Some Issues in Dense Linear Algebra for Multicore and Special Purpose Architectures
LAPACK Working Note 200

Marc Baboulin[1,2], Jack Dongarra[2,3,4], and Stanimire Tomov[2]

[1] University of Coimbra (Portugal)
[2] University of Tennessee (USA)
[3] Oak Ridge National Laboratory (USA)
[4] University of Manchester (UK)

Abstract. We address some key issues in designing dense linear algebra (DLA) algorithms that are common for both multi/many-cores and special purpose architectures (in particular GPUs). We present them in the context of an LU factorization algorithm, where randomization techniques are used as an alternative to pivoting. This approach yields an algorithm based entirely on a collection of small Level 3 BLAS type computational tasks, which has emerged as a common goal in designing DLA algorithms for new architectures. Other common trends, also considered here, are block asynchronous task execution and "Block" layouts for the data associated with the separate tasks. We present numerical results and other specific experiments with DLA algorithms on NVIDIA GPUs using CUDA. The GPU results are also of interest themselves as we show a performance of up to 160 Glop/s on a single Quadro FX 5600 card.
Keywords: dense linear algebra, parallel algorithms, LU factorization, multicore processors, graphic process units.

1 Introduction

Parallel platforms based on multicore chips are becoming dominant systems in High Performance Computing (HPC). In the Top500 list released in November 2007, almost 90% of the systems are based on dual-core or quad-core architectures. Moreover, special-purpose hardware, like GPUs or the CELL BE, and even reconfigurable hardware (e.g FPGAs), are also becoming pervasive in the HPC world, as evident from many current conferences in the field (e.g. including PARA08) and as they are included more often as accelerators in HPC systems. The changes introduced in these new architectures create a need for the development of innovative algorithms that would efficiently use the new hardware. Major common challenges here are not only to design algorithms of high parallelism but also algorithms that would overcome the exponentially growing gap between processor speed and memory (e.g. CPU speeds have been improving at 59% per year, main memory bandwidth at only 23%, and main memory latency at a mere 5.5% [6]). In other words, we need parallel algorithms of high enough ratio of floating point calculations to data required to mask slow memory speeds.

There is a common understanding on how to design certain DLA algorithms for current multicores chips. As mentioned in [4], algorithms should satisfy the following criteria to take advantage of multicore processors:

- fine granularity, as cores are associated with relatively small local memories,
- asynchronicity, to hide the latency of access to memory.

These ideas are applied in current efforts for developing efficient DLA algorithms for multicore [4, 18]. The fine granularity is achieved by splitting the operations into tasks that operate on smaller blocks, resulting in so-called "tiled" algorithms while asynchronicity is achieved by dynamically scheduling the tasks using a Directed Acyclic Graph (DAG). Data storage is also essential for effective computations and Block Data Layout [9] can be successfully applied to tiled algorithms. Variations of these ideas can be also recognized in algorithms for GPUs as we show in Section 3, the CELL BE [13], and even FPGAs (e.g. in the case of out-of-core FPGA problems or multi-FPGA use).

The general directions just outlined work well when an algorithm can generate a collection of independent tasks, each of high ratio of floating point calculations to data required. A subject of current research in the field is to design algorithms where all the tasks involved are of Level 3 BLAS. For example, block Cholesky already has this property, but the traditional block Householder QR and block LU with partial pivoting do not, as they have panels involving Level 2 BLAS. For QR, certain out-of-core versions remove this limitation [8], and for LU, the randomization techniques (among others) lead to entirely Level 3 BLAS algorithms, as described in Section 2.

2 An Alternative to Pivoting in Algorithms for New Architectures

2.1 Randomization Technique to Avoid Pivoting

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 (GEPP) is implemented in current linear algebra libraries for solving square linear systems $Ax = b$ resulting in very stable algorithms. In the LAPACK [1] implementation of GE, during pivoting rows are swapped at once, which inhibits the exploitation of more asynchronicity between block operations.

