Informe Técnico ICC 01-11-2008

Exploiting the capabilities of modern GPUs for dense matrix computations

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

Noviembre de 2008

Departamento de Ingeniería y Ciencia de Computadores

Correo electrónico: {`barrachi, castillo, figual, mayo, quintana, gquintan`}`@icc.uji.es`

Universidad Jaime I
Campus de Riu Sec, s/n
12.071 - Castellón
España

Exploiting the capabilities of modern GPUs for dense matrix computations

Sergio Barrachina[1],
Maribel Castillo[2],
Francisco D. Igual[3],
Rafael Mayo[4],
Enrique S. Quintana-Ortí[5],
Gregorio Quintana-Ortí[6],

Abstract:

We present several algorithms to compute the solution of a linear system of equations on a GPU, as well as general techniques to improve their performance, such as padding and hybrid GPU-CPU computation. We compare single and double precision performance of a modern GPU with unified architecture, and show how iterative refinement with mixed precision can be used to regain full accuracy in the solution of linear systems, exploiting the potential of the processor for single precision arithmetic. Experimental results on a GTX280 using CUBLAS 2.0, the implementation of BLAS for NVIDIA® GPUs with unified architecture, illustrate the performance of the different algorithms and techniques proposed.

Keywords:

Graphics processors (GPUs), general purpose computing on GPU, linear algebra, BLAS, high performance.

[1]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: `barrachi@icc.uji.es`.
[2]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: `castillo@icc.uji.es`.
[3]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: `figual@icc.uji.es`.
[4]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: `mayo@icc.uji.es`.
[5]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: `quintana@icc.uji.es`.
[6]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: `gquintan@icc.uji.es`.

Utilización de GPUs de última generación para cálculo matricial denso

Sergio Barrachina[7],
Maribel Castillo[8],
Francisco D. Igual[9],
Rafael Mayo[10],
Enrique S. Quintana-Ortí[11],
Gregorio Quintana-Ortí[12],

Resumen:

El presente informe describe diferentes algoritmos para calcular la solución de un sistema lineal sobre una GPU, así como técnicas generales para mejorar su renidimento, como *padding* y técnicas híbridas CPU-GPU. Además, se hace uso de técnicas de refinamiento iterativo con precisión mixta, para conseguir mayor precisión en la solución obtenida. Se incluyen resultados experimentales sobre el procesador GTX280 para simple y doble precisión, haciendo uso de CUBLAS 2.0, la implementación de BLAS desarrollada por NVIDIA® para GPUs con arquitectura unificada.

Palabras clave:

Procesadores gráficos (GPUs), procesamiento de carácter general sobre GPUs, álgebra lineal, BLAS, altas prestaciones.

[7]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: `barrachi@icc.uji.es`.
[8]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: `castillo@icc.uji.es`.
[9]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: `figual@icc.uji.es`.
[10]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: `mayo@icc.uji.es`.
[11]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: `quintana@icc.uji.es`.
[12]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: `gquintan@icc.uji.es`.

Exploiting the capabilities of modern GPUs for dense matrix computations

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

Depto. de Ingeniería y Ciencia de los Computadores,
Universidad Jaume I, 12.071–Castellón, Spain
{barrachi,castillo,figual,mayo,quintana}@icc.uji.es

Abstract

We present several algorithms to compute the solution of a linear system of equations on a GPU, as well as general techniques to improve their performance, such as padding and hybrid GPU-CPU computation. We compare single and double precision performance of a modern GPU with unified architecture, and show how iterative refinement with mixed precision can be used to regain full accuracy in the solution of linear systems, exploiting the potential of the processor for single precision arithmetic. Experimental results on a GTX280 using CUBLAS 2.0, the implementation of BLAS for NVIDIA® GPUs with unified architecture, illustrate the performance of the different algorithms and techniques proposed.

Keywords: Linear systems, Cholesky factorization, LU factorization, graphics processors (GPUs), dense linear algebra, high performance.

1 Introduction

The improvements in performance, functionality, and programmability of the current generation of graphics processors (GPUs) have renewed the interest in this class of hardware for general-purpose computations. These advances also apply to dense linear algebra, with important gains in the performance delivered for basic linear algebra operations. The interest in using GPUs for dense linear algebra is not new. Several earlier studies have evaluated the performance of this type of operations on former generations of GPUs. Some of them were specifically focused in the evaluation of different procedures for solving dense linear systems [1, 2].

In this paper we focus on the Cholesky and LU factorizations and update the studies in [1, 2], using the current generation of GPUs and the implementation of BLAS optimized for graphics processors with *unified architecture*. In particular, we compare several algorithmic variants of the factorization procedures and evaluate their performance on a GTX280 graphics processor. In addition, we describe techniques to improve the performance of the basic implementations and, as a result, we obtain optimized routines that outperform the CPU-based implementations in both single and double precision. Finally, we also employ an iterative method, which combines single and double precision arithmetic, to refine the solution of a linear system of equations to attain full precision accuracy.

By evaluating the double precision performance of the GTX280, we extend the study in [3], evaluating the loss of performance of the first generation of GPUs when operating in double precision. In addition, we can observe the real value of the iterative refinement technique exposed, by comparing a mixed precision algorithm and a full double precision implementation on GPU.

The new generation of GPUs, that exhibit a new unified architecture, solves many of the problems that limited the performance of older generations of graphics processors, mainly in terms of memory hierarchy, interconnection buses and programmability. In particular, CUDA has been released by NVIDIA as a general-purpose oriented API (application programming interface) for its graphics hardware, with the G80 and newer processors (for example, the GTX280 GPU) as the target platforms. In addition, CUBLAS is an optimized version of the BLAS built on top of CUDA, and adapted to the peculiarities of this type of platforms [4, 5].

