A Randomized LU-based Solver Using GPU and Intel Xeon Phi Accelerators

Marc Baboulin, Amal Khabou(✉), and Adrien Rémy

Université Paris-Sud, Orsay, France
{baboulin,amal.khabou,aremy}@lri.fr

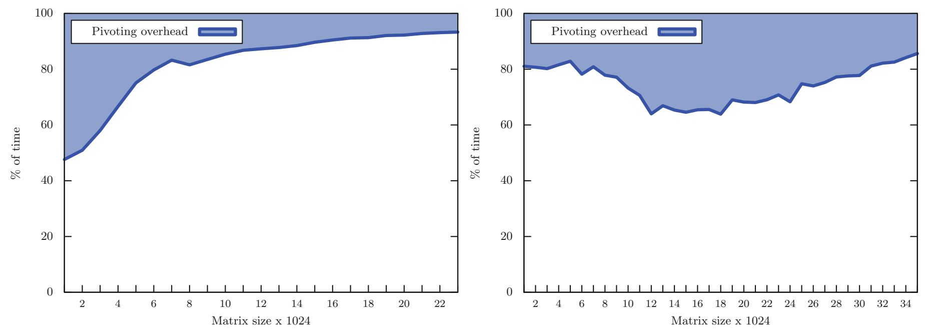
Abstract. We present a fast hybrid solver for dense linear systems based on LU factorization. To achieve good performance, we avoid pivoting by using random butterfly transformations for which we developed efficient implementations on heterogeneous architectures. We used both Graphics Processing Units and Intel Xeon Phi as accelerators. The performance results show that the pre-processing due to randomization is negligible and that the solver outperforms the corresponding routines based on partial pivoting.

Keywords: Random Butterfly Transformations (RBT) · LU factorization · Graphics Processing Units (GPU) · Intel Xeon Phi

1 Introduction

The LU factorization with partial pivoting is the most commonly used method to solve general dense linear systems. The pivoting step aims at improving the numerical stability of the method. Even though it does not require extra floating point operations, selecting the pivots involves $\mathcal{O}(n^2)$ comparisons. Moreover swapping the rows of the matrix involves extra data movements. These aspects can deteriorate the performance of the LU factorization due to the cache invalidations they induce.

As a motivation of this work, let us evaluate the overhead of the pivoting step of the LU factorization with partial pivoting using both GPU and Intel Xeon Phi accelerators. To use accelerators in dense linear algebra computations, we base our work on the MAGMA library [4,8,16], which provides LAPACK interface functions, using GPUs or Intel Xeon Phi. Figure 1a shows the results obtained running the corresponding MAGMA routine on an NVIDIA Tesla K20 GPU accelerator. We observe that pivoting takes more than 20% of the total computational time for matrices of size smaller than 10^4. However for larger matrices, the pivoting overhead is reduced and most of the computational time is spent performing the matrix-matrix products (DGEMM) on the GPU. Figure 1b displays the pivoting overhead using an Intel Xeon Phi 7120 coprocessor. Experiments have shown that the Intel Xeon Phi version of the factorization needs a greater amount of data than that of the GPU to be efficient. Indeed, for a matrix size of order 6000, the performance of the LU based solver is around 200 Gflop/s on the Xeon Phi whereas it is around 500 Gflop/s on the GPU. Increasing the size of the problem, the performance of both versions tends towards 800 Gflop/s (for double precision). We note that for small matrices, the pivoting overhead on Xeon Phi is proportionally smaller than on GPU.



(a) on GPU. (b) on Intel Xeon Phi.

Fig. 1. Time breakdown for pivoting in the LU factorization.

The previous experiments show that pivoting is a bottleneck in terms of communication cost and parallelism for hybrid CPU/accelerator architectures. To reduce the communication cost of the classical pivoting strategies such as partial pivoting, some alternative pivoting techniques were proposed in the context of the communication-avoiding LU algorithms (CALU then CALU_PRRP) [6,12]. These techniques are based on tournament pivoting, which was shown to be as stable as partial pivoting in practice.

Another approach consists in avoiding pivoting, and therefore improving the performance of the factorization. This approach is based on the use of Random Butterfly Transformations (RBT). It was first described in [14,15], and recently revisited for general systems in [3] and for symmetric indefinite systems in [1,2]. The main difference of the RBT based methods with respect to the classical factorization methods consists in a randomization step, which recursively applies a sequence of butterfly matrices to the input matrix. The main advantage of randomizing is that it allows us to avoid the communication overhead due to pivoting. Tests performed on a collection of matrices [3] show that in practice two recursions are sufficient to obtain a satisfactory accuracy.

The RBT solvers are particularly suitable for accelerators. On one hand, avoiding pivoting on accelerators has an important impact on the performance. On the other hand, the structure of the butterfly matrices can be exploited to perform the randomization at a very low cost. In this work we present the implementation details of a randomized LU-based solver using GPU and Intel Xeon Phi accelerators and discuss its performance on both accelerators.

