(i)} + z^{(i)}$
until $x^{(i+1)}$ **is accurate enough**

- $O(n^3)$ work is done in lower precision (GPU)
- $O(n^2)$ work is done in higher precision (CPU)

Experimental setup

Experimental setup

- System based on an Intel Core2Duo CPU (1.86 Ghz)
- GPU: Nvidia 8800 Ultra board (G80 processor)
- Used CUDA and CUBLAS 1.0 for our evaluation purposes
- GotoBLAS 1.19 and LAPACK 3.0 when necessary
- GNU Fortran Compiler 3.3.5 and NVCC release 1.0

Developed software

- FORTRAN code developed, based on LAPACK
- Ease of programming: far from Cg + OpenGL
- Blocked and unblocked versions of the codes

Experimental setup

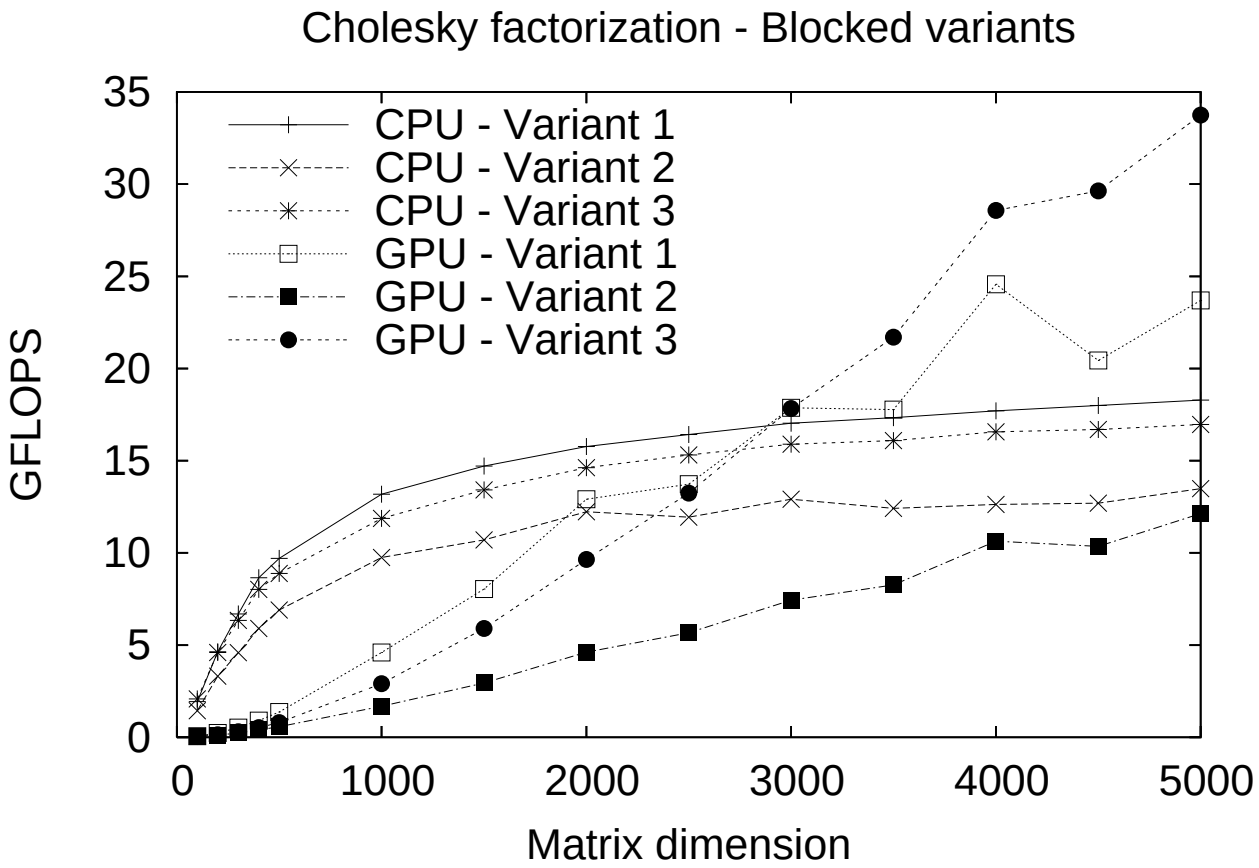
Experimental setup

- System based on an Intel Core2Duo CPU (1.86 Ghz)
- GPU: Nvidia 8800 Ultra board (G80 processor)
- Used CUDA and CUBLAS 1.0 for our evaluation purposes
- GotoBLAS 1.19 and LAPACK 3.0 when necessary
- GNU Fortran Compiler 3.3.5 and NVCC release 1.0

Developed software

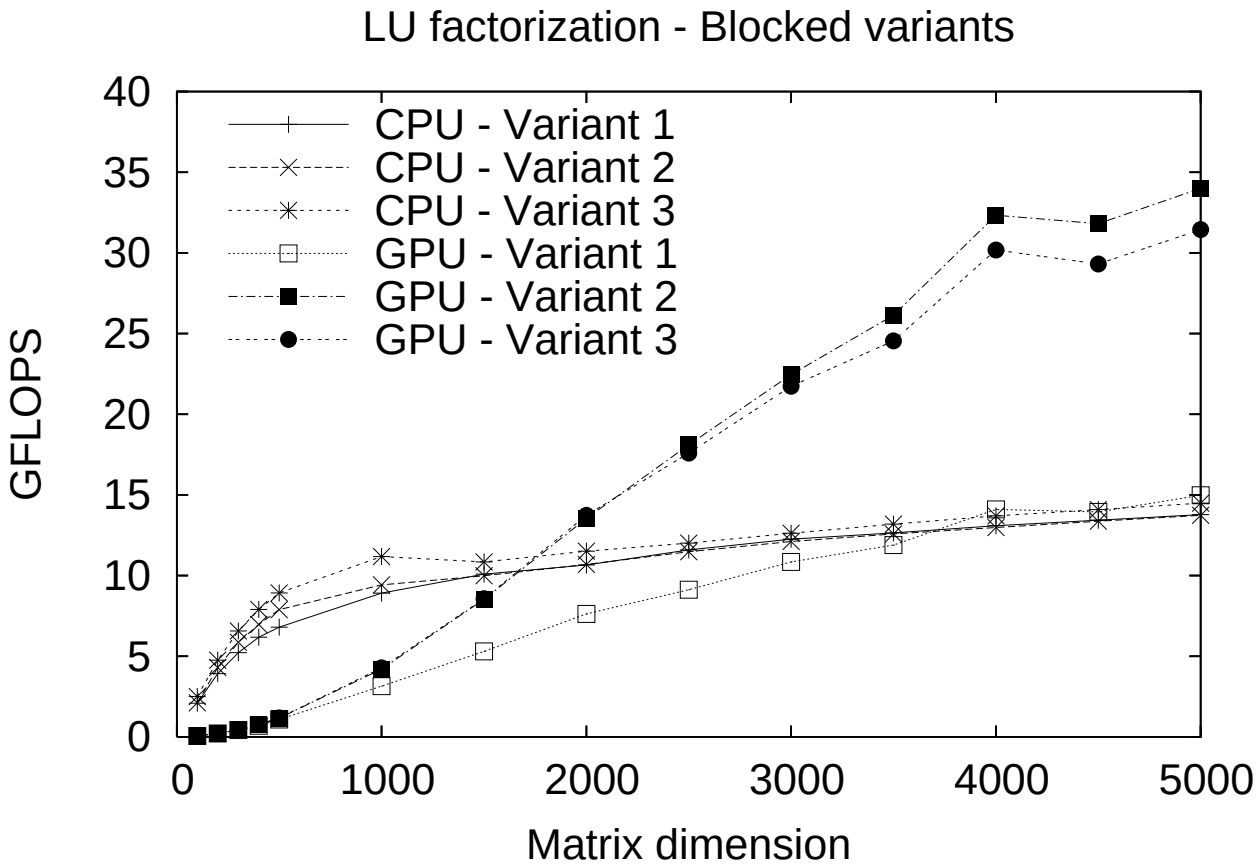
- FORTRAN code developed, based on LAPACK
- Ease of programming: far from Cg + OpenGL
- Blocked and unblocked versions of the codes

