Informe Técnico ICC 02-02-2008

Solving Dense Linear Systems on Graphics Processors

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

Febrero de 2008

Departamento de Ingeniería y Ciencia de Computadores

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

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

Solving Dense Linear Systems on Graphics Processors

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

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 also show how iterative refinement with mixed-precision can be used to regain full accuracy in the solution of linear systems. Experimental results on a G80 using CUBLAS, the implementation of BLAS for NVIDIA® GPUs with unified architecture, are given to 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.

Solución de Sistemas Lineales Densos sobre Procesadores Gráficos

Sergio Barrachina[6],
Maribel Castillo[7],
Francisco D. Igual[8],
Rafael Mayo[9],
Enrique S. Quintana-Ortí[10],

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 G80 haciendo uso de CUBLAS, 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.

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

Solving Dense Linear Systems on Graphics Processors*

Sergio Barrachina, Maribel Castillo, Francisco D. Igual, Rafael Mayo, and Enrique S. 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 also show how iterative refinement with mixed-precision can be used to regain full accuracy in the solution of linear systems. Experimental results on a G80 using CUBLAS, the implementation of BLAS for NVIDIA® GPUs with unified architecture, are given to 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 computation. 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 GPU. 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 G80 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. 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 achieve full precision accuracy.

* 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. Francisco Igual is supported by a research fellowship from the UJI (PREDOC/2006/02).

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 programmation abilities. In particular, CUDA has been released by NVIDIA as a general-purpose oriented API for its graphics hardware, with the G80 processor as target platform. 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 [3,4].

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 variants for obtaining the Cholesky factorization [5]. Blocked algorithms for the different variants are given in Figure 1 (left) in a notation that has been developed as part of the FLAME project [6,7]. The thick lines in the figure denote how far the computation of the factorization has proceeded; the notation 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 employed.

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.

Three different variants for obtaining the LU factorization with partial pivoting are given in Figure 1 (right) 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 TRILU(B) stands for the unit lower triangular matrix stored in B.

For each variant shown in Figure 1, 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