In a recent paper, [7] describes a pivoting strategy that minimizes the number of messages exchanged during the panel factorization and shows that this approach is stable in practice. For multicore, pairwise pivoting is often considered (e.g in [4]) but this generates a significant overhead since the rows are swapped in pairs of blocks. Still for multithreaded architecture, [18] 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 [11]. For implementation of GE on GPUs, the cost of pivoting may represent more than 30% of the global computation. To achieve higher performance on the new architectures like multicore or GPUs, it is worth investigating other forms of GE,

possibly less stable than GEPP.
This study is experimental and based on statistical results and observations. The first question we may ask is: do we have to pivot for random matrices? In Figure 1, we consider matrices normally distributed $\mathcal{N}(0,1)$ of various sizes (sample of 100 matrices for each size) and we compare the error in the LU factorization obtained when we do partial pivoting (GEPP) and no pivoting at all (GENP). We observe that the error obtained with GENP is almost always between 10^{-10} and 10^{-14} and thus, following [19] and [10, p. 239], we could get a solution as accurate as GEPP just by adding iterative refinement in fixed precision. Then a first empirical result here is that there would be no need for pivoting when the matrix is $\mathcal{N}(0,1)$. Moreover, it is observed in [20] that for many distributions of

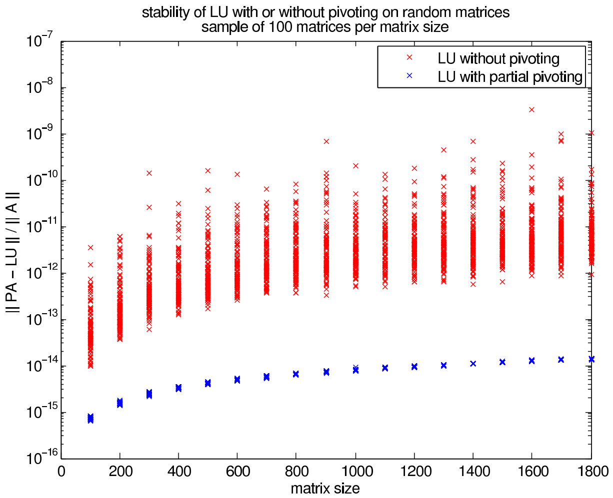

Fig. 1. Pivoting vs nonpivoting in LU on matrices $\sim \mathcal{N}(0,1)$

matrices, the matrix elements after the first few steps of GE (using partial or complete pivoting) are approximately normally distributed.

Then the case without pivoting was studied in [22] where satisfying probabilistic bounds on growth factors are given for the occurrence of small pivots and for the growth factors when the entries of A are $\mathcal{N}(0,1)$.

The idea of [16, 17] was 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 [17] is based on the Discrete Fourier Transform and the transformation proposed in [16] is referred to as Random Butterfly Transformation (RBT) which consists of preconditioning a given matrix A using particular random matrices referred to as butterfly matrices or products of them. Having the butterfly matrices U

and V, then GENP is performed on the matrix U^*AV and, to solve $Ax = b$, we instead solve $(U^*AV)y = U^*b$ followed by $x = Vy$, where U^* denotes the conjugate transpose of U. Both transformations use complex arithmetic and require efficient implementations of FFT-like computations. In addition to the drawback of handling complex-valued matrices, resulting in extra-storage and extra-computation, we may also have an overhead due to random numbers generation. This is why, in Section 2.3, we perform numerical experiments with real-valued butterfly matrices. For better stability, we add systematically iterative refinement (in the working precision) when we do GENP on a randomized matrix.

2.2 Using QR Factorization for Solving Linear Systems

Another element that we would like to point out is that, if we want to avoid pivoting, this is always possible to use the QR factorization to solve linear systems. We recall here the following theorem from [10, p. 361] that shows the interest in terms of backward stability for using the Householder QR factorization for linear systems.