Blocked Cholesky variants



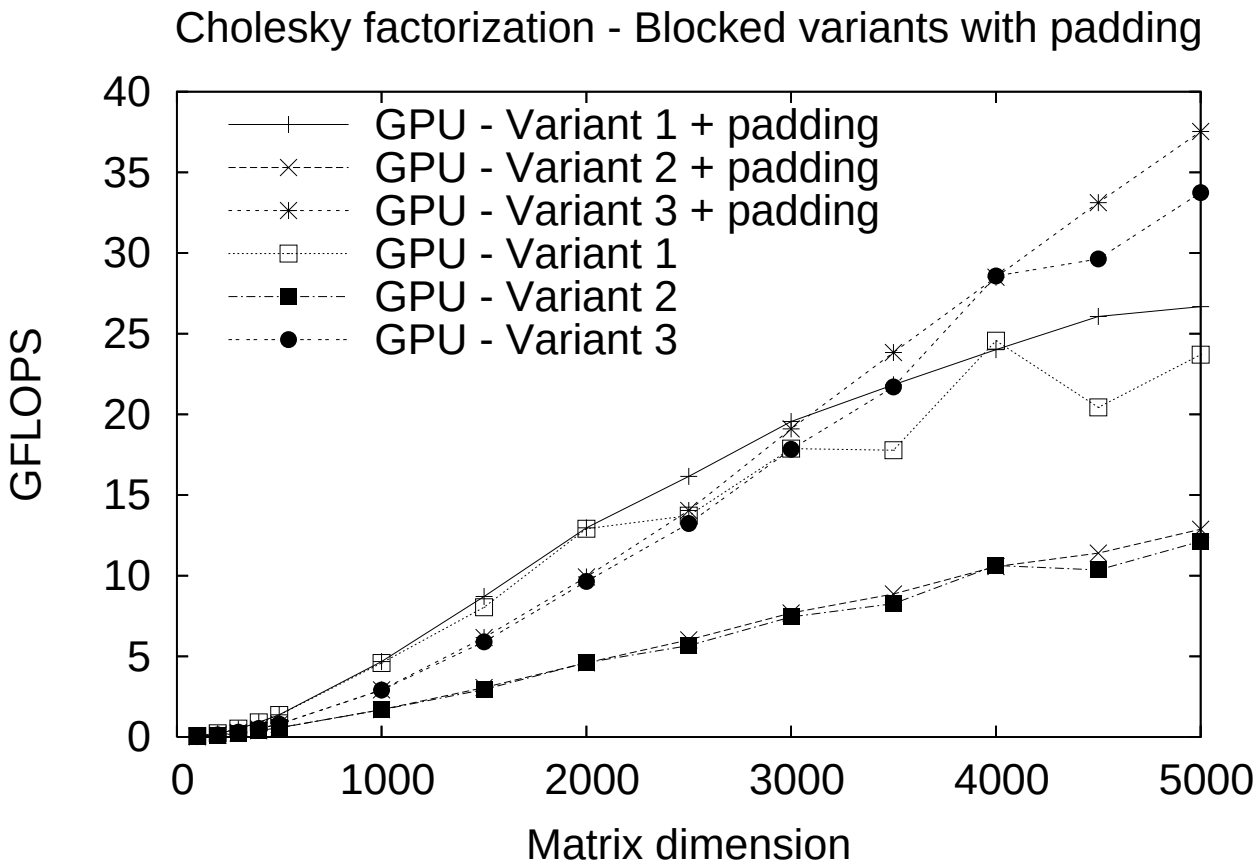
- Only shown blocked variants $\Rightarrow$ better performance
- GPU outperforms CPU for large dimensions
- Note the differences between variants