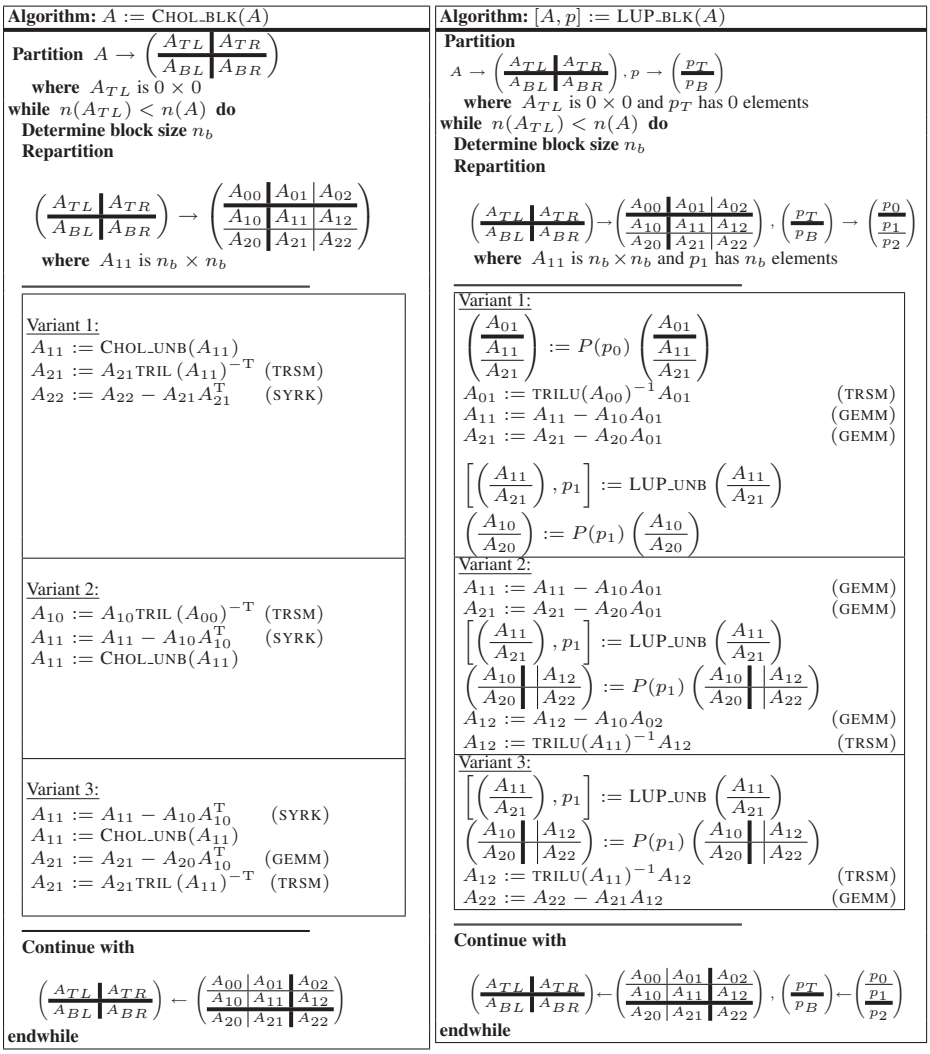

Figure 1. Multiple blocked variants of the Cholesky factorization (left) and the LU factorization with partial pivoting (right). CHOL_UNB and LUP_UNB refer to the unblocked versions of the Cholesky and the LU factorization procedures.

update A_{10} and A_{11}, are the dominant operations for Variant 2; and the majority 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 1 (right).

3 Computing the Cholesky and LU factorizations on GPUs

Starting from these basic implementations, the following subsections 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 [8] 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 [3].

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

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

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 transfered 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.

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 type of processor to deal with specific operations, and additionally, take profit from the higher performance of the CPU when operating with small matrices, see Figure 2. In particular, the factorization of the diagonal block A_{11} on the blocked algorithm (for the Cholesky factorization) carries out a series of fine-grained arithmetic operations, specially the square root calculation, that are not well suited for graphics processors.

The hybrid algorithm sends the diagonal block from video memory to main memory, factorizes this block on the CPU (where square root are easily computed, and the computation of small matrices is more efficient), 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

The G80 processor only provides single-precision arithmetic. Therefore, computing the Cholesky or LU factorization on the GPU will yield half of the precision that is usually employed in numerical linear algebra. However, iterative refinement 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 [9] 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 1.1. 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 arithmetic operations involve only single-precision operands.

Algorithm 1.1. Solution of a symmetric positive definite system using mixed precision with iterative refinement. The Cholesky factorization is computed on the GPU. A similar strategy can be applied to general systems using the LU factorization.

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

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

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

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 [9].

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 final performance and accuracy of the GPU implementations.

4.1 Experimental setup

The system used for the performance evaluation is based on an Intel Core2Duo CPU (codename Crusoe E6320) running at 1.86 GHz. On the GPU side, all the implementations have been tested on a Nvidia 8800 Ultra board, with a Nvidia G80 processor.

We have developed Fortran 77 implementations of the blocked factorization algorithms linked with CUDA 1.0 (with the same version of CUBLAS library) for the GPU. In the CPU, the algorithms were implemented on top of GotoBLAS version 1.19, using LAPACK version 3.0 when necessary. The compilers include GNU Fortran Compiler version 3.3.5 and NVCC (NVIDIA compiler) release 1.0, version 0.2.1221.

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 all operate on single-precision real data (except when iterative refinement is considered) and results are reported in terms of GFLOPS (10^9 flops per second). A single core of the Intel processor was employed in the experiments.

4.2 Basic blocked implementations on CPU and GPU

The first set of experiments are based on the basic blocked implementations illustrated in Figure 1, executed on both CPU and GPU. Figure 2 reports the performance of the three implemented variants of the blocked algorithms for the Cholesky and LU factorizations. These results are a first comparison between the CPU and the GPU implementations.

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, the GPU only outperforms the CPU starting from matrices of large dimension (around $n = 3000$ for Cholesky, and $n = 1500$ for LU). These initial implementations on GPU obtain speedups of 1.91 and 2.10 for Cholesky and the LU, respectively, comparing the best variants on each platform.

A detailed study of the results shows how 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 [8]. 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 fact helps to explain 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.