The rest of the paper is structured as follows. Section 2 reviews the algorithms for the Cholesky and LU factorization implemented in our study. Section 3 describes several strategies that are applied to improve the performance of the initial algorithms. The impact of these techniques is evaluated in Section 4. Finally, Section 5 collects the conclusions of this analysis.

2 Overview of the Cholesky and LU factorization methods

Let $A \in \mathbb{R}^{n\times n}$ be symmetric positive definite, and consider its Cholesky factorization given by

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

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

There exist three different algorithmic variants for obtaining the Cholesky factorization [6]. Blocked algorithms for the different variants are given in Figure 1 in a notation that has been developed as part of the FLAME project [7, 8]. The thick lines in the figure denote how far the computation of the factorization has proceeded; the notation $\text{TRIL}\,(B)$ refers to the lower triangular part of matrix B, and $n(B)$ stands for the number of columns of B. We believe the rest of the notation to be intuitive. Upon completion, the entries of the Cholesky factor L overwrite the corresponding entries of A. Despite being different from the algorithmic point of view, all variants perform exactly the same operations. However, the performance of the implementations depends on the way and order in which these operations are executed, and also on the specific BLAS implementation that is employed.

Given a matrix $A \in \mathbb{R}^{m\times n}$, 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.

Three different variants for obtaining the LU factorization with partial pivoting are given in Figure 2 in FLAME notation. As for the Cholesky factorization, all variants perform the same operations, but in different order, and the triangular factors L and U overwrite the corresponding entries of A upon completion. The notation $\text{TRILU}(B)$ stands for the unit lower triangular matrix stored in B. The sequence of permutations is registered in vector $p \in \mathbb{R}^m$, which initially contains $\{1, 2, \ldots, m\}$. Given a vector $q \in \mathbb{R}^k$, $P(q) \in \mathbb{R}^{k\times k}$ denotes the corresponding permutation matrix.

For each variant shown in Figures 1 and 2, we also include the name of the BLAS-3 kernel used to carry out the corresponding operation. For the Cholesky factorization, the performance of the SYRK kernel, invoked to update A_{22}, will determine the final performance of Variant 1 of the blocked algorithm; the TRSM and SYRK kernels, used to update A_{10} and A_{11}, are the dominant operations for Variant 2; and the majority

```
Algorithm: $A := \text{CHOL\_BLK}(A)$
Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$
while $n(A_{TL}) < n(A)$ 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:
      $A_{11} := \text{CHOL\_UNB}(A_{11})$
      $A_{21} := A_{21}\text{TRIL}\,(A_{11})^{-T}$      (TRSM)
      $A_{22} := A_{22} - A_{21}A_{21}^{T}$          (SYRK)
    ----------------------------------------
    Variant 2:
      $A_{10} := A_{10}\text{TRIL}\,(A_{00})^{-T}$      (TRSM)
      $A_{11} := A_{11} - A_{10}A_{10}^{T}$          (SYRK)
      $A_{11} := \text{CHOL\_UNB}(A_{11})$
    ----------------------------------------
    Variant 3:
      $A_{11} := A_{11} - A_{10}A_{10}^{T}$          (SYRK)
      $A_{11} := \text{CHOL\_UNB}(A_{11})$
      $A_{21} := A_{21} - A_{20}A_{10}^{T}$          (GEMM)
      $A_{21} := A_{21}\text{TRIL}\,(A_{11})^{-T}$      (TRSM)
    ========================================
    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
```

Figure 1: Multiple blocked variants of the Cholesky factorization. CHOL_UNB refers to the unblocked versions of the Cholesky procedures.

```
Algorithm: $[A,p] := \text{LUP\_BLK}(A)$
Partition
  $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$
      where $A_{TL}$ is $0 \times 0$ and $p_T$ has 0 elements
while $n(A_{TL}) < n(A)$ do
    Determine block size $n_b$
    Repartition
        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$, $\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$
          where $A_{11}$ is $n_b \times n_b$ and $p_1$ has $n_b$ elements
    ============================================================
    Variant 1:
      $\left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline A_{21} \end{array}\right) := P(p_0) \left(\begin{array}{c} A_{01} \\ \hline A_{11} \\ \hline A_{21} \end{array}\right)$
      $A_{01} := \text{TRILU}(A_{00})^{-1} A_{01}$                 (TRSM)
      $A_{11} := A_{11} - A_{10} A_{01}$                           (GEMM)
      $A_{21} := A_{21} - A_{20} A_{01}$                           (GEMM)
      $\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LUP\_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$
      $\left(\begin{array}{c} A_{10} \\ \hline A_{20} \end{array}\right) := P(p_1) \left(\begin{array}{c} A_{10} \\ \hline A_{20} \end{array}\right)$
    ------------------------------------------------------------
    Variant 2:
      $A_{11} := A_{11} - A_{10} A_{01}$                           (GEMM)
      $A_{21} := A_{21} - A_{20} A_{01}$                           (GEMM)
      $\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LUP\_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$
      $\left(\begin{array}{c|c|c} A_{10} & & A_{12} \\ \hline A_{20} & & A_{22} \end{array}\right) := P(p_1) \left(\begin{array}{c|c|c} A_{10} & & A_{12} \\ \hline A_{20} & & A_{22} \end{array}\right)$
      $A_{12} := A_{12} - A_{10} A_{02}$                           (GEMM)
      $A_{12} := \text{TRILU}(A_{11})^{-1} A_{12}$                 (TRSM)
    ------------------------------------------------------------
    Variant 3:
      $\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := \text{LUP\_UNB}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$
      $\left(\begin{array}{c|c|c} A_{10} & & A_{12} \\ \hline A_{20} & & A_{22} \end{array}\right) := P(p_1) \left(\begin{array}{c|c|c} A_{10} & & A_{12} \\ \hline A_{20} & & A_{22} \end{array}\right)$
      $A_{12} := \text{TRILU}(A_{11})^{-1} A_{12}$                 (TRSM)
      $A_{22} := A_{22} - A_{21} A_{12}$                           (GEMM)
    ============================================================
    Continue with
        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$, $\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$
endwhile
```

Figure 2: Multiple blocked variants of the LU factorization with partial pivoting. LUP_UNB refer to the unblocked versions of the LU factorization procedures.

of the operations in Variant 3 are performed through the GEMM kernel when updating the submatrix A_{21}. As a result, the performance of these BLAS-3 kernels will determine which of the proposed variants of the Cholesky factorization yields a higher performance.

