Parallel Computing 36 (2010) 232–240

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Towards dense linear algebra for hybrid GPU accelerated manycore systems

Stanimire Tomov [a,*], Jack Dongarra [a,b,c], Marc Baboulin [a,d]

[a] University of Tennessee, Department of Electrical Engineering and Computer Science, 1122 Volunteer Blvd, Knoxville, TN 37996-3450, USA
[b] Oak Ridge National Laboratory, USA
[c] University of Manchester, UK
[d] University of Coimbra, Department of Mathematics, 3001-454 Coimbra, Portugal

ARTICLE INFO

Article history:
Received 14 October 2008
Received in revised form 25 June 2009
Accepted 7 December 2009
Available online 28 December 2009

Keywords:
Hybrid computing
Dense linear algebra
Parallel algorithms
Multicore processors
Graphics processing units

ABSTRACT

We highlight the trends leading to the increased appeal of using hybrid multicore + GPU systems for high performance computing. We present a set of techniques that can be used to develop efficient dense linear algebra algorithms for these systems. We illustrate the main ideas with the development of a hybrid LU factorization algorithm where we split the computation over a multicore and a graphics processor, and use particular techniques to reduce the amount of pivoting and communication between the hybrid components. This results in an efficient algorithm with balanced use of a multicore processor and a graphics processor.

1. Introduction

Computing technology is currently undergoing a transition driven by power and performance limitations that provide more and more on-die x86 cores each year. The current standard is *homogeneous* quad-core chips and the development road map indicates that 8, 16, and 32 core chips will follow in the coming years. There is widespread recognition that performance improvement on CPU-based systems in the near future will come from the use of multicore platforms. On the other hand, homogeneous x86-based multicore machines are not the only proposed way forward. IBM, for example, introduced its own *heterogeneous* multicore architecture, the CELL Broadband Engine. Other innovative solutions include GPUs, FPGAs, and ASICs. GPUs stand out in a unique way from all these innovative solutions because they are produced as commodity processors and their floating point performance has significantly outpaced that of CPUs in recent years (see Fig. 1). Moreover, GPUs have become easier to program, which allows developers to effectively exploit their computational power [38]. Currently, major chip manufacturers are developing next-generation microprocessor designs that integrate multicore CPU and GPU components [18,31].

The problems and challenges for developers in this new and quickly changing computational landscape are daunting. Many familiar and widely used algorithms for prior sequential CPUs need to be rethought and rewritten to take advantage of the new, highly parallel heterogeneous CPUs [5,8]. In many cases the new algorithmic designs may well be hybrid, mapping algorithmic requirements to the strengths of each component of the heterogeneous CPU. The ultimate goal of our work in this area is to develop a dense linear algebra (DLA) library similar to LAPACK [1] but for heterogeneous CPUs, starting with current multicore + GPUs systems. The underlying design philosophy consists of representing algorithms as a collection of BLAS-based tasks that are executed over the multicore and the GPU. This abstracts us from the specificities in programming

* Corresponding author. Tel.: +1 865 974 8295; fax: +1 865 974 8296.
E-mail addresses: tomov@eecs.utk.edu (S. Tomov), dongarra@eecs.utk.edu (J. Dongarra), baboulin@mat.uc.pt (M. Baboulin).

0167-8191/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2009.12.005

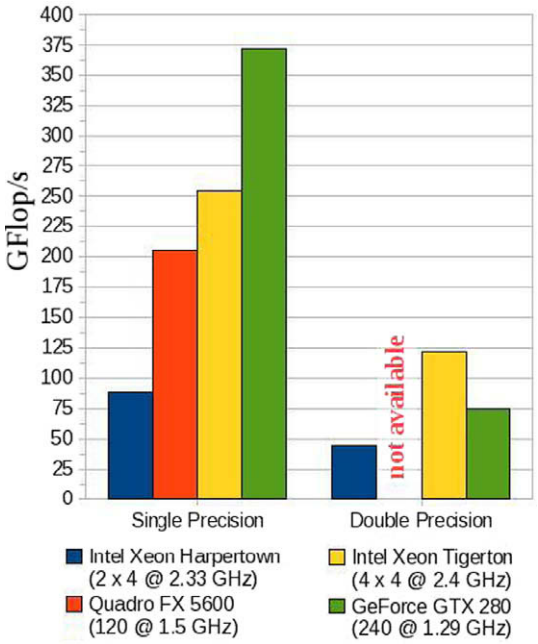

Fig. 1. Peak measured performances of matrix–matrix multiplications on current multicore (from Intel) and GPU (from NVIDIA) architectures.

a GPU. The approach presented in this paper relies on high performance BLAS implementations, which are available for current multicores and GPUs (see Fig. 1). The main difference from previous work in the field [2,4,13,34,36] is that our approach allows us to use the multicore hosting the GPU more efficiently, resulting in algorithms of higher performance.

This paper is organized as follows. Section 2 gives an overview of the use of GPUs for HPC and DLA in particular, along with highlights on GPU's evolution and future trends. Section 3 describes our approach for combining GPUs and multicores in DLA algorithms. Section 4 presents a specific example, namely a hybrid LU factorization, which further illustrates the concept of hybrid multicore + GPU computing for DLA. Finally, Section 5 gives conclusions and future directions.