Theorem 1 *Let $A \in \mathbb{R}^{n\times n}$ be non singular. Suppose we solve the system $Ax = b$ with the aid of a QR factorization computed by the Householder algorithm. The computed $\hat{x}$ satisfies*

$$(A + \Delta A)\hat{x} = b + \Delta b,$$

where

$$\|\Delta a_j\|_2 \leq \tilde{\gamma}_{n^2}\|a_j\|_2,\ j = 1 : n, \qquad \|\Delta b\|_2 \leq \tilde{\gamma}_{n^2}\|b\|_2.$$

In Theorem 1, Δa_j denotes the jth column of A and $\tilde{\gamma}_{n^2}$ is an integer constant of the form $\frac{cn^2u}{1-cn^2u}$ where c is a small integer constant and u is the unit roundoff. This theorem implies a small column-wise relative backward error but not a small component-wise relative backward error ω. However, [10] shows also that ω will be small after one step of iterative refinement, provided that A is not too ill conditioned and $|A||\hat{x}|$ is not too badly scaled. Contrary to GEPP, we do not have to worry here about large element growth.

The computational cost for solving a linear system with Householder QR is about twice that of an LU factorization ($4n^3/3 + n^2$ vs $2n^3/3 + 2n^2$) but QR is well suited for tiled algorithms since it is rich in Level 3 BLAS operations. In the worst case where pivoting requires half the time of the whole factorization, QR is a very competitive option because of its stability properties.

2.3 Numerical Experiments

Experiments on accuracy were performed using Matlab on matrices of size 1024 from Matlab gallery and Higham's Matrix Computation Toolbox [10] (matrix gfpp for which the growth factor for GEPP is maximum). The right-hand side is generated from a uniform distribution on $[0, 1]$. We use here a simple form of real-valued butterfly, also given in [16], of the form $\begin{pmatrix} P & Q \\ R & S \end{pmatrix}$, where $P,\ Q,\ R$ and S are diagonal random $n/2 \times n/2$ matrices.

In Table 1, we compare the linear system solution obtained using GEPP (as it is implemented in LAPACK), GENP, GENP followed by RBT, and QR. In each case we have the possibility to add iterative refinement, depending on the stopping criterion given below. We report the component-wise relative backward error from [15] expressed by

$$\omega = \max_i \frac{|A\hat{x} - b|_i}{(|A| \cdot |\hat{x}| + |b|)_i},$$

and, similarly to [2, 19], the iterative refinement algorithm is activated while $\omega > (n+1)u$. We also report the number of iterations in the iterative refinement process. For the 3 first matrices, using RBT is not useful because GENP gives a good solution. However this shows that these matrices are not degenerated by the randomization applied to them. We observe that GENP fails on the 3 last matrices. Iterative refinement turns out to be necessary when using RBT and in each case, it gives a backward error that is similar or better than GEPP except for the matrix "chebspec". This latter case is in accordance with [16] who mentioned that RBT is less accurate than GEPP for ill-conditioned matrices. Note also that even though iterative refinement was not necessary for QR on the matrices of Table 1, it may be useful in some cases [10, p. 240].

Matrix	chebspec	circul	condex	fiedler	orthog	gfpp
Cond	$6 \cdot 10^{14}$	$5 \cdot 10^{2}$	$1 \cdot 10^{2}$	$2 \cdot 10^{5}$	$1 \cdot 10^{0}$	$2 \cdot 10^{2}$
GEPP	$5 \cdot 10^{-16}$	$1 \cdot 10^{-15}$	$2 \cdot 10^{-15}$	$2 \cdot 10^{-15}$	$2 \cdot 10^{-15}$	$2 \cdot 10^{-2}$
# iter	0	0	0	0	0	10
GENP	$5 \cdot 10^{-16}$	$1 \cdot 10^{-15}$	$4 \cdot 10^{-15}$	Fail	Fail	Fail
# iter	0	1	0	–	–	–
QR	$9 \cdot 10^{-16}$	$2 \cdot 10^{-15}$	$3 \cdot 10^{-15}$	$6 \cdot 10^{-15}$	$3 \cdot 10^{-16}$	$1 \cdot 10^{-16}$
# iter	0	0	0	0	0	
RBT+GENP	$6 \cdot 10^{-14}$	$1 \cdot 10^{-15}$	$4 \cdot 10^{-15}$	$1 \cdot 10^{-15}$	$4 \cdot 10^{-16}$	$2 \cdot 10^{-16}$
# iter	3	1	1	1	2	1