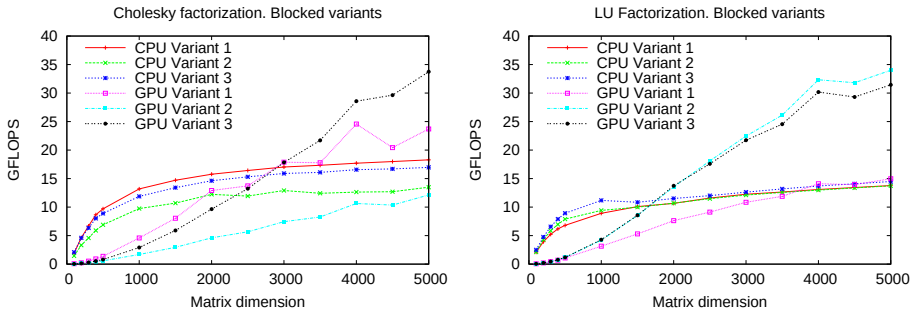

Figure 2. Performance of the three blocked variants for the Cholesky and LU factorization. Highest performances attained on the GPU are 23.7, 12.2, and 33.7 GFLOPS for Cholesky, and 14.9, 34, and 31.4 GFLOPS for the LU. Peak performances on CPU for the best variants are 17.6 GFLOPS for Cholesky and 14.9 GFLOPS for the LU.

Variant 1 of the LU factorization in Figure 2 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 the A_{01} block 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 $m \gg n$.

4.3 Blocked implementation with padding

Padding is a simple but effective method for improving the performance of the Level 3 CUBLAS implementations [8]. 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 [8] for more details) to improve the overall performance.

Figure 3 shows the results of the three variants of the Cholesky and LU factorizations when the appropriate padding is applied to the input matrices. 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. In [8] 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.

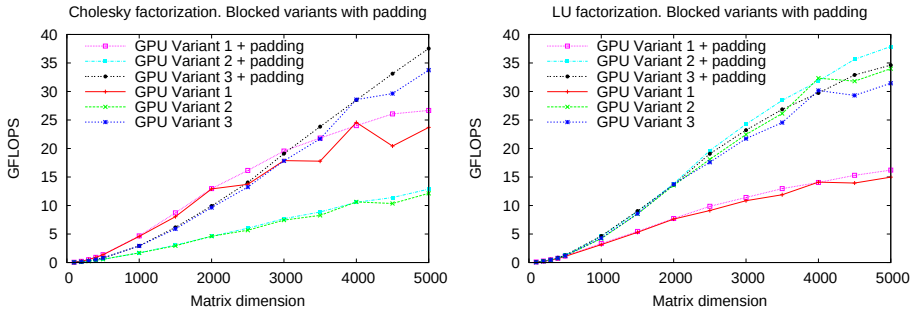

Figure 3. Performance of the three blocked variants for the Cholesky and LU factorization with padding applied. Highest performances attained on the GPU are 26.7, 12.9, and 37.6 GFLOPS for Cholesky, and 16.2, 37.8, and 34.6 GFLOPS for the LU. Comparing the best CPU and GPU implementations, the achieved speedup is 2.13 for Cholesky, and 2.32 for the LU.

The application of padding masks the irregular behavior of all the implementations for matrix sizes multiples of 32 (see Figure 2, for $m = 4000$). In addition, the overall performance is considerably improved: maximum speedups for the Cholesky factorization variants compared with the basic GPU implementations are 1.27, 1.10, and 1.12, while the speedups attained for the LU are 1.09, 1.12, and 1.12, respectively.

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 obtained the best results for each type of factorization. Figure 4 shows 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, not to the LU.