2. GPUs for HPC

Driven by the demands of the gaming industry, graphics hardware has substantially evolved over the years to include more functionality and programmability. This, combined with the impressive floating point performance of the graphics cards, has enabled and motivated their use in applications well beyond graphics. In this section, we give an overview of the GPU's evolution over the years and its use in the area of dense linear algebra.

2.1. GPU evolution and future trends

The gaming industry and the large market that it enjoys have pushed GPUs over the years to excell in graphics rendering. There are three computational characteristics of graphics rendering [24]. Namely, it

1. Requires enormous computational power,
2. Allows for high parallelism, and
3. Stresses more on high bandwidth than low latency, as latency requirements can be compensated for by the deep graphics pipeline.

This pattern of computation is common with many other applications. Therefore, GPUs have benefited a large number of applications, turning them into General Purpose GPUs (GPGPUs), as often referred to in the literature.

The need for additional computational power, accuracy, and ability to implement complex simulations has pushed the GPUs development for higher speed, higher precision arithmetic, and more programmability. Currently, GPUs have reached a theoretical peak performance of 1 TFlop/s in single precision, support fully the IEEE double precision arithmetic standard [22], and have a programming model (e.g. see CUDA [23]) that may revive the prospects of getting high performance for little coding efforts [16]. CUDA, as an architecture and programming language, is not only easy to use but has also added and exposed to the user more hardware resources than what other languages and previous generation cards have offered. For example, CUDA extends the previous vision that GPUs are going to evolve towards more powerful stream processors [28],

by providing not only the data parallelism inherent for stream processors, but also multi-threading parallelism. CUDA also provides multiple levels of memory hierarchy, support for pointers, asynchronicity, etc [23]. These features have cemented even further the important role of GPUs in today's general purpose HPC [25,37,38]. With the introduction of CUDA, software developers do not have to know about graphics in order to use GPUs for general purpose computing. As CUDA numerical libraries become rapidly available, users may not even have to learn CUDA to benefit from the GPUs.

Over the years GPUs have moved "closer" to CPUs in terms of functionality and programmability. CPUs have also acquired functionalities similar to that of GPUs. For example, Intel's SSE and PowerPC's AltiVec instructions offer a vector programming model similar to GPUs' (see the argument in [35] that modern GPUs should be viewed as multi-threaded multicore vector units). Moreover, there are the *AMD Fusion* plans [18] to integrate a CPU and GPU on a single chip, and other hybrids between multicore x86 and a GPU, e.g. Intel's Larrabee system [31]. These trends make it more evident that future architectures will be hybrid and will rely on the integration (in varying proportions) of homogeneous multicore and GPU type of components.

2.2. GPUs for DLA

Due to the high ratio of floating point calculations to data required, many DLA algorithms have been of very high performance (e.g., close to the machine's peak) on standard CPU architectures. Older generation GPUs did not have memory hierarchy and their performance exclusively relied on high bandwidth. Therefore, although there has been some work in the field, the use of older GPUs has not led to significantly accelerated DLA. For example Fatahalian et al. [12] studied SGEMM and their conclusion was that CPU implementations outperform most GPU implementations. Only the ATI X800XT produced comparable results (close to 12 GFlop/s) with a 3 GHz Pentium 4. Similar results were produced by Galoppo et al. [14] on LU factorization. Their results were, in general, outperformed by LAPACK routines using ATLAS on 3.4 GHz Pentium IV. Their best result on LU with partial pivoting was approximately 5.7 GFlop/s on an NVIDIA 7800.

But this lack of acceleration has recently changed due to a combination of factors. First, GPUs have exponentially outpaced CPUs in performance. GPUs have been approximately doubling their performance every year vs. every year and a half for CPUs. Second, GPUs have significantly outpaced CPUs in bandwidth/throughput, e.g., the GTX 280 has bandwidth of 141.7 GB/s, which is about an order of magnitude higher than current multi-socket multicore systems. Finally, by having memory hierarchy the GPUs can be programmed for memory reuse and hence not rely exclusively on high bandwidth in order to achieve a high percentage of their theoretical performance peak. A simple illustration about the impact on performance of this combination of factors can be seen in Fig. 1 where we give the matrix-matrix multiplication performance of two modern multicore processors and two GPUs. Note that in single precision, the GTX 280 is about 115 GFlop/s faster than the quad-socket, quad-core Intel Xeon Tigerton (cores running at 2.4 GHz), and in double precision about 30 GFlop/s faster than the dual-socket, quad-core Intel Xeon Harpertown (cores running 2.33 GHz).

The first CUDA GPU results that significantly outperformed standard CPUs on single precision DLA started appearing at the beginning of 2008. To mention a few, in January, a poster by Volkov and Demmel [36] described an SGEMM kernel that significantly outperformed the one released by NVIDIA in the CUBLAS library (125 GFlop/s in CUBLAS vs. more than 180 GFlop/s in their implementation in single precision). Moreover, the improved kernels were used in developing an LU factorization running at up to 140 GFlop/s. In March, Tomov [32] presented at PPSC08 a Cholesky factorization running at up to 160 GFlop/s in single precision using Volkov's sgemm kernel (later described in LAPACK Working Note 200 [2]). In May, Volkov and Demmel [35] described LU, QR, and Cholesky factorizations running at up to 180 GFlop/s in single precision (with QR a little bit more). The first results on a pre-released next-generation G90 NVIDIA card were presented at UGC2008 in May, where Dongarra et al. [11] reported Cholesky running at up to 327 GFlop/s in single precision. Using again the newest generation card, in this paper, we describe an LU algorithm running at up to 388 GFlop/s in single precision and 99.4 Gflop/s in double precision.