The remainder of this paper is organized as follows. Section 2 recalls the main principles of the RBT algorithm and how it can be used in a hybrid CPU/accelerator factorization. Sections 3 and 4 describe the implementation and performance of the RBT solver for GPU and Intel Xeon Phi, respectively. Section 5 has concluding remarks.

2 Hybrid RBT Solver

To solve a general linear system $Ax = b$ using a solver based on RBT, we perform the following steps:

- Compute $A_r = U^T AV$ with U, V *random* matrices (recursive butterfly matrices),
- Factorize A_r using Gausian Elimination with No Pivoting (GENP),
- Solve $A_r y = U^T b$, then $x = Vy$.

We recall that an n-by-n butterfly matrix B has the following structure,

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} R & S \\ R & -S \end{pmatrix},$$

where R and S are two random non singular $n/2$-by-$n/2$ diagonal matrices. The matrix B can then be stored in an n elements vector. A recursive butterfly matrix of depth d is defined as

$$W^{<n,d>} = \begin{pmatrix} B_1^{<n/2^{d-1}>} & & 0 \\ & \ddots & \\ 0 & & B_{2^{d-1}}^{<n/2^{d-1}>} \end{pmatrix} \times \cdots \times \begin{pmatrix} B_1^{<n/2>} & 0 \\ 0 & B_2^{<n/2>} \end{pmatrix} \times B^{<n>}, \tag{1}$$

where all $B_i^{<n>}$ blocks are size n butterfly matrices. When n is not a multiple of 2^d, we "augment" the matrix A with additional 1's on the diagonal.

Note that the GENP algorithm can be unstable due to potentially large growth factor. This is why we systematically perform iterative refinement on the computed solution of the randomized system. In this work, we use two recursion levels for the randomization ($d = 2$). The randomization cost is $8n^2$ flops, due to the block diagonal structure of the butterfly matrices, as demonstrated in [3]. Then the RBT algorithm adapted for hybrid architectures (CPU with an accelerator) performs the following steps:

1. Random generation and packed storage of the butterflies U and V on the host (CPU), while sending A to the device (accelerator) memory (padding is added if the size of the matrix A is not a multiple of 4).
2. The packed U and V are sent from the host memory to the device memory.
3. The randomization of A is performed on the device. It is done in-place (no additional memory needed).

4. The randomized matrix is factorized with GENP: the panel is factored on the host and the update of the trailing submatrix on the device.
5. We compute $U^T b$, and then solve $A_r y = U^T b$ on the device.
6. If necessary, iterative refinement is performed for y, on the device.
7. We compute the solution of $x = Vy$ on the device, and then send x to the host memory.

Let us now describe the randomization phase (step 3) using two n-by-n recursive butterfly matrices U and V of depth two. We consider that the input matrix A can be split into 4 blocks of same size, $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$. We consider the matrices $U = U_2 \times U_1$ and $V = V_2 \times V_1$, where U_1, V_1 are two butterfly matrices, and U_2, V_2 are two matrices of the form $\begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix}$. B_1 and B_2 are two $n/2$-by-$n/2$ butterfly matrices as illustrated in Eq. 1. We have $A_r = U^T AV = U_1^T \times U_2^T \times A \times V_2 \times V_1$. Thus we first apply U_2^T and V_2 to A. We note $A_r^1 = U_2^T \times A \times V_2$, the resulting matrix from the first recursion level. Then,

$$A_r^1 = \begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix} \times \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_1^T & 0 \\ 0 & B_2^T \end{pmatrix} = \begin{pmatrix} B_1 A_{11} B_1^T & B_1 A_{12} B_2^T \\ B_2 A_{21} B_1^T & B_2 A_{22} B_2^T \end{pmatrix}$$

This step consists of four independent products with depth-1 butterfly matrices of size $n/2$-by-$n/2$. We call the kernel used for each product of the form $U^T \times A \times V$ `Elementary multiplication`. We then compute A_r by applying U_1^T and V_1 to A_r^1. For that we use again the `Elementary multiplication` kernel. Implementation details of this kernel will be given in the next sections for both GPU and Intel Xeon Phi accelerator.

3 RBT for Graphics Processing Units

Here we present our randomized LU-based solver using GPU. In particular, we give implementation details of the randomization step, which are specific to the targeted accelerator. We note that our RBT solver exists for all precisions used in LAPACK (simple, double, simple complex and double complex) and is part of the MAGMA library since the 1.6.0 version[1].