Blocked LU variants



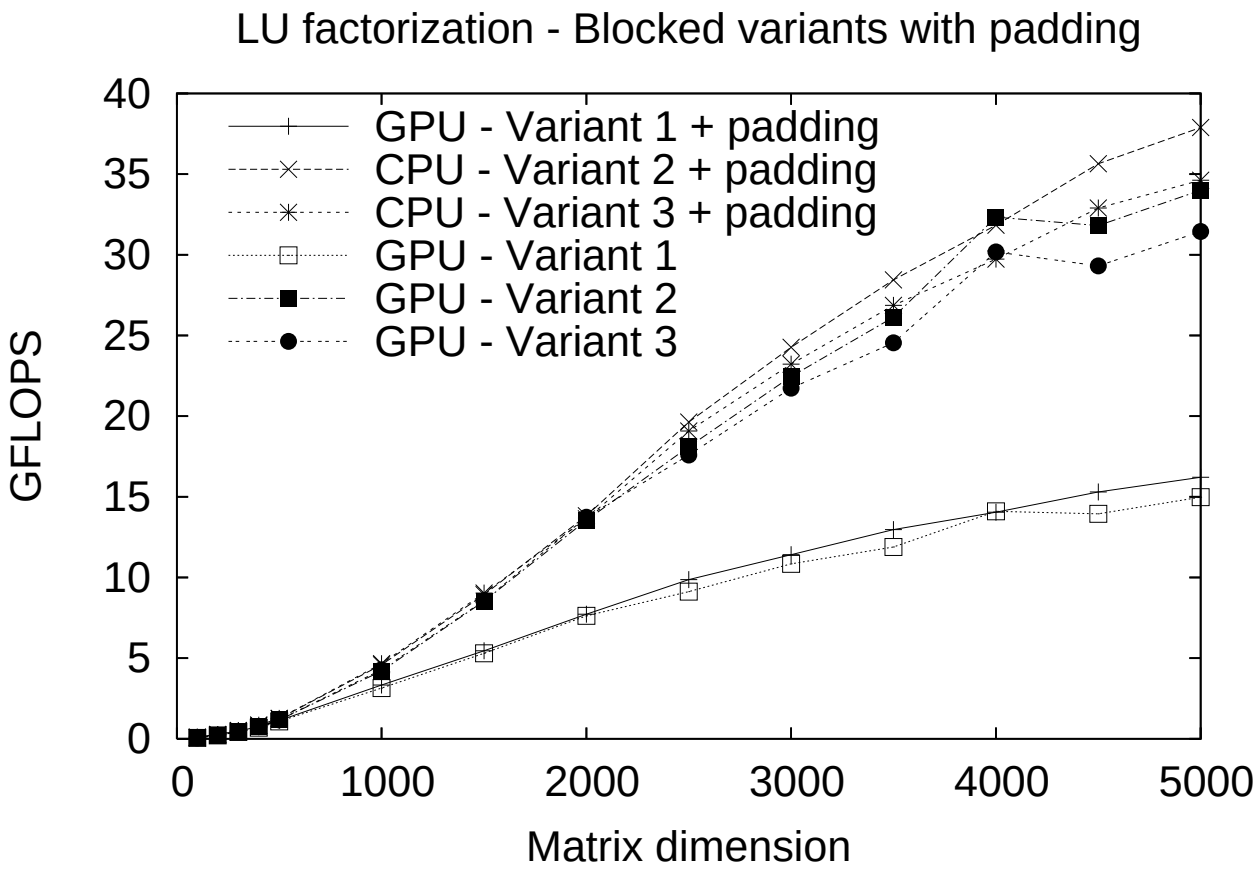
- Similar behavior than Cholesky

Blocked Cholesky with padding

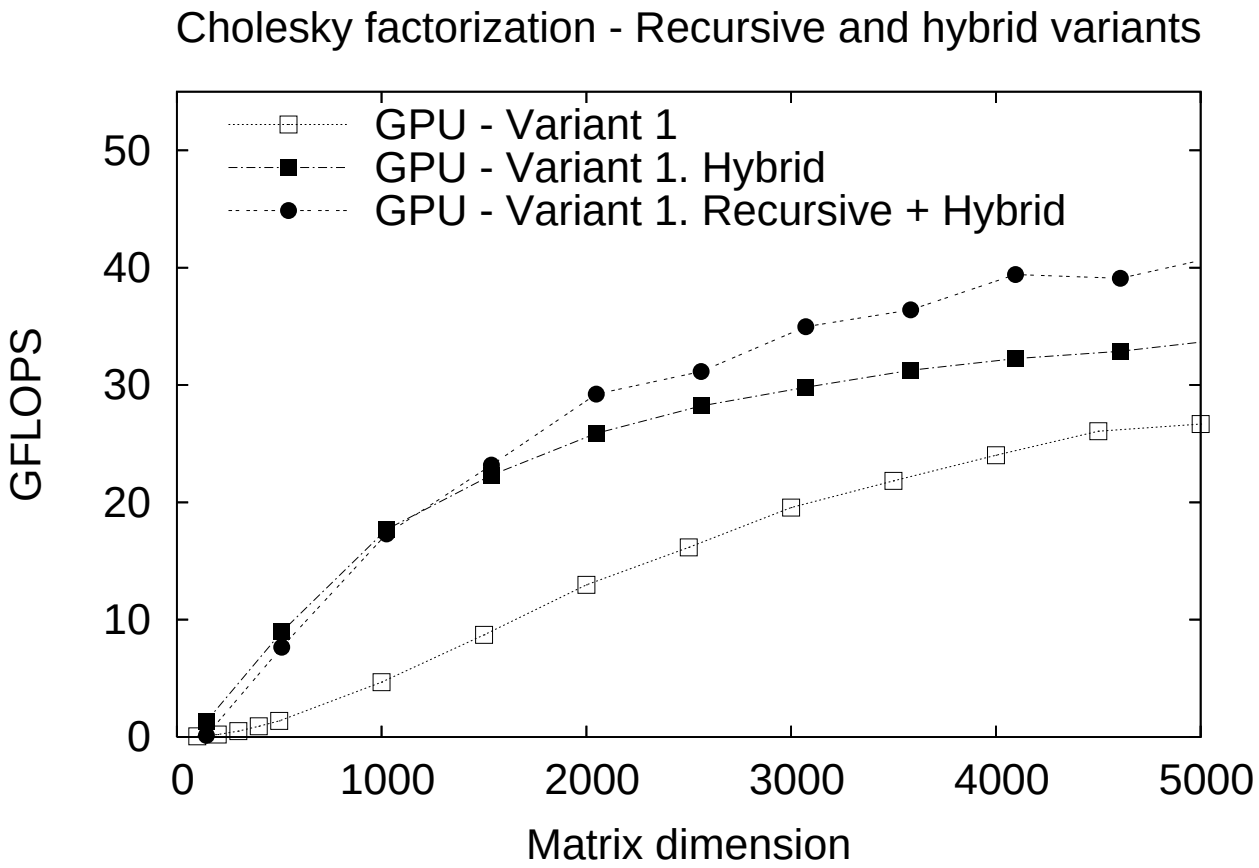


- Improving the performance of CUBLAS implies an improvment in our results
- Note how the irregularities in the performance disappear