The first results just described form the general understanding on how to program DLA using CUDA. Namely, there are three main ideas that define the approach:

1. Use BLAS-level parallelism, where the matrix resides on the GPU, and the CPU is running, for example, LAPACK-style code, e.g., represented as a sequence of CUBLAS kernels, using the GPU pointer arithmetic,
2. Offload to the CPU small kernels that are inefficient for the GPU,
3. Use asynchronicity between CPU and GPU whenever possible in the offload/load process.

This is illustrated for Cholesky factorization (so called left-looking version) in Fig. 2 (the case reported in [11]). The matrix to be factorized is allocated on the GPU memory and the code is as in LAPACK with BLAS calls replaced by CUBLAS, which represents the first idea from the list above. As steps two and three of the algorithm are independent, and the Cholesky factorization of matrix B (notations as on Fig. 2) would have been inefficient for the GPU (small problem of size 128×128 for example, i.e. cannot have enough parallelism to utilize 240 cores GPU), B is offloaded and factorized on the CPU, which illustrates the second idea. Finally, steps 2 and 3 of the algorithm are done in parallel as calls to CUDA are asynchronous, namely as the CPU calls cublasSgemm the execution continues (i.e., to SPOTRF, without waiting for the completion of cublasSgemm, which illustrates the third idea). In addition to overlapping just the computation, for cards that support it, sending B to the

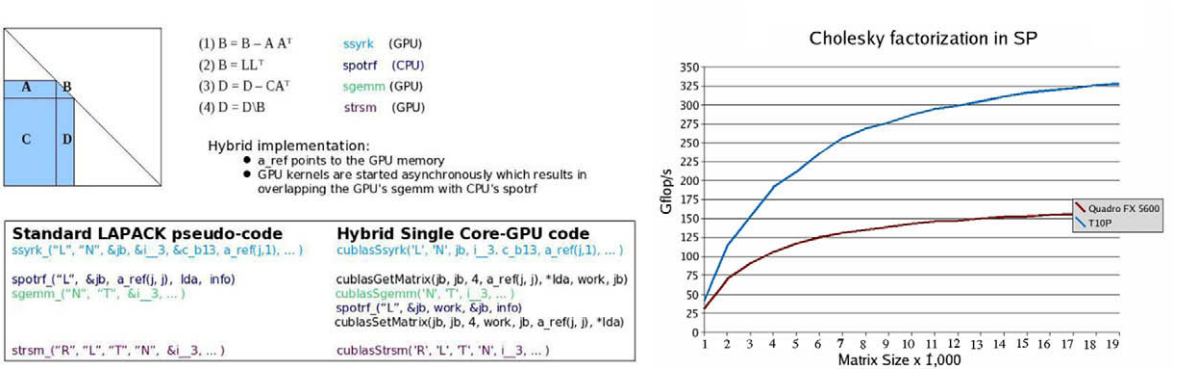

Fig. 2. Left looking Cholesky factorization: implementation (Left) and performance running the algorithm on an NVIDIA T10P and NVIDIA Quadro FX 5600 (Right) in single precision arithmetic.

CPU and moving the result back could be overlapped with the GPU computation (of cublasSgemm in this case) when asynchronous copy calls are used.

3. DLA for hybrid multicore + GPU systems

3.1. Design philosophy

Whether designing DLA algorithms for multicores or GPUs, the requirements for efficient execution are the same, namely algorithms should be of *high parallelism* and *reduced communication* to mask slow memory speeds. When we combine the two architectures, algorithms should also be properly *hybridized*. This means that the work load should be balanced throughout the execution, and the work scheduling/mapping should ensure matching of architectural strengths to algorithmic requirements.

Elements of hybridization can be seen in previous work on DLA for GPUs. Namely, while developing LAPACK-style one-sided matrix factorizations for GPUs, several groups [2,4,36] observed that the panel factorizations are often faster on the CPU than on the GPU (the approach described in Section 2.2), which led to the development of highly efficient factorizations [11,29,34] where a single CPU core is used to control the GPU and to factor the panels. In contrast to this previous work, the importance here is on developing algorithms that will efficiently use both a multicore and a GPU. We note that the early work did not use available cores of the multicore hosting the GPU. In the case of LU and QR factorizations, only one core is used to factor the panel and, because the panel factorization is bandwidth limited, the core used is able to largely saturate the bus of a multi-socket multicore system. This is important because it shows that modifications to those algorithms by just scheduling work to the available multicore would not be the optimal solution [7,30].