3.1 Implementation

On hybrid CPU/GPU architectures, the RBT solver is performed as described in Sect. 2. Algorithm 1 describes the randomization steps performed on a given matrix A. It applies the depth-two RBT to the matrix A by processing first each $n/2$-by-$n/2$ quarter block of A (lines $5 \ldots 8$ in Algorithm 1), and then applying the level one recursion to the whole n-by-n matrix (line 10 in Algorithm 1) as described in Sect. 2. The application of the level two randomization consists in calling a specific GPU kernel, the `Elementary Multiplication GPU`, on each quarter of the matrix. This is due to the block diagonal structure of the butterfly matrix. Each call to `Elementary Multiplication GPU` kernel is performed using one GPU thread per element.

[1] http://icl.cs.utk.edu/magma/news/news.html?id=351.

Algorithm 1. Two-level randomization on GPU

```
Require: $A$ a pointer to the matrix $A$ on the GPU.
Require: $U$ a pointer to the matrix $U$ stored as a vector on the GPU.
Require: $V$ a pointer to the matrix $V$ stored as a vector on the GPU.
Require: $n$ the size of the matrix $A$
Ensure: $A \leftarrow U^T AV$
 1: $block\_height \leftarrow 32$
 2: $block\_width \leftarrow 4$
 3: Define a grid of threads per block, size : $(block\_height, block\_width)$
 4: Define a grid of blocks, size : $(\frac{n}{4 \times block\_height}, \frac{n}{4 \times block\_width})$
    {Assuming $n$ is divisible by $4 \times block\_height$ and $4 \times block\_width$}
    { All GPU kernels are called with the threads and grid dimensions defined before the call}
 5: Call : Elementary Multiplication GPU($A$, $\&U(n)$, $\&V(n)$, $n/2$)
 6: Call : Elementary Multiplication GPU($\&A(0, n/2)$, $\&U(n)$, $\&V(n + n/2)$, $n/2$)
 7: Call : Elementary Multiplication GPU($\&A(n/2, 0)$, $\&U(n + n/2)$, $\&V(n)$, $n/2$)
 8: Call : Elementary Multiplication GPU($\&A(n/2, n/2)$, $\&U(n + n/2)$, $\&V(n + n/2)$, $n/2$)
 9: Redefine a grid of blocks, size : $(\frac{n}{2 \times block\_height}, \frac{n}{2 \times block\_width})$
    {Assuming $n$ is divisible by $2 \times block\_height$ and $2 \times block\_width$}
10: Call : Elementary Multiplication GPU($A$, $U$, $V$, $n$) {Applying level 1 recursion}
```

The `Elementary Multiplication GPU` kernel performs $A \leftarrow U^T AV$, where U and V are vectors of size n containing the entries of the depth-one random butterfly matrices. Algorithm 2 shows the implementation details of the `Elementary Multiplication GPU` kernel. We use shared memory arrays for each block of threads to store the elements of U and V relative to this block and thereby improve the efficiency of the access to these elements.

3.2 Performance Results

In this section, we present performance results of our randomized LU-based solver on GPU. The experiments were carried out on a system composed of a GPU, NVIDIA Kepler K20, with 2496 CUDA cores running at 706 MHz and 4800 MB of memory and a multicore host composed of two Intel Xeon X5680 processors, each with 6 physical cores running at 3.33 GHz, and a Level 3 memory cache of 12 MB. The CPU parts of our code are performed using the multithreaded Intel MKL library [9].

Figure 2a shows that the CUDA [13] implementation of our RBT solver (either with or without iterative refinement) outperforms the classical LU factorization with partial pivoting from MAGMA. For large enough matrices (from size 6000) the obtained performance is about $20 - 30\%$ faster than the solver based on Gaussian elimination with partial pivoting. In our experiments, when we enable iterative refinement, one iteration is generally enough to improve the computed solution giving an accuracy similar to the one obtained using LU factorization with partial pivoting. The iterative refinement is performed on the

Algorithm 2. GPU Kernel: Elementary Multiplication GPU(A, U, V, n)