Similar considerations can be made for the study of the LU factorization variants described in Figure 2.

3 Computing the Cholesky and LU factorizations on GPUs

Starting from these basic implementations, the following sections introduce refinements that can be applied simultaneously in order to improve both the performance of the factorization process and the accuracy of the solution of the linear system. These improvements include padding, a hybrid CPU-GPU implementation, a recursive implementation, and an iterative refinement procedure.

3.1 Padding

Experiments in [9] have shown that Level 3 BLAS implementations of CUBLAS (specially the GEMM kernel) deliver much higher performance when operating on matrices with dimensions that are a multiple of 32. This is due to memory alignment issues [4].

Therefore, it is possible to improve the overall performance of the blocked Cholesky factorization (and, similarly, the LU factorization) process by applying the correct pad to the input matrix and selecting the appropriate block sizes. Starting from a block size n_b that is multiple of 32, we pad the $n \times n$ matrix A to compute the factorization

$$\bar{A} = \left(\begin{array}{cc} A & 0 \\ 0 & I_k \end{array} \right) = \left(\begin{array}{cc} L & 0 \\ 0 & I_k \end{array} \right) \left(\begin{array}{cc} L & 0 \\ 0 & I_k \end{array} \right)^T,$$

where I_k denotes the identity matrix of order k, and k is the difference between the matrix size n and the nearest integer multiple of n_b larger than n. By doing this, all BLAS-3 calls operate on submatrices of dimensions that are a multiple of 32, and the overall performance is improved. Moreover, there is no communication overhead associated with padding as only the matrix A and the resulting factor L are transferred between main memory and video memory. On the other hand, we incur in a computation overhead due to useless arithmetic operations which depends on the relation between n and 32. For moderate to large n, this overhead is negligible.

3.2 Hybrid algorithm

We have also developed a hybrid version of the blocked algorithm for the Cholesky and LU factorizations which delegates some of the calculations previously performed on the GPU to the CPU. This approach aims to exploit the different abilities of each processor to deal with specific operations. The advantage of the CPU is twofold: due to the stream-oriented nature of the GPU it offers higher performance when operating with small matrices, and it delivers higher performance for some fine-grained arithmetic operations, specially the square root calculation, heavily used in the factorization of the diagonal block A_{11}, for which the GPU is not fully optimized.

The hybrid algorithm sends the diagonal block from video memory to main memory, factorizes this block on the CPU, and transfers back the results to video memory before the computation on the GPU continues. Whether this technique delivers a performance gain will depend on the overhead introduced by the transference between video memory and main memory.

The same technique has been applied in the LU factorization. In this case, the factorization of the current column panel $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ is computed on the CPU.

3.3 Recursive implementation

It is quite straight-forward to obtain a recursive version of the blocked variants for the Cholesky factorization. The recursive version partitions the matrix into 2×2 square blocks, of similar dimensions, and then factorizes the upper-left block using the same algorithm, which results in a first level of recursion; the procedure is then repeated recursively at each deeper level.

We have implemented recursive implementations of Variants 1 and 2 for the Cholesky and LU factorizations, respectively, which perform a single level of recursion and employ the hybrid algorithm at the bottom stage. Performing several recursive steps did not improve the performance of the algorithm in our experiments.

3.4 Iterative refinement

GPUs have recently introduced double precision support, as mentioned in Section 1. Unfortunately, the double precision performance of the GTX280 is far from the peak performance of the processor when operating in single precision. However, computing the Cholesky or LU factorization on the GPU using single precision arithmetic will yield half the precision that is traditionally employed in numerical linear algebra.

The iterative refinement approach can be used to regain full (double-) precision when the factors obtained after the factorization process on the GPU are employed to solve the linear system $A \cdot x = b$, as described next.

This basic procedure for iterative refinement can be modified to use a mixed precision approach following the strategy in [10] for the Cell B.E. The factorization of matrix A is first computed on the GPU (in single precision arithmetic) using any of the algorithms proposed in previous sections. A first solution is then computed and iteratively refined on the CPU to double precision arithmetic; see Algorithm 3.4. In this algorithm, the `(32)` subscript indicates single precision storage, while the absence of subscript means double precision format. Thus, only the matrix-vector product $A \cdot x$ is performed in double precision (kernel GEMV), at a cost of $O(n^2)$ flops (floating-point arithmetic operations), while the rest of the nonnegligible arithmetic operations involve only single precision operands. The following algorithm illustrates the solution of a symmetric definite positive system using mixed precision with iterative refinement. The Cholesky factorization is performed on GPU. A similar strategy can be applied to general systems using the LU factorization.

Our implementation of the iterative refinement algorithm iterates until the solution, $x^{(i+1)}$, satisfies the following condition:

$$\frac{\|r^{(i)}\|}{\|x^{(i+1)}\|} < \sqrt{\varepsilon},$$

where ε corresponds to the machine precision of the platform. When this condition is met, the algorithm iterates twice more, and the solution is then considered to be accurate enough [10].

```
$A_{(32)}, b_{(32)} \leftarrow A, b$
$L_{(32)} \leftarrow$ 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
```

4 Experimental results

Starting from a basic blocked implementation, we show how the techniques proposed in the previous section (padding, hybrid approaches, and recursive implementation) improve the performance and accuracy of the GPU implementations.