Another approach is to hybridize BLAS and leave the LAPACK code almost untouched. Using this approach, Fatica [13] developed hybrid DGEMM and DTRSM for multicore + GPU (and GPU-enhanced clusters) and used them to accelerate the LINPACK benchmark. BLAS level parallelism has limitations for multicore use [7] as well as hybrid systems (experiments are showing BLAS-level parallelism being up to 2–3× slower).

Our approach – based on hybridization of LAPACK – extends the previous work to overcome the shortcomings related to multicore use. We describe it in Section 3.2. We also address the issue of designing algorithms recently referred to as communication reducing/minimizing/avoiding/optimal algorithms [3,9,15]. This is a difficult problem and is a subject of current research in the field of DLA. A classic example is the transition from algorithms based on optimized Level 1 BLAS (from the LINPACK and EISPACK libraries) to algorithms that use block matrix operations in their innermost loops, which actually formed LAPACK's design philosophy. Current examples include work on LU and QR factorizations, in particular in the so called *tiled* [7] and *communication avoiding* [9,15] algorithms. We developed a hybrid LU algorithm, described in Section 4, that is yet another example of a communication-optimal algorithm.

3.2. Hybridization of LAPACK

Our design philosophy is the hybridization of LAPACK:

- Represent LAPACK algorithms as a collection of BLAS-based *tasks* and *dependencies* among them (described in Section 3.2.1),
- Properly *schedule* the (BLAS-based) tasks execution over the multicore and the GPU (described in Section 3.2.2).

This approach allows us to reuse parts of LAPACK in a systematic way, and moreover, abstracts us from the specificities in programming a GPU. We have applied it successfully for both one and two-sided matrix factorizations.

3.2.1. Task splitting

A fundamental concept in programming current parallel architectures is the flexible control over the data and execution flow. Algorithms and their execution flows can be represented as Directed Acyclic Graphs (DAGs), where nodes represent the tasks and the edges the dependencies among them. Fig. 3 gives a typical example of how a DLA algorithm may look like when represented as a DAG. The nodes in red in this case represent the sequential part of the algorithm and the ones in green the tasks that can be performed in parallel. Ideally the scheduling should be asynchronous and dynamic, so that the tasks in red are overlapped with the tasks in green, without violating any dependency. This can be done by defining a "critical path", which is the most time-consuming sequence of basic operations that must be carried out sequentially even allowing for all possible parallelism and scheduling for execution the tasks from the critical path as soon as possible (i.e., when all dependencies have been computed).

The concept of representing algorithms and their execution flows as DAGs is used in the context of developing DLA for multicores [5]. Similarly to multicore, we can apply the DAGs concept to the hybrid case. One of the differences is the task granularity. For multicores small tasks work well [7]. For GPUs a task would be one GPU kernel invocation. Therefore, in order to efficiently execute it, e.g. on the 240 processing elements of the GTX 280 GPU, we need the task to be larger than in the multicore case (as shown in Fig. 3, right). Tasks from the critical path can be smaller and in general executed on the GPU's host.

Note that splitting LAPACK code into tasks is easy, because one has to only split the underlying BLAS calls. A challenge is what to be the granularity of the tasks. A solution that we employ in our codes it to parameterize the granularity and tune it empirically [21].

3.2.2. Task scheduling

The tasks scheduling is of crucial importance for the efficient execution of an algorithm. Proper scheduling, for example scheduling tasks from the critical path to be executed as soon as possible, results in techniques that have been used in the past. In particular these are the "look-ahead" techniques that have been extensively applied to the LU factorization. Such methods can be used to remedy the problem of synchronizations introduced by non-parallelizable tasks by overlapping their execution with the execution of more efficient ones [10]. It has been applied also in the context of GPUs in [35] as well as here. See Fig. 3, right, where if we overlap the execution of the 2 circled tasks on the critical path (by the host), with the execution of the green tasks circled and marked GPU (by the GPU), we get a hybrid version of the look-ahead technique.

Another aspect of the scheduling, besides the order mentioned above, is where to schedule tasks – on the multicore or the GPU. Currently we do this type of scheduling statically. We use the multicore for tasks on the critical path because they are in general small and may have conditional statements. The LU example below demonstrates this for the one-sided factorizations. The hybrid Hessenberg reduction algorithm [33] demonstrates this for the two-sided factorizations. The case of Hessenberg reduction is particularly interesting because it cannot be accelerated on multicore alone. Using multicore + GPU though, and applying the approach descried here, we can achieve up to 16× performance acceleration compared to just its multicore performance [33]. The panel factorization for the Hessenberg reduction is mostly scheduled on the multicore except for a large Level 2 BLAS part of it, which is bandwidth limited and therefore is scheduled on the GPU to use its high bandwidth.

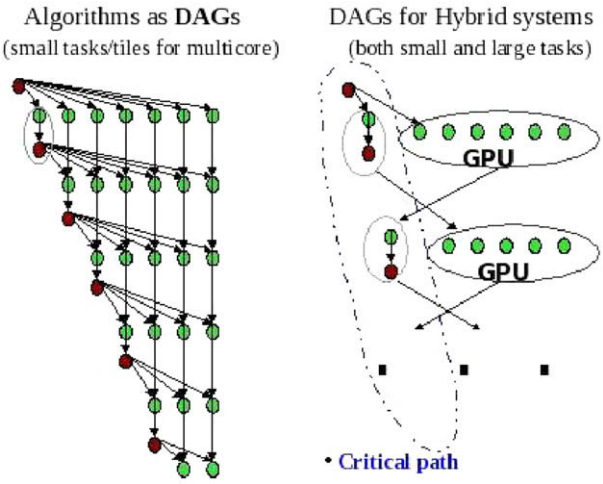

Fig. 3. Algorithms as DAGs.