Table 1. Comparison of linear system solutions using GEPP and RBT on some matrices.

In Figure 2 we report performance in Gflop/s for the LU factorization on GPU where GEPP and GENP are LAPACK-like implementations in which we have changed BLAS routines by CUDA BLAS ones. RBT+GENP gets the performance of GENP since the cost of the random transformation is negligible on GPU for this particular type of butterfly matrices. We observe that performance of the LU factorization is improved by more than 30% by using randomization before GENP with LAPACK-like implementations. However, since by using randomization we know that we are not going to pivot, we can optimize GENP by using only Level 3 BLAS (contrary to what is done in LAPACK). In that case, as shown in Figure 2, the performance of GENP+RBT is more than twice that of GEPP. More GPU implementation details and performance related issues will be discussed in the next section.

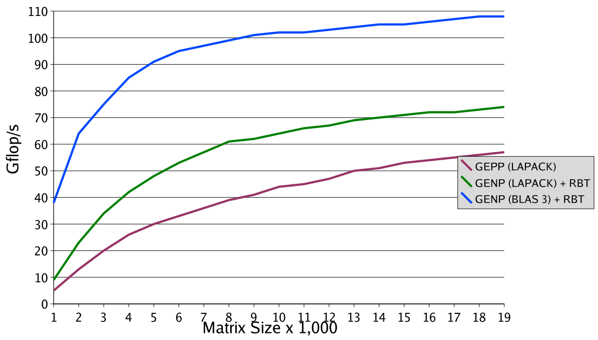

Fig. 2. Performance of single precision LU factorization using GEPP and RBT+GENP on GPUs

3 DLA Experiments Using GPUs

We have done a series of experiments using NVIDIA GPUs, trying to further develop the methodology in efficiently using these architectures for DLA. The computations presented here are on a Quadro FX 5600 installed on 4× Dual Core AMD Opteron(tm) Processor 256 (1800 MHz, 1024 KB cache) with PCIe×16. For most part we used CUDA 1.0 and an optimized sgemm kernel from V. Volkov [21] running at about 180 Gflop/s. We also used LAPACK and ATLAS. The techniques discovered during our work tend to have common ground with algorithm design principles for other new architectures, as we show in this section.

The main challenges to effectively program DLA for GPUs, similarly to many-cores and other new architectures, is to design algorithms of high parallelism and high ratio of floating point calculations to data required (also known as high computational intensity). And indeed, algorithms for GPUs, with processors count much higher than for current multicores (e.g. NVIDIA's Quadro FX 5600 has 128), have to be designed to split the computation into many parallel tasks and each task to have high enough computational intensity (CI) in order to be efficiently executed on a single processing element (PE). Figure 3 shows NVIDIA GPU's hardware model and a typical programming pattern [14] where blocks of data are "pooled" into the fast shared memory followed by computation. Usually many more treads of execution (than number of PE available) are encouraged in order to overlap computation and communication (hide high latencies to get data from the main memory). We outlined in Section 1 one general idea that seems promising for current multicores, namely to design algorithms of

- fine granularity (through **task splitting**), and
- **asynchronicity**.