4.1 Experimental setup

The system used for the performance evaluation is based on an AMD Phenom 9550 Quad-Core Processor running at 2.2 GHz, with 512 Kbytes of L2 cache memory per core, and 4 Gbytes of DDR2 RAM memory. The interconnection bus to the GPU is a PCIExpress Gen2 interface, with a peak bandwith of 8 Gbits/s.

On the GPU side, all the implementations have been tested on a NVIDIA Geforce GTX280 board, which implements *the unified architecture* introduced in the NVIDIA G80 series. This architecture is built on top of a unified shader or massively parallel processor, composed of hundreds of cores. Each core is a scalar, general purpose floating point processor operating in SIMD mode.

There are subtle differences between the G80 and the GT200 architecture implemented in the GTX280 GPU. As the G80, the GT200 is an array of SIMD processors, with up to 240 scalar streaming (or shader) cores grouped in clusters (also called *Streaming Multiprocessors*, SM). Figure 3 is a schematic diagram of the architecture. There are 8 *Streaming Processors* (SP) per multiprocessor, each one with single precision float point and 32-bit integer computation capabilities, and as a novelty, a single 64-bit ALU for FP64 support. All the Streaming Processors inside a Streaming Multiprocessor share a fast pool of shared memory of 16 Kbytes and a registry file of 16384 32-bit registers. In addition, every core has access to a global DDR video memory, with higher latency than the shared memory, but also of much higher capacity (1 Gbyte in the tested board).

Running at 1.296 Mhz, each SM can dual-issue 8 `madd+mul`[1] instructions per clock, achieving a peak performance of 933 GFLOPs per GPU when operating in single precision. Each FP64 ALU can issue one `madd` instruction per clock, with a total peak performance of nearly 78 double precision GFLOPS.

We have developed Fortran 77 implementations of the blocked factorization algorithms based on LAPACK codes, linked with the latest available CUDA and CUBLAS versions (2.0) for the GPU. In the CPU, the algorithms were implemented on top of GotoBLAS 1.26, tuned for the AMD architecture, using LAPACK 3.0 when necessary; LAPACK relies on the underlying BLAS implementation to extract all the performance of the QuadCore processor. The compilers include GNU Fortran Compiler version 4.1.2 and NVCC (NVIDIA CUDA compiler) release 1.1, version 0.2.1221. No extra optimization flags where used in our codes.

[1]The `madd` (multiply and accumulate) operation is prevalent in many graphics operations such as transformation and lighting, so GPUs are usually optimized for this type of operations. `mul` denote a multiplication of two scalars.

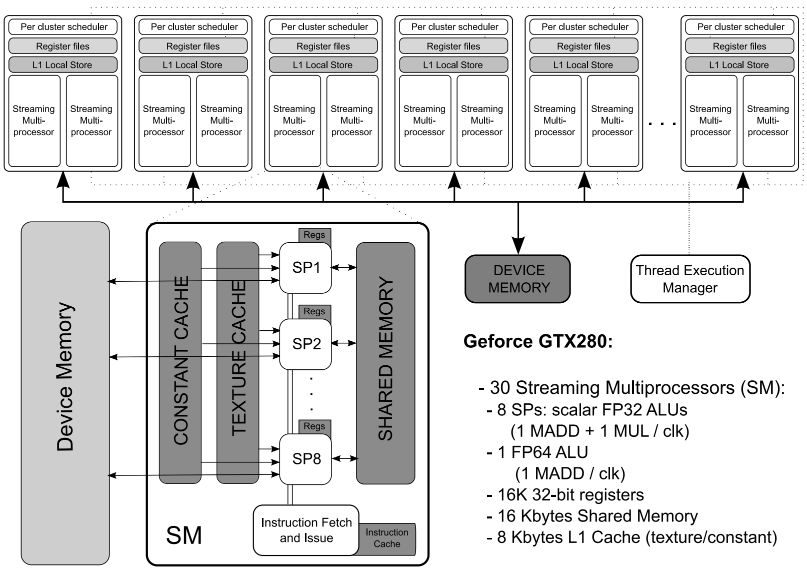

Figure 3: Architectural diagram of the GT200 architecture.

All the results on the GPU presented hereafter include the time required to transfer the data from the main memory to the GPU memory and retrieve the results back. The kernels for both the CPU and the GPU implementations operate on single and on double precision real data, and results are reported in terms of GFLOPS (10^9 flops per second). The four cores of the AMD processor were employed in the experiments in order to achieve a fair comparison between the CPU and the GPU. For the CPU, all parallelism was extracted within the multi-threaded implementation of BLAS.

4.2 Basic blocked implementations on CPU and GPU

The first set of experiments is based on the basic blocked implementations illustrated in Figures 1 and 2, executed on both CPU and GPU. Figures 4 and 5 report the performance of the three variants of the blocked algorithms for the Cholesky and LU factorizations, respectively. On the left-hand plot of the figures, we show the single precision performance of the implementations executed on CPU and GPU. On the right-hand plot, the double precision performance of both processors is shown.

On both the CPU and the GPU, the variants of the blocked algorithm deliver a considerable higher performance than their unblocked counterparts; therefore, results for the unblocked implementations are not included in the figures. Due to its stream-oriented architecture and the overhead introduced by the data transfers, the GPU only outperforms the CPU starting from matrices of large dimension (around $n = 6000$ for Cholesky, and $n = 3000$ for LU). For single precision data, these initial implementations on GPU obtain speed-ups of 2.66 and 2.95 for Cholesky and the LU, respectively, comparing the best variants on each platform. Although being more powerful in terms of peak performance, the SIMD stream-oriented architecture of current GPUs, and the big penalty introduced by the video memory latency still limits their actual performance when operating with small problem sizes, as can also be observed for the Level 3 BLAS implementations in CUBLAS [9].