4. An example of DLA algorithms for hybrid systems

To further motivate and illustrate the main ideas on the hybrid multicore + GPU approach for DLA, we give an example of an algorithm for hybrid systems along with its performance results.

4.1. An LU factorization design for hybrid multicore + GPU systems

We consider a right looking block LU factorization and design an algorithm for hybrid multicore + GPU systems. The approach is based on splitting the computation as shown in Fig. 4. The numbers are under the assumption that the host has 8 cores, which is the case for our test system. Having an N×N matrix, the splitting is such that the first N – 7 NB columns reside on the GPU memory and the last 7 NB on the host where NB is the algorithm's block size [1]. The value of NB is determined empirically. The tuned NB values corresponding to different matrix sizes N are given in Section 4.3.

A description of the hybrid algorithm is given as follows:

1. Current panel is downloaded to the CPU. For example the dark blue part of the matrix of NB columns is the panel for the first iteration.
2. The panel is factored on the CPU and the result is sent back to the GPU to update the trailing sub-matrix (colored in red for the first iteration).
3. The GPU updates the first NB columns (next panel) of the trailing matrix, the updated panel is sent to the CPU and asynchronously factored on the CPU while the GPU updates the rest of the matrix (note that this is the look-ahead technique described in Section 3.2.2).
4. The rest of the host cores (7 in this case) update the last 7 NB columns (one core per block of NB columns).
5. There is a synchronization between the 7 cores and the 1 core + 1 GPU after the LU factorization of the sub-matrix colored in red (and the corresponding updates of the trailing matrix). At this point, based on N and NB, we determine empirically what hardware to use in factoring the trailing 7 NB × 7 NB matrix. Choices are multicore + GPU (with block size tuned for matrix of size 7 NB), 1 core + GPU, or multicore.

This algorithm is general enough to be applicable to many forms of LU factorizations, where the distinction can be made based on the form of pivoting that they employ.

4.2. The issue of pivoting in LU factorizations

Pivoting is a well-known technique to ensure stability in matrix algorithms. In particular, the commonly used method of Gaussian elimination (GE) with partial pivoting (PP) is implemented in current linear algebra libraries for solving square linear systems $Ax = b$, resulting in reliable algorithms. In the LAPACK [1] implementation of GE, rows are swapped at once during pivoting, which inhibits the exploitation of more asynchronicity between block operations.

Reference [15] describes a pivoting strategy that minimizes the number of messages exchanged during the panel factorization and demonstrates that this approach is stable in practice. For multicore, pairwise pivoting (PwP) is often considered

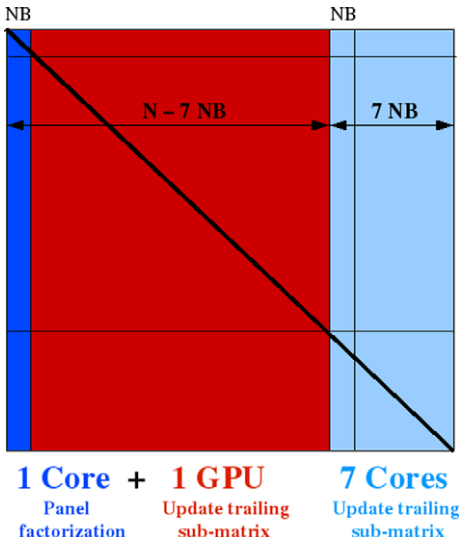

Fig. 4. Load splitting for a hybrid LU factorization.

(e.g., in [7]) but this generates a significant overhead since the rows are swapped in pairs of blocks. Still for multi-threaded architectures, [30] describes an algorithm by blocks for LU factorization that uses a pivoting technique referred to as incremental pivoting based on principles used for out-of-core solvers.

For implementation of PP LU on GPUs, [35] designs an algorithm using innovative data structures where storing the matrix in row-major layout helps in reducing the pivoting overhead from 56% of the total factorization time to 1–10% (depending on the machine and on the problem size).

In the following, we show the performance of our hybrid design using an extension of the technique proposed in [2]. Briefly, the approach in [2] follows the idea of [26,27] to transform the original matrix into a matrix that would be sufficiently "random" so that, with probability close to 1, pivoting is not needed. These transformations are, in general, chosen as unitary because they are numerically stable and they keep the condition number of the matrix unchanged (when using the 2-norm). The random transformation proposed in [27] is based on the Discrete Fourier Transform and the transformation proposed in [26] is referred to as Random Butterfly Transformation (RBT), which consists of preconditioning a given matrix using particular random matrices referred to as butterfly matrices or products of them. We will refer to the resulting method as RBT NP LU. The easiest way to think of the method is as performing LU with no pivoting (NP) on a preconditioned matrix, where the cost of preconditioning is negligible compared to the cost of the factorization itself.