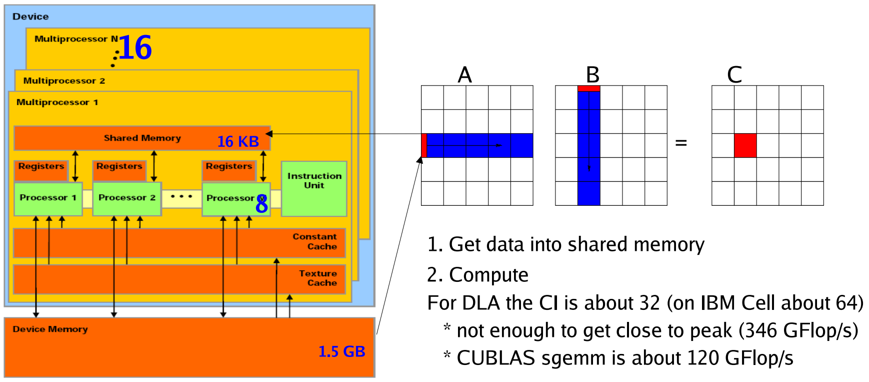

Fig. 3. Hardware model and a programming example model for NVIDIA GPU.

Here, as GPUs are better suited for data-parallel computations,

1. The tasks splitting has to be done within the BLAS level (BLAS level parallelism), and
2. Asynchronicity can be added in **hybrid CPU-GPU** computations where a large computationally intensive task is run on the GPU and small tasks, independent of the task being run on the GPU and usually sequential part of an algorithm (that we would like to "hide"), are asynchronously started on the CPU or other fast computational devices, and in particular FPGAs.

In view of this broad design direction, our first experiment is to test the performance of LAPACK routines by just replacing their BLAS calls with BLAS for GPUs (e.g. CUBLAS [5]). Figure 4, Left shows the performance of LAPACK's LU, QR, and Cholesky factorizations on NVIDIA's Quadro FX 5600 using CUBLAS. In this experiment memory is allocated on the GPU, the CPU runs the LAPACK code which is a sequence of BLAS calls that get executed on the GPU. There is no large memory transfers as the matrix to be factored and the work space stay only on the GPU throughout the computation. Note that the programming efforts here are insignificant and we get a "good" performance for large problems, but still, we are not getting close to the peak (e.g. sgemm in CUBLAS 1.0 runs at 120 Gflop/s). Similar deficiency about this approach, just based on BLAS level parallelism, has also been observed for multicores. Finally, to underline the importance of having entirely Level 3 BLAS algorithms, note that the block Cholesky factorization is the fastest of the three.

Our next experiment shows the effect of asynchronicity in hybrid CPU-GPU algorithm designs. An easy way to demonstrate it is using the left-looking Cholesky factorization [5, p. 86]. A step of the algorithm involves two tasks that are independent and can be asynchronously scheduled, namely a "large" sgemm-type update of the trailing matrix can be started on the GPU and at the same time a Cholesky factorization of a "small" block (from the diagonal

[5] See http://www.nvidia.com/object/cuda_home.html

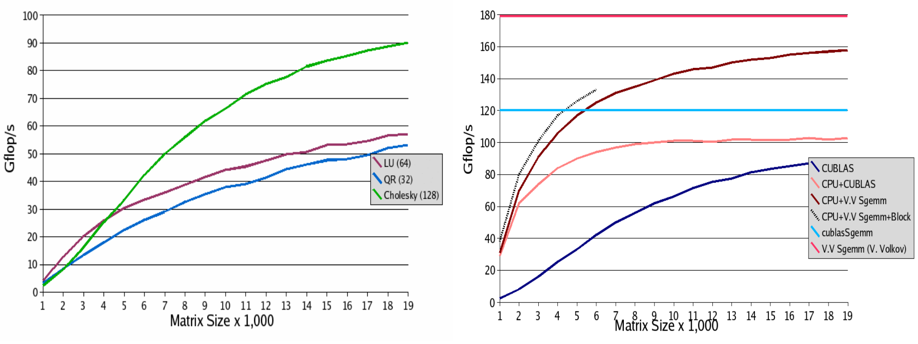

Fig. 4. Left: Performance of LAPACK's single precision LU, QR, and Cholesky factorizations using NVIDIA's CUDA BLAS (CUBLAS) on a Quadro FX 5600; Right: hybrid CPU + GPU computation on Cholesky factorization.