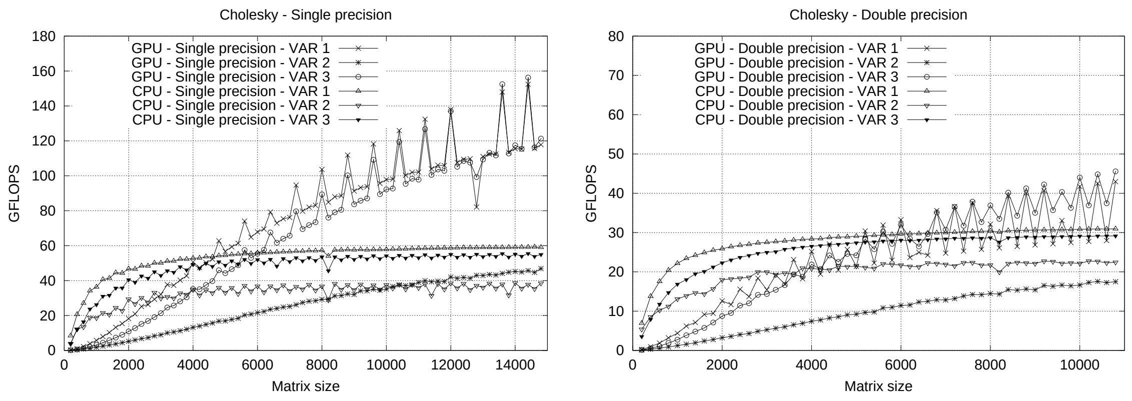

Figure 4: Performance of the three blocked variants for the Cholesky factorization, operating on single precision data (left) and double precision data (right). Highest performances attained on the GPU are 152.4, 46.8, and 156.2 GFLOPS for single precision, and 42.9, 17.5, and 45.6 GFLOPS for the double precision. Peak performances on CPU for the best variants are 59.6 GFLOPS for single precision and 30.9 GFLOPS for double precision.

The different variants of the blocked algorithm executed on GPU exhibit a much different performance. This can be explained by the different behavior of the underlying CUBLAS kernels, as we argue next. A detailed comparison between the Level 3 CUBLAS routines underlying the Cholesky and LU factorization routines (GEMM, TRSM, and SYRK) can be found in [9]. The results show that the GEMM kernel in CUBLAS is thoroughly tuned, while considerably less attention has been paid to the optimization of SYRK and TRSM. This explains the differences in the performance of the three variants of the Cholesky factorization. As noted in Figure 1, SYRK is the dominant operation in Variant 1; the bulk of the computation in Variant 2 is cast in terms of TRSM and SYRK; and the GEMM kernel is the most important in Variant 3.

Variant 1 of the LU factorization in Figure 4 obtains a poor performance compared with Variants 2 and 3. As explained before, the underlying BLAS implementation determines the final performance of the LU factorization process. The update of block A_{01} in this variant is implemented on top of the TRSM routine. Through a detailed performance evaluation of the CUBLAS TRSM routine, we have observed that this operation yields worse results when large triangular matrices are involved. The variant implemented suffers from this poor performance of the TRSM implementation of CUBLAS when updating matrices with $m \gg n$.

Similar conclusions can be extracted from the analysis of the double precision performance for both algorithms. However, in this case, the maximum speed-up compared with the multicore CPU is not as important as the attained for the single precision implementations. Nevertheless, the matrix sizes for which the GPU attain better performances than CPU are similar to those for single precision.

4.3 Blocked implementation with padding

Padding is a simple but effective method for improving the performance of the Level 3 CUBLAS implementations [9]. Our goal here is to exploit the high performance achieved by padding the Level 3 CUBLAS operations (see the difference between GEMM with and without padding in [9] for more details) to improve

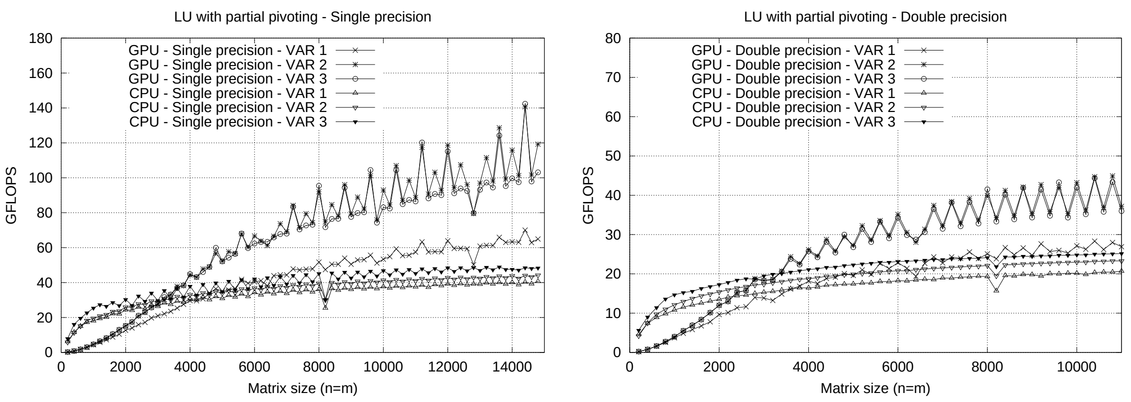

Figure 5: Performance of the three blocked variants for the LU factorization, operating on single precision data (left) and double precision data (right). Highest performances attained on the GPU are 64.9, 140.6, and 142.4 GFLOPS for single precision, and 26.9, 44.9, and 44.3 GFLOPS for double precision. Peak performances on CPU for the best variants are 48.3 GFLOPS for single precision and 25.1 GFLOPS for double precision.

the overall performance.