Similarly to the Cholesky factorization, where no pivoting is required for symmetric and positive definite matrices, the NP LU can be of direct use for diagonally dominant matrices, as this case does not require pivoting. For general matrices, the RBT transformation helps but in general the accuracy is reduced and requires adding iterative refinement in the working precision (see [2] where solutions of linear systems using PP LU, RBT NP LU and QR are compared for some matrices from Higham's collection [17]). Another technique to improve the stability is to add "limited" pivoting (LP). By limited pivoting we mean that the search for pivots will not be the entire panel as in PP LU, but will be limited to the first few rows of the panel, e.g., NB rows or NB + 64. In Fig. 5, we compare, for different sizes of random matrices, the error in the LU factorization expressed by the relative error $\|PA - LU\|_2/\|A\|_2$. This error is plotted for the following algorithms: partial pivoting (PP LU), limited pivoting (RBT LP LU), and no pivoting (RBT NP LU; for this case we mention the maximum and minimum values obtained for a sample of matrices). PP LU is the LU factorization as it is implemented in LAPACK. The accuracy of RBT LP LU is computed when pivoting within the first NB rows and within the first NB + 64 rows of the panel (or less if this exceeds the rows in the panel). RBT LP LU(NB + invert) corresponds to the case where we pivot within the first NB rows; the obtained L factor is explicitly inverted and the inverse is used in updating the trailing sub-matrix on the right of the current block. Note that the computational cost of adding limited pivoting is affordable because it does not change the Level 3 BLAS nature of the current implementation (performance results are given in the next section).

4.3. Performance and numerical results

Here we give the performance of our implementation of the RBT LP LU algorithm and put it in the context of other LU factorizations and their performances. The performance results are for NVIDIA's GeForce GTX 280 GPU and its multicore host, a dual-socket quad-core Intel Xeon running at 2.33 GHz. On the multicore we use LAPACK and BLAS from MKL 10.0 and on the GPU CUBLAS 2.1. To compile we use gcc version 4.1.2. The parameter NB is tuned empirically [21]. In particular, for double precision arithmetic for matrices of size up to 1024, we use NB = 64, for matrices from 1024 to 4096 we use NB = 128, and for larger we use NB = 256. For single precision arithmetic for matrices of size up to 4032, we use NB = 64, from 4032 to 8064

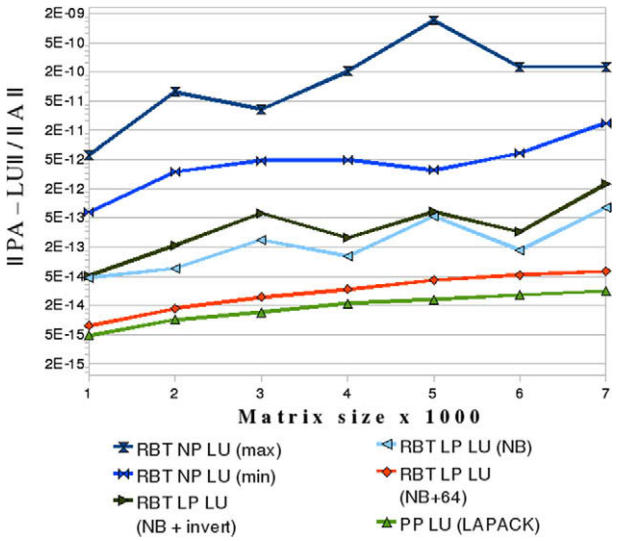

Fig. 5. Accuracy of double precision LU factorizations on random matrices.

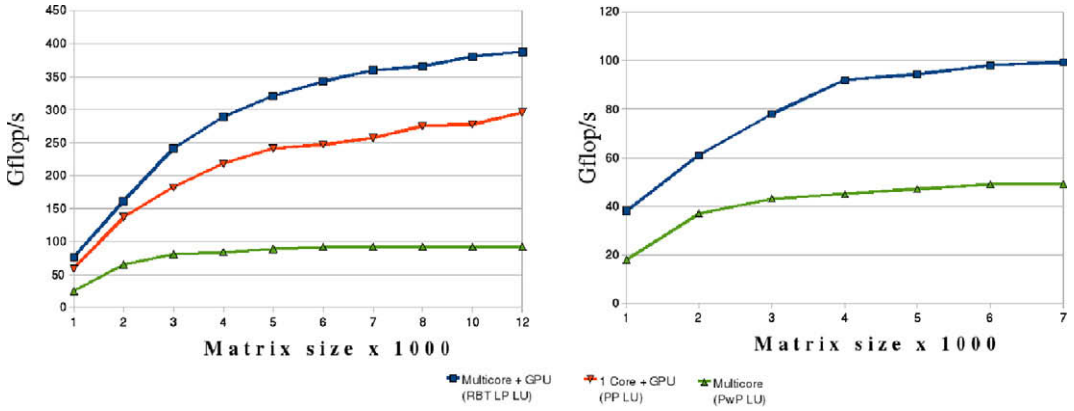

Fig. 6. Performance for the RBT LP LU algorithm on a hybrid Intel Xeon (2 × 4 at 2.33 GHz) + GeForce GTX 280 (240 at 1.30 GHz) for correspondingly single (left) and double (right) precision arithmetic.