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 over to the basic implementation of both Cholesky/LU factorization processes. Similar benefits are to be expected for the other two variants. In addition, Figure 4 also shows the improvement attained for a hybrid implementation combined with a recursive approach for the factorization process. The combination of padding, hybrid execution

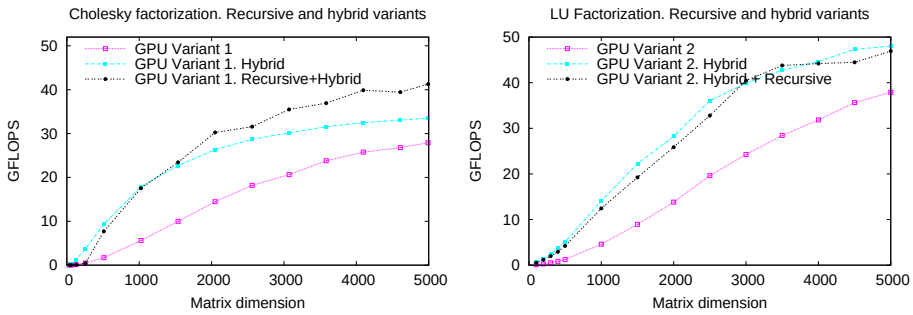

Figure 4. Left: performance of the implementations of Variant 1 of the blocked algorithm for the Cholesky factorization: basic implementation, hybrid implementation, and a combination of the recursive and hybrid implementations. Highest performances are 27.9, 33.5, and 41.2 GFLOPS, respectively. Right: same implementations for the Variant 2 of the blocked algorithm for the LU factorization. Peak performances are 37.8, 47.9, and 46.9 GFLOPS, respectively.

and recursion has improved the original blocked implementation on GPU (see Section 4.2), achieving a maximum speedup of 2.34 for the best Cholesky variant, and 3.14 for the LU when comparing the GPU implementations with the CPU ones.

4.5 Iterative refinement

We next perform a time-based comparison using the basic implementation of Variant 1 for the blocked algorithms. Using 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 5 as the difference between the mixed and single-precision implementations. The figure also includes the time for the corresponding full double-precision routine in LAPACK, executed exclusively on CPU.

Although the mixed-precision version introduces some overhead, the execution time is much lower than that of a full double-precision version executed on the CPU. 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 GPU implementations, the mixed-precision strategy is a good choice to achieve accureate results in less time, as the overhead introduced by the refinement process does not have a significant impact on the overall performance.

5 Conclusions

We have evaluated three blocked variants of the Cholesky and the LU factorizations using highly tuned implementations of BLAS on a G80 graphics processor and an Intel processor. The study reports that padding, hybrid GPU-CPU computation, and re-

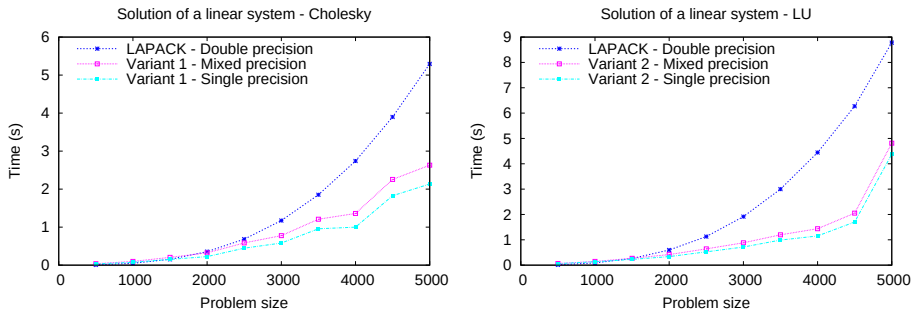

Figure 5. Execution time of mixed-precision iterative refinement compared with those of a full single-precision solution on the GPU and a full double-precision solution on the CPU. A single right-hand side vector is considered.

cursion are simple but attractive techniques which deliver important increases in the performance of the implementations.

Furthermore, iterative refinement with mixed precision is revealed as an inexpensive technique to regain full accuracy in the solution of a linear system of equations. In addition, similar results and techniques (padding, hybrid CPU-GPU computation, recursion and iterative refinement) 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.

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. NVIDIA: Nvidia CUDA Compute Unified Device Architecture. Programming Guide. (2007)
4. NVIDIA: CUBLAS Library. (2007)
5. Watkins, D.S.: Fundamentals of Matrix Computations. 2nd edn. John Wiley and Sons, inc., New York (2002)
6. 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
7. 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
8. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S.: Evaluation and tuning of the level 3 CUBLAS for graphics processors. PDSEC08 (2008. To appear)
9. 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