of the matrix) on the CPU (using LAPACK's spotf2), resulting in overlapping (or hiding) the sequential small task with the large highly parallel task. Adding this optimization more than doubles overall performance on smaller problems, as shown on Figure 4, Right (the pink line 'CPU + CUBLAS' *vs* the blue line 'CUBLAS'). The sequence of starting the kernels from the CPU is as follows:
(1) cublasGetMatrix to get the data for spotf2 from the GPU to the CPU,
(2) cublasSgemm, which is asynchrononous so the CPU can start
(3) spotf2, and finally
(4) cublasSetMatrix to move the result of spotf2 back to the GPU.
On current NVIDIA cards communications steps (1) and (4) can not be overlapped with computations but in future cards will be provided. We note that we have also tried to code small tasks (like the diagonal block factorizations in Cholesky) directly in CUDA but there is not enough parallelism available in these small problems, and as a result even if no overlap was possible performance is slower compared to performance of transferring and factoring them on the CPU. The same is true for Level 2 BLAS panel factorizations, up to certain size, when the GPU would become more efficient to execute them. The same observations were made also in [3, 21]. Another idea that we are currently exploring, and that is related to this observation, is to use reconfigurable computing, and in particular FPGAs, in hybrid calculations of this type.

The approach, as described so far involves very little programming development efforts since we are using LAPACK which plugs into the CUDA BLAS. The performance though greatly depend on having optimized BLAS. For example, Figure 4, Right shows the effect of using better sgemm for the Cholesky factorization. Namely, we used a code from V. Volkov [21] that achieves 180 Gflop/s on sgemm *vs* the 120 Gflop/s of CUBLAS [6]. We note that based on optimized sgemm other kernels can be easily derived [12], e.g. it is trivial (two

[6] CUDA Beta 2.0 includes sgemm from V. Volkov, achieving 206 Gflop/s.

if statements) to get a ssyrk implementation from sgemm that achieves around 120 Gflop/s *vs* 36 Gflop/s in CUBLAS 1.0.

As stressed before, we are able to apply efficiently the techniques mentioned because we can represent the Cholesky factorization as a collection of small Level 3 BLAS type computational tasks. The techniques from Section 2 allows us to do the same for the LU factorization, and as a result we see performance of the new algorithm comparable to that of Cholesky (the blue curve "GEPP (LAPACK)" from Figure 2 and the orange "CPU+CUBLAS" from Figure 4, Right, both done using CUBLAS 1.0).

Related to data structures, we found that Block Data Layout (BDL) can give performance benefits in several ways. First, when using hybrid CPU + GPU computations small blocks of data have to be transferred between the CPU and GPU. This operation is much faster when performed on contiguous data, which is the case when using BDL (and pinned memory). Example is given again on Figure 4, Right, where the dotted line 'CPU+V.V.Sgemm + Block' shows a performance improvement of about 10 Gflop/s. The BDL can also help to speed up BLAS kernels. Finally, we want to point out that mixed precision iterative refinement is important for hybrid CPU+GPU computations since, similarly to [13], it enables us to improve the performance while maintaining the accuracy.

4 Conclusions

We addressed some key issues in designing DLA algorithms and showed that they are common for both multi/many-cores and special purpose architectures. We extended these common ideas in an innovative way to GPUs where we designed an LU and a Cholesky factorization algorithms to obtain an impressive performance of up to 160 GFlop/s. The approach relied on third party optimized BLAS for GPUs and required insignificant programming efforts. Crucial for the speedups obtained was the use of a hybrid CPU-GPU calculation and a randomization technique as an alternative to pivoting that allowed us to cast the LU factorization as an entirely Level 3 BLAS computation.