Figures 6 and 7 show the results of the three variants of the Cholesky and LU factorizations, respectively, when the appropriate padding is applied to the input matrices operating with single and double precision. Comparing the results with those without padding, it is possible to distinguish a small improvement in the final performance of the three variants of both factorizations for the single precision experiments. In [9] it was noted that the performance gain that is attained when applying padding to the Level 3 BLAS routines in CUBLAS is higher for GEMM than for SYRK. Thus, it is natural that Variant 3 of the Cholesky factorization (based on GEMM) benefits more than the other two variants. In fact, the improvement for Variant 2 is minimal when applying this optimization, as for Variant 1 of the LU factorization, in which TRSM is the main routine.

The application of padding masks the irregular behavior of the implementations, when the matrix size is not a multiple of 32 (see Figures 4 or 5 for $n = 12000$, for example). In addition, the overall performance is considerably improved: maximum speed-ups for the Cholesky factorization variants compared with the CPU implementations operating on single precision data are 2.59 and 2.89 for variants 1 and 3 (the second variant attains better performance on CPU), while the speed-ups attained for the LU are 1.63, 3.25, and 2.98, respectively.

This technique can be also applied with success to the double precision implementations, as can be observed on the right-hand plots of Figures 6 and 7. We achieved peak speed-ups of 1.12, 0.77 and 1.41 respectively for each variant of the Cholesky factorization, and 1.19, 1.66 and 1.57 respectively for the LU variants. More important than the peak performances is the constant behavior of the implementations for any matrix size.

4.4 Hybrid and recursive implementations

We next evaluate our hybrid and recursive blocked algorithms, including padding, for the Cholesky and LU factorizations based on Variants 1 and 2, respectively. We have chosen these variants because they have

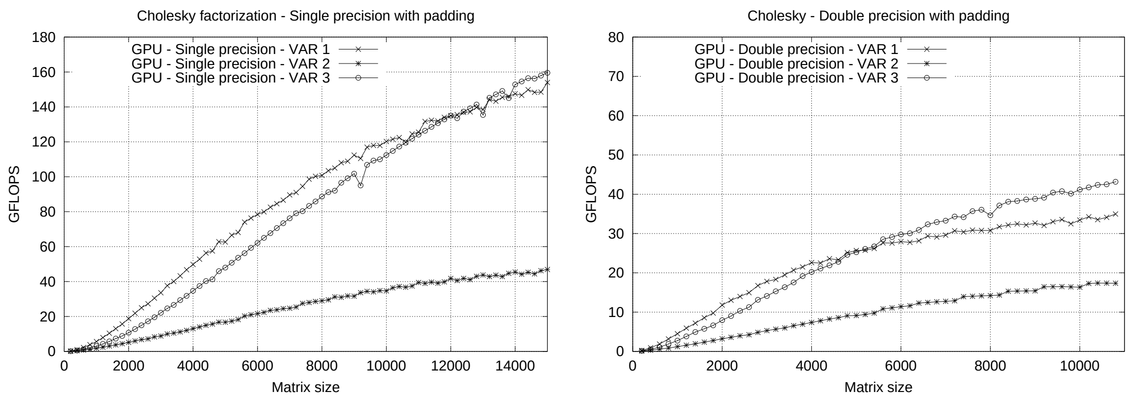

Figure 6: Performance of the three blocked variants for the Cholesky factorization with padding applied, operating on single precision data (left) and double precision data (right). Highest performances attained on the GPU are 154.1, 46.8, and 159.5 GFLOPS for single precision, and 16.2, 37.8, and 43.1 GFLOPS for double precision.

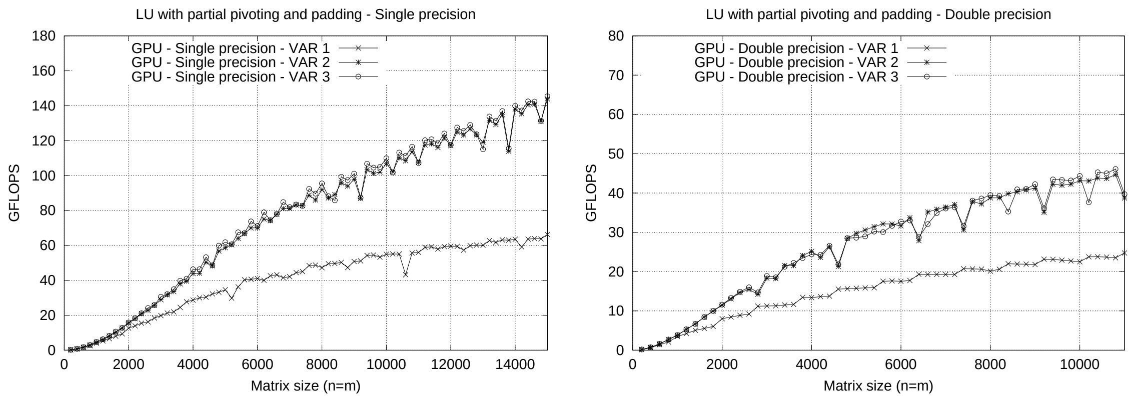

Figure 7: Performance of the three blocked variants for the LU factorization with padding applied, operating on single precision data (left) and double precision data (right). Highest performances attained on the GPU are 63.2, 145.4, and 143.9 GFLOPS for single precision, and 24.7, 44.7, and 46.1 GFLOPS for double precision.