Blocked LU with padding

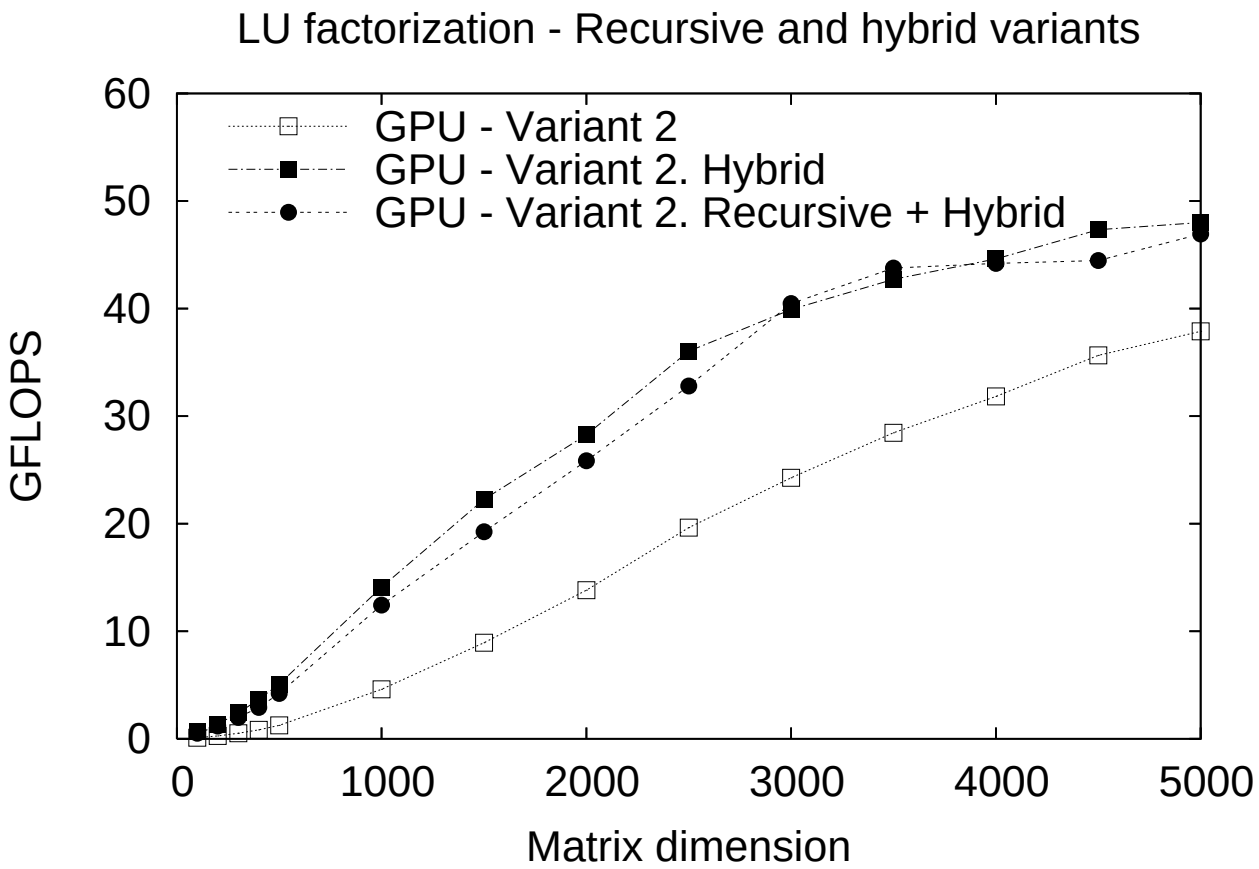


Hybrid computation. Cholesky factorization

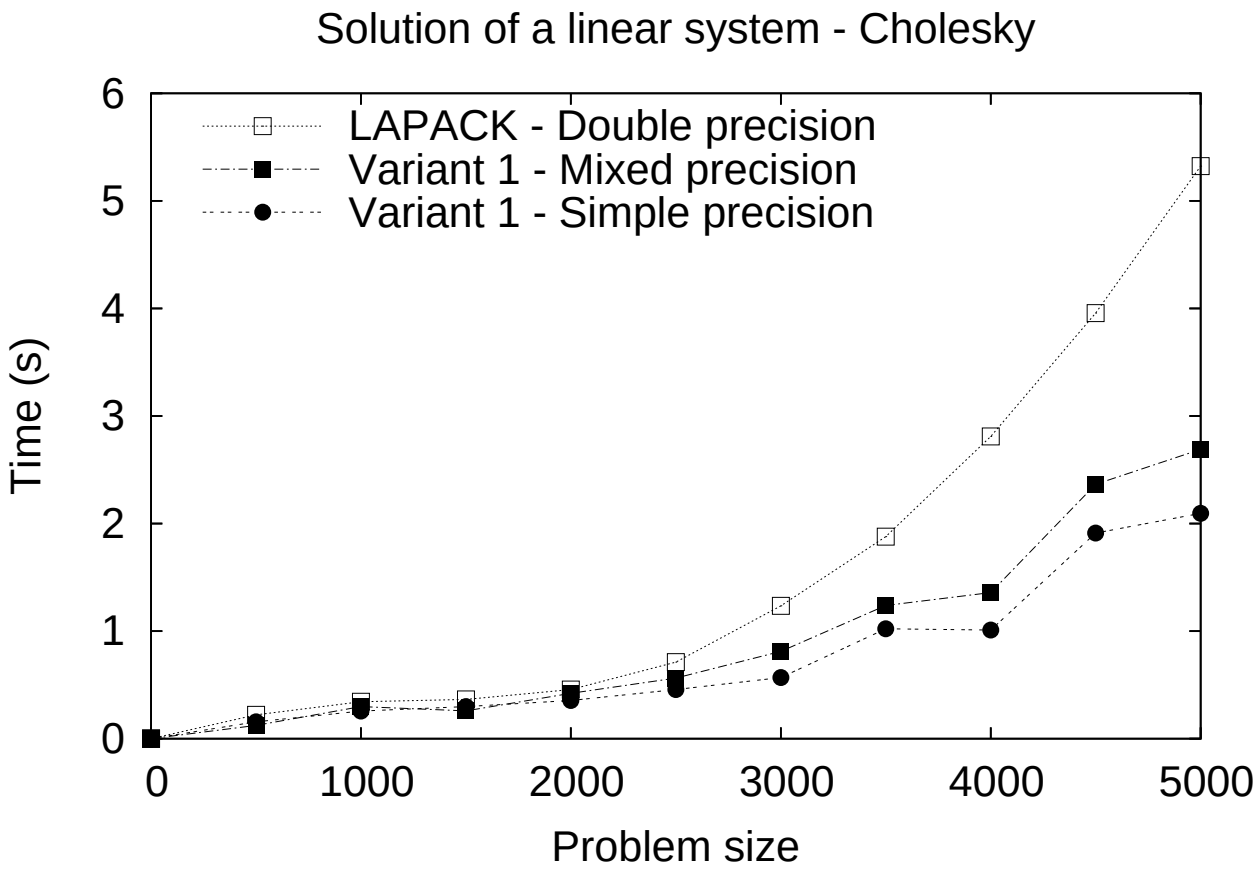


- No overhead associated with the factorization of the small current diagonal block
- Similar behaviour to be expected for the other variants