Acknowledgments. Part of this work was supported by Air Force Office of Scientific Research under contract # FA9550-07-C-0089. We thank NVIDIA and NVIDIA's Professor Partnership Program for their hardware donations. We thank also Jim Demmel, Vasily Volkov, and Julien Langou for helpful discussions related to GPU computing and pivoting issues.

References

1. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK user's guide*, SIAM, 1999, Third edition.
2. M. Arioli, J. W. Demmel, and I. S. Duff, *Solving sparse linear systems with sparse backward error*, SIAM J. Matrix Analysis and Applications **10** (1989), no. 2, 165–190.

3. S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and Enrique S. Quintana-Ort, *Solving dense linear systems on graphics processors*, Tech. report, February 2008, Universidad Jaime I, ICC 02-02-2008.
4. A. Buttari, J. Langou, J. Kurzak, and 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.
5. J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst, *Numerical linear algebra for high-performance computers*, SIAM, 1998.
6. Susan L. Graham, Marc Snir, and Cynthia A. Patterson (eds.), *Getting up to speed, the future of supercomputing*, The National Academies Press, 2006.
7. L. Grigori, J. W. Demmel, and H. Xiang, *Communication avoiding Gaussian elimination*, Technical Report, INRIA, 2008.
8. B. Gunter and R. van de Geijn, *Parallel out-of-core computation and updating of the QR factorization*, ACM Trans. Math. Softw. **31** (2005), no. 1, 60–78.
9. F. G. Gustavson, *New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms*, (June 20-23, 2004), 11–20, In Proceedings of PARA 2004, Workshop on state-of-the art in scientific computing.
10. N. J. Higham, *Accuracy and stability of numerical algorithms*, SIAM, 2002, Second edition.
11. T. Joffrain, E. S. Quintana-Orti, and R. A. van de Geijn, *Rapid development of high-performance out-of-core solvers.*, (June 20-23, 2004), 413–422, In Proceedings of PARA 2004, Workshop on state-of-the art in scientific computing.
12. B. Kärgström and C. Van Loan, *Gemm-based level-3 blas*, Tech. report, December 1989, Report CTC91TR47, Department of Computer Science, Cornell University.
13. Jakub Kurzak and Jack Dongarra, *Implementation of mixed precision in solving systems of linear equations on the cell processor: Research articles*, Concurr. Comput. : Pract. Exper. **19** (2007), no. 10, 1371–1385.
14. NVIDIA, *NVIDIA CUDA Programming Guide*, 6/23/2007, Version 1.0.
15. W. Oettli and W. Prager, *Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides*, Numerische Mathematik **6** (1964), 405–409.
16. D. S. Parker, *Random butterfly transformations with applications in computational linear algebra*, Technical Report CSD-950023, Computer Science Department, UCLA, 1995.
17. D. S. Parker and B. Pierce, *The randomizing FFT: an aternative to pivoting in Gaussian elimination*, Technical Report CSD-950037, Computer Science Department, UCLA, 1995.
18. G. Quintana-Orti, E. S. Quintana-Orti, E. Chan, F. G. van Zee, and R. A. 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.
19. R. D. Skeel, *Iterative refinement implies numerical stability for Gaussian elimination*, Math. Comput. **35** (1980), 817–832.
20. L. N. Trefethen and R. S. Schreiber, *Average-case stability of Gaussian elimination*, SIAM J. Matrix Analysis and Applications **11** (1990), no. 3, 335–360.
21. V. Volkov and J. W. Demmel, *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.
22. M. Yeung and T. F. Chan, *Probabilistic analysis of Gaussian elimination without pivoting*, SIAM J. Matrix Analysis and Applications **18** (1997), no. 2, 499–517.