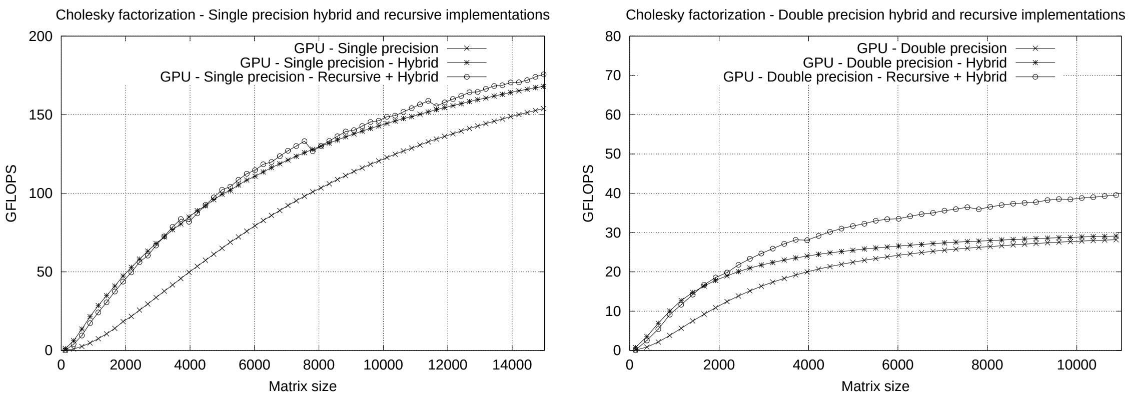

Figure 8: Left: performance of the implementations of Variant 1 of the blocked algorithm for the Cholesky factorization operating on single precision data: basic implementation, hybrid implementation, and a combination of the recursive and hybrid implementations. Highest performances are 153.9, 168, and 175.6 GFLOPS, respectively. Right: same implementations operating on double precision data. Peak performances are 28.1, 29.1, and 39.5 GFLOPS, respectively.

obtained the best results for each type of factorization. Figures 8 and 9 show that the hybrid approach delivers notable performance gains compared with the basic implementation for both algorithms. Recursion, however, is only positive when applied to the Cholesky factorization.

Due to the overhead associated with the factorization of the small current diagonal block/column panel on the GPU, the hybrid approach introduces a significant improvement compared with the basic implementation of both Cholesky/LU factorization processes. Similar benefits are to be expected for the other two variants. In addition, Figures 8 and 9 also show the improvement attained for a hybrid implementation combined with a recursive approach for the factorization process.

The combination of padding, hybrid execution and recursion improves the original single precision blocked implementation on GPU (see Section 4.2), achieving a maximum speed-up of 2.97 for the best Cholesky variant, and 3.04 for the chosen version of the LU factorization (version 2) when comparing the GPU implementations with the CPU ones. However, for the LU factorization, the highest speed-ups are not attained for the biggest input matrices. For double precision data, the overhead associated with the data transfers to/from main memory implies an important penalty on the final performance, making the improvements of the application of these techniques less important.

4.5 Iterative refinement

We next perform a time-based comparison using the basic implementation of Variant 1 for the blocked algorithms. Similar qualitative results can be extracted for the other variants of the algorithms. Employing the GPU as a general-purpose coprocessor, our mixed precision implementation first computes a solution using the Cholesky or LU factorization computed on the GPU (single precision), which is then refined to double precision accuracy. The overhead of the iterative refinement stage is reported in Figure 10 as the difference between the mixed and single precision implementations. The figure also includes the time for the corresponding full double precision routine executed exclusively on GPU.

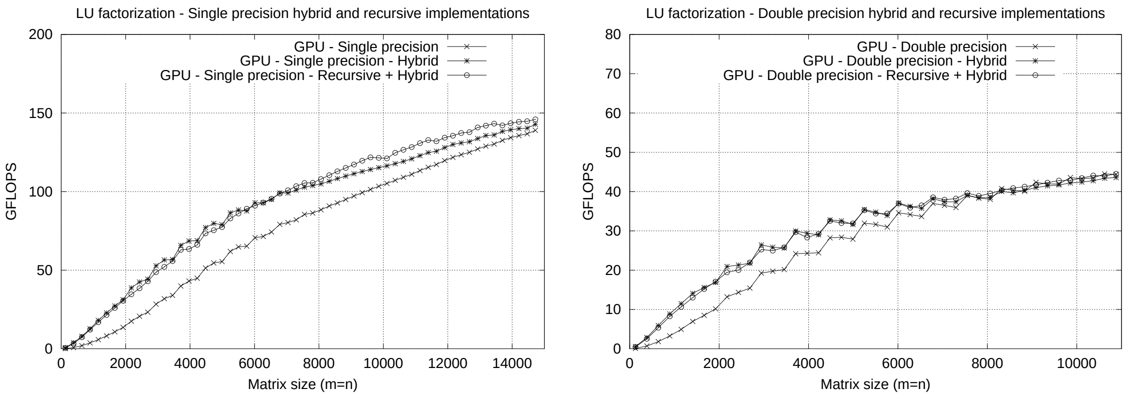

Figure 9: Left: performance of the implementations of Variant 2 of the blocked algorithm for the LU factorization operating on single precision data: basic implementation, hybrid implementation, and a combination of the recursive and hybrid implementations. Highest performances are 138.9, 142.8, and 145.9 GFLOPS, respectively. Right: same implementations operating on double precision data. Peak performances are 44.3, 43.5, and 44.5 GFLOPS, respectively.