Hybrid computation. LU factorization

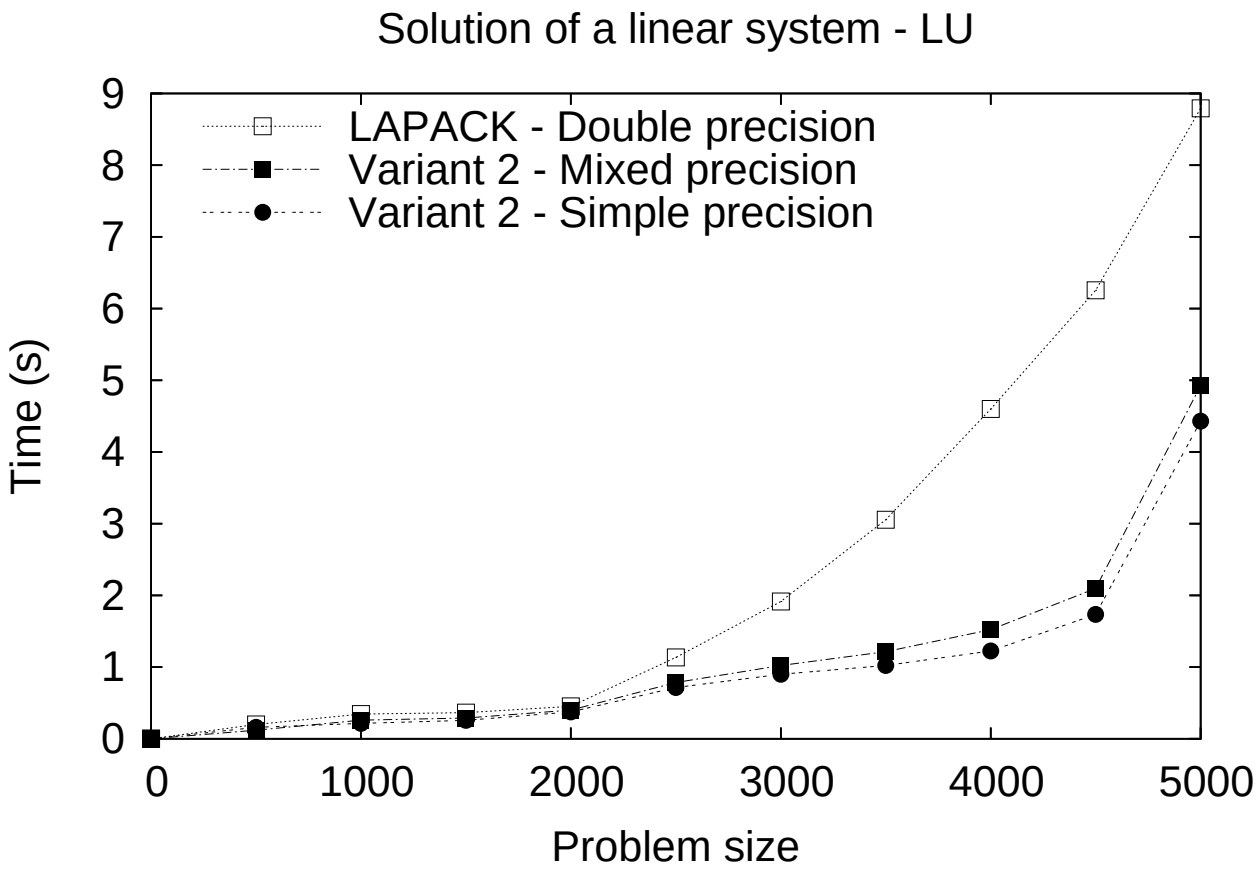


Iterative refinement. Cholesky factorization



- Iterative refinement introduces some overhead, but results are better than those on the CPU

Iterative refinement. LU factorization



Conclusions

- Our study reveals the most suitable variants for the Cholesky and LU implementations for unified GPUs
- We report how techniques such as padding, hybrid CPU-GPU computation, and recursion are effective to attain better performance
- Iterative refinement with mixed precision is an inexpensive technique to regain full accuracy
- This idea can be exported to other accelerators with simple precision accuracy
- Similar ideas can be applied to other procedures, such as the QR factorization

Future work

Which is the future of GPGPU?

Maybe, multi-GPU systems

- Nvidia Tesla systems
- Double precision support
- Programming style: DAG + runtime
- Ideas can be applied to other multi-accelerator systems, not only GPUs

Future work

Which is the future of GPGPU?
Maybe, multi-GPU systems

- Nvidia Tesla systems
- Double precision support
- Programming style: DAG + runtime
- Ideas can be applied to other multi-accelerator systems, not only GPUs

Thanks for your attention!!