we use NB = 128, from 8064 to 9984 we use NB = 192, and for larger we use NB = 256. For this particular algorithm the work done by the core processing the panels is proportional to the work for the other cores as they all perform Level 3 BLAS operations on NB columns of the same length. This ensures load balance among the cores. The trailing 7 NB × 7 NB matrix from item 5 of the algorithm (in Section 4.1) is factored using 1 core + GPU with new block size equal to NB divided by 2. Fig. 6 shows the performance results. On the left we have the performance for single precision arithmetic and on the right for double precision. The algorithm denoted by 'RBT LP LU' is performing local pivoting within the block size and uses explicitly inverted lower triangular matrices resulting from the panel factorization to update the trailing matrix (as suggested in [35] for performance reasons). All the pivoting is done on the CPU. The pivoting and the following update on the submatrix on the right of the diagonal is replaced by flipping the corresponding rows of the explicitly inverted lower triangular matrix from the panel factorization (done on the CPU) and multiplying it by the rectangular submatrix on the right of the current block.

Both graphs show performances on a multicore and on a multicore enhanced with a GPU. The multicore performance is for a pairwise pivoting LU algorithm from the PLASMA library [7]. In both cases the host is an Intel Xeon Harpertown with GEMM performance as shown on Fig. 1.

Compared to V. Volkov's single precision PP LU, our RBT LP LU code runs from 17 Gflop/s (for small matrices) to 82 Gflop/s faster on the platform under consideration. Overall this is about 28% faster.

As already mentioned, one of the techniques to improve the accuracy consists in adding limited pivoting. Another possibility is to add iterative refinement in the working precision. The cost of adding it can also be reduced by having explicitly available triangular matrix inverses that are byproducts of the factorization. For example, we developed blocked CUDA STRSV-like and DTRSV-like routines [19] that replace triangular solves (within the block) with matrix multiplication to get a corresponding performance of up to 14 GFlop/s (for matrix of size 14,000) and 6.7 GFlop/s (for matrix of size 7000). Note that just using cublasStrsm or cublasDtrsm, the performance would be, respectively, 0.24 GFlop/s and 0.09 GFlop/s. And then the cost of iterating refinement will not be negligible compared to the cost of the factorization (cublasStrsv and cublasDtrsv are about 5 times faster but the matrix size should not exceed, respectively, 4070 and 2040). We note that the new routines can be used for mixed-precision iterative refinement solvers [6,20] as well, where performing iterative refinement on the GPU has a negligible cost compared to that of the factorization itself.

5. Conclusions and future directions

Major chip manufacturers are developing future next-generation microprocessors that are heterogeneous, integrating multicore CPU and GPU components. In order to fully exploit these architectures, software designers would need to use both of their GPU and CPU multicore components. We presented a new approach for developing efficient DLA algorithms on hybrid multicore + CPU architectures and we illustrated this technique with a hybrid LU factorization. Besides its hybridization, this new LU algorithm is of interest by itself, as it reduces the amount of pivoting (to an asymptotic minimum) while maintaining accuracy comparable to the accuracy of LU with partial pivoting. This paper is a first step in describing an example for the benefits of hybrid systems for DLA, motivating future work for creating a self contained DLA library similar in functionality to LAPACK but for hybrid architectures.