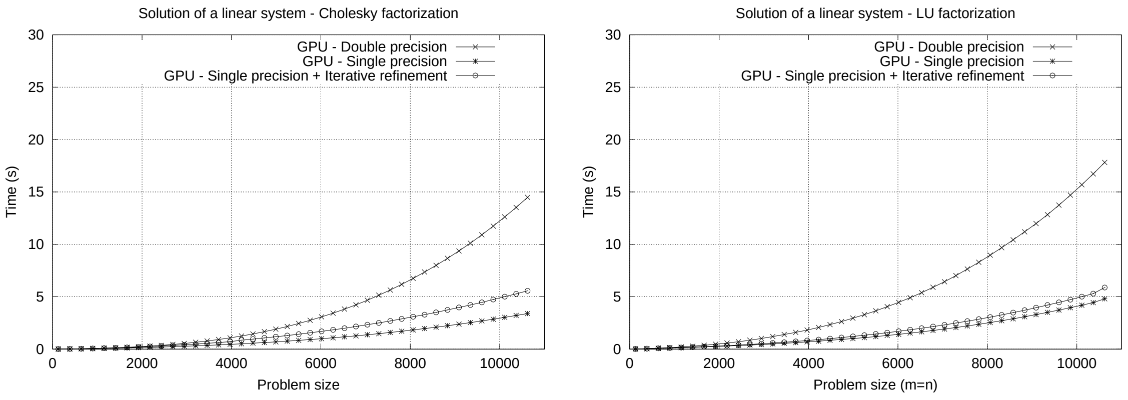

Figure 10: Execution time of mixed precision iterative refinement compared with those of a full single precision solution on the GPU and a full double precision solution on the GPU. A single right-hand side vector is considered.

Although the mixed precision version introduces some overhead, the execution time is much lower than that of a full double precision version executed exclusively on GPU. In fact, the number of iterations required to achieve the desired accuracy was lower than 6 in our experiments. Due to the higher performance of the single precision GPU implementations, the mixed precision strategy is a good choice to achieve accurate results in less time, as the overhead introduced by the refinement process does not have a significant impact on the overall performance.

4.6 Experimental results summary

The set of experiments performed for the Cholesky and LU with partial pivoting factorizations have showed that blocked implementations achieve much better performance results than the unblocked counterparts. However, there are still some points that can be tuned in the basic implementations.

Padding is an effective way to attain a regular behavior in the performance of both algorithms; although peak performances can be similar to the non-padded ones, higher performance can be achieved for all matrix sizes. Similar conclusions can be extracted for double precision data. Hybrid variants better exploit the potential of each type of processor achieving the biggest jump in performance, even more in combination with the recursive variant.

Double precision implementations also outperform the corresponding variants implemented on CPU, but achieving lower speed-ups. Techniques such as hybrid computation do not have the same impact on the final performance as in the case of single precision. Current GPUs are not fully prepared to beat top-end CPUs for double precision data and this type of algorithms. Fortunately, techniques such as iterative refinement can combine the power of the GPUs for single precision arithmetic and the accuracy of double precision implementations.

5 Conclusions

We have evaluated three blocked variants of the Cholesky and the LU factorizations using tuned implementations of BLAS on a GT200 graphics processor and an AMD QuadCore processor. The study reports that padding, hybrid GPU-CPU computation, and recursion are attractive techniques which deliver important increases in the performance of the implementations. These improvements can be applied to double precision implementations executed natively on GPU, with similar impact on the overall performance. Native support for double precision in the last generation of GPUs is still far from being impressive. Iterative refinement with mixed precision is thus revealed as an inexpensive technique to regain full accuracy in the solution of a linear system of equations, exploiting the high performance of the GPU when it operates on single precision data.

Similar results and techniques can be expected to apply also to other dense linear algebra factorization procedures, such as the QR factorization, attaining high performance and accuracy on a low cost and widely available hardware platform.

6 Acknowledgements

This research was supported by the CICYT project TIN2005-09037-C02-02 and FEDER, and project No. P1-1B2007-32 of the *Fundación Caixa-Castellón/Bancaixa* and UJI. We thank NVIDIA for the generous donation of the graphics hardware used in our evaluation.

References

[1] Galoppo, N., Govindaraju, N.K., Henson, M., Manocha, D.: LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In: SC ’05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, Washington, DC, USA, IEEE Computer Society (2005) 3

[2] Junk, J.H., O’Leary, D.P.: Cholesky decomposition and linear programming on a GPU. Master’s thesis, University of Maryland, College Park

[3] Barrachina, S., Castillo, M., Igual, F., Mayo, R., Quintana-Ortí, E.S.: Solving dense linear systems on graphics processors. In: Proceedings of Euro-Par 2008. Number 5168 in LNCS, Springer-Verlag Berlin Heidelberg (2008) 739–748

[4] NVIDIA: Nvidia CUDA Compute Unified Device Architecture. Programming Guide. NVIDIA (2007)

[5] NVIDIA: CUBLAS Library. NVIDIA (2007)

[6] Watkins, D.S.: Fundamentals of Matrix Computations. 2nd edn. John Wiley and Sons, inc., New York (2002)

[7] Gunnels, J.A., Gustavson, F.G., Henry, G.M., van de Geijn, R.A.: FLAME: Formal Linear Algebra Methods Environment. ACM Trans. Math. Soft. **27**(4) (December 2001) 422–455

[8] Bientinesi, P., Gunnels, J.A., Myers, M.E., Quintana-Ortí, E.S., van de Geijn, R.A.: The science of deriving dense linear algebra algorithms. ACM Trans. Math. Soft. **31**(1) (March 2005) 1–26

[9] Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S.: Evaluation and tuning of the level 3 CUBLAS for graphics processors. In: 9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing – PDSEC’08 (CD-ROM). (2008)

[10] Buttari, A., Dongarra, J., Langou, J., Langou, J., Luszczek, P., Kurzak, J.: Mixed precision iterative refinement techniques for the solution of dense linear systems. Int. J. High Perform. Comput. Appl. **21**(4) (2007) 457–466