Acknowledgments

Part of this work was supported by the US National Science Foundation and the US Department of Energy. We thank NVIDIA and NVIDIA's Professor Partnership Program for their hardware donations. We thank also Jim Demmel and Vasily Volkov from UC Berkeley, and Massimiliano Fatica from NVIDIA for helpful discussions related to GPU computing.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK user's guide, SIAM, third ed., 1999.
[2] M. Baboulin, J. Dongarra, S. Tomov, Some issues in dense linear algebra for multicore and special purpose architectures, Technical Report UT-CS-08-615, University of Tennessee, 2008, LAPACK Working Note 200.
[3] G. Ballard, J. Demmel, O. Holtz, O. Schwartz, Minimizing communication in linear algebra, Technical Report, LAPACK Working Note 218, May 2009.
[4] S. Barrachina, M. Castillo, F. Igual, R. Mayo, E. Quintana-Ortí, Solving dense linear systems on graphics processors, Technical Report ICC 02-02-2008, Universidad Jaime I, February, 2008.
[5] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, S. Tomov, The impact of multicore on math software, PARA 2006, in: B. Kågström et al. (Ed.), Lecture Notes in Computer Science, vol. 4699, Springer, 2007, pp. 1–10.
[6] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, S. Tomov, Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy, ACM Trans. Math. Software 34 (4) (2008).
[7] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Technical Report UT-CS-07-600, University of Tennessee, 2007, LAPACK Working Note 191.
[8] J. Demmel, J. Dongarra, B. Parlett, W. Kahan, M. Gu, D. Bindel, Y. Hida, X. Li, O. Marques, E. Riedy, C. Vömel, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, J. Langou, S. Tomov, Prospectus for the next LAPACK and ScaLAPACK libraries, in: PARA'06: State-of-the-Art in Scientific and Parallel Computing (Umeå, Sweden), High Performance Computing Center North (HPC2N) and the Department of Computing Science, Umeå University, Springer, June 2006.
[9] J. Demmel, L. Grigori, M. Hoemmen, J. Langou, Communication-avoiding parallel and sequential QR factorizations, CoRR abs/0806.2159, 2008.
[10] J. Dongarra, P. Luszczek, A. Petitet, The LINPACK benchmark: past, present, and future, Concurrency and Computation: Practice and Experience 15 (2003) 820.
[11] J. Dongarra, S. Moore, G. Peterson, S. Tomov, J. Allred, V. Natoli, D. Richie, Exploring new architectures in accelerating CFD for air force applications, in: Proceedings of HPCMP Users Group Conference 2008, July 14–17, 2008. <http://www.cs.utk.edu/~tomov/ugc2008_final.pdf>.
[12] K. Fatahalian, J. Sugerman, P. Hanrahan, Understanding the efficiency of GPU algorithms for matrix–matrix multiplication, in: HWWS '04: Proceedings of the ACM Siggraph/Eurographics Conference on Graphics Hardware (New York, NY, USA), ACM, 2004, pp. 133–137.
[13] M. Fatica, Accelerating LINPACK with CUDA on heterogenous clusters, in: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (New York, NY, USA), ACM, 2009, pp. 46–51.
[14] N. Galoppo, N. Govindaraju, M. Henson, D. Manocha, 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, p. 3.
[15] L. Grigori, J. Demmel, H. Xiang, Communication avoiding Gaussian elimination, Technical Report 6523, INRIA, 2008.
[16] Wolfgang Gruener, Larrabee, CUDA and the quest for the free lunch, TGDaily. <http://www.tgdaily.com/content/view/38750/113/,08/2008>.
[17] N. Higham, Accuracy and Stability of Numerical Algorithms, second ed., SIAM, 2002.
[18] J. Hruska, AMD fusion now pushed back to 2011, Art Technica (2008).
[19] B. Kågström, P. Ling, C. van Loan, GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark, ACM Trans. Math. Software 24 (3) (1998) 268–302.
[20] Julie Langou, Julien Langou, P. Luszczek, J. Kurzak, A. Buttari, J. Dongarra, Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems), in: SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (New York, NY, USA), ACM, 2006, p. 113.
[21] Y. Li, J. Dongarra, S. Tomov, A note on auto-tuning GEMM for GPUs, Technical Report, LAPACK Working Note 212, January 2009.
[22] NVIDIA, Nvidia Tesla doubles the performance for CUDA developers, Computer Graphics World (06/30/2008).
[23] NVIDIA, NVIDIA CUDA Programming Guide, 6/07/2008, Version 2.0.
[24] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips, GPU computing, Proceedings of the IEEE 96 (5) (2008) 879–899.
[25] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, T. Purcell, A survey of general-purpose computation on graphics hardware, Comput. Graphics Forum 26 (1) (2007) 80–113.
[26] D. Parker, Random butterfly transformations with applications in computational linear algebra, Technical Report CSD-950023, Computer Science Department, UCLA, 1995.
[27] D. Parker, B. Pierce, The randomizing FFT: an alternative to pivoting in Gaussian elimination, Technical Report CSD-950037, Computer Science Department, UCLA, 1995.
[28] M. Pharr, R. Fernando, GPU Gems 2: Programming Techniques for High-performance Graphics and General-purpose Computation (gpu gems), Addison-Wesley Professional, 2005.
[29] G. Quintana-Ortí, F.Igual, E.Quintana-Ortí, R. van de Geijn, Solving dense linear systems on platforms with multiple hardware accelerators, in: PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (New York, NY, USA), ACM, 2009, pp. 121–130.
[30] G. Quintana-Orti, E. Quintana-Orti, E. Chan, F. van Zee, R. van de Geijn, Programming algorithms-by-blocks for matrix computations on multithreaded architectures, Technical Report TR-08-04, University of Texas at Austin, 2008, FLAME Working Note 29.
[31] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan, Larrabee: a many-core $\times$ 86 architecture for visual computing, ACM Trans. Graph. 27 (3) (2008) 1–15.
[32] S. Tomov, M. Baboulin, J. Dongarra, S. Moore, V. Natoli, G. Peterson, D. Richie, Special-purpose hardware and algorithms for accelerating dense linear algebra, in: Parallel Processing for Scientific Computing, Atlanta, March 12–14, 2008. <http://www.cs.utk.edu/~tomov/PP08_Tomov.pdf>.
[33] S. Tomov, J. Dongarra, Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing, Technical Report 219, LAPACK Working Note, May 2009.
[34] V. Volkov, J. Demmel, Benchmarking gpus to tune dense linear algebra, in: SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Piscataway, NJ, USA), IEEE Press, 2008, pp. 1–11.
[35] LU, QR, Cholesky factorizations using vector capabilities of GPUs, Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.
[36] Using GPUs to accelerate linear algebra routines, Poster at PAR lab winter retreat, January 9, 2008. <http://www.eecs.berkeley.edu/~volkov/volkov08-parlab.pdf>.
[37] General-purpose computation using graphics hardware, <http://www.gpgpu.org>.
[38] Nvidia cuda zone, NVIDIA. <http://www.nvidia.com/object/cuda_home.html>