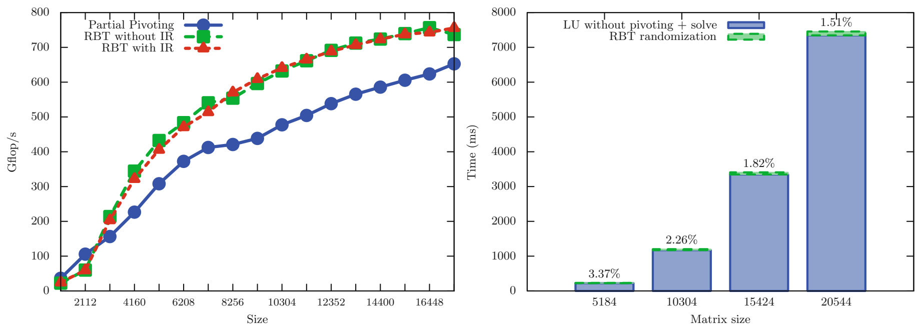
```
 1: for each thread block of size $bsize.x \times bsize.y$ of coordinates $b.x$ and $b.y$ do
 2:   for each Thread of coordinates $t.x$ and $t.y$ in the block do
 3:     $idx \leftarrow b.x \times bsize.x + t.x$
 4:     $idy \leftarrow b.y \times bsize.y + t.y$
 5:     if $idx < n/2$ and $idy < n/2$ then
 6:       Declare 4 shared memory arrays : $U_1[bsize.x]$, $U_2[bsize.x]$, $V_1[bsize.y]$,
          $V_2[bsize.y]$
 7:       $U_1(t.x) \leftarrow U(idx)$
 8:       $U_2(t.x) \leftarrow U(idx + n/2)$
 9:       $V_1(t.y) \leftarrow V(idy)$
10:       $V_2(t.y) \leftarrow V(idy + n/2)$
11:       Synchronize the threads in the block
12:       $a_{00} \leftarrow A(idx, idy)$
13:       $a_{01} \leftarrow A(idx, idy + n/2)$
14:       $a_{10} \leftarrow A(idx + n/2, idy)$
15:       $a_{11} \leftarrow A(idx + n/2, idy + n/2)$
16:       $b_1 \leftarrow a_{00} + a_{01}$
17:       $b_2 \leftarrow a_{10} + a_{11}$
18:       $b_3 \leftarrow a_{00} - a_{01}$
19:       $b_4 \leftarrow a_{10} - a_{11}$
20:       $A(idx, idy) \leftarrow U_1(t.x) \times V_1(t.y) \times (b_1 + b_2)$
21:       $A(idx, idy + n/2) \leftarrow U_1(t.x) \times V_2(t.y) \times (b_3 + b_4)$
22:       $A(idx + n/2, idy) \leftarrow U_2(t.x) \times V_1(t.y) \times (b_1 - b_2)$
23:       $A(idx + n/2, idy + n/2) \leftarrow U_2(t.x) \times V_2(t.y) \times (b_3 - b_4)$
24:     end if
25:   end for
26: end for
```

GPU and requires $\mathcal{O}(n^2)$ extra floating point operations, which is a low order term in our case and has no significant impact on the performance of our RBT solver.

In Fig. 2b, we can see that the time required to perform the randomization is less than 4% of the computational time for small matrices and becomes less than 2% for larger matrices. This is due to the low computational cost of the randomization ($8n^2$ flops) and to our optimized implementation that use the capabilities of the GPU accelerator.

4 RBT for Intel Xeon Phi

Similarly to the previous section, we present our implementation of the RBT on an Intel Xeon Phi coprocessor and discuss its performance. This solver and all the required routines (randomization, no pivoting LU factorization, iterative refinement) are part of the MAGMA MIC library (version 1.3).



(a) Performance. (b) Time breakdown.

Fig. 2. Randomized LU-based solver on GPU

4.1 Implementation

Algorithm 3 presents the randomization routine, using depth-two butterfly matrices. It is similar to its GPU counterpart, except that there are no blocks or threads to deal with inside this routine.

Algorithm 3. Two-level randomization on Intel Xeon Phi

```
Require: $A$ a pointer to the matrix $A$ on the Phi.
Require: $U$ a pointer to the matrix $U$ stored as a vector on the Phi.
Require: $V$ a pointer to the matrix $V$ stored as a vector on the Phi.
Require: $n$ the size of the matrix $A$
Ensure: $A \leftarrow U^T AV$
1: Call : Elementary Multiplication Phi($A$, &$U(n)$, &$V(n)$, $n/2$)
2: Call : Elementary Multiplication Phi(&$A(0, n/2)$, &$U(n)$, &$V(n + n/2)$, $n/2$)
3: Call : Elementary Multiplication Phi(&$A(n/2, 0)$, &$U(n + n/2)$, &$V(n)$, $n/2$)
4: Call : Elementary Multiplication Phi(&$A(n/2, n/2)$, &$U(n + n/2)$, &$V(n + n/2)$,
   $n/2$)
5: Call : Elementary Multiplication Phi($A$, $U$, $V$, $n$) {Applying level one recursion}
```

The `Elementary multiplication Phi` kernel, described in Algorithm 4, uses SIMD instructions [5] to improve the performance of each core, and OpenMP to handle thread parallelism between cores. This algorithm is well adapted to the SIMD programming model as the dependencies between the data are separated by a large number of values. In Algorithm 4, we use double precision floating point numbers, each of them using 64 bits. This explains why 8 values are stored in each 512-bits SIMD vector. When using 32 bits reals, 16 values are stored in each vector. For complex numbers, 8 numbers are stored in single precision and 4 in double. We take advantage of the SIMD capabilities of the Intel Xeon Phi coprocessor by using the low level Knight's Corner intrinsics set

Algorithm 4. Phi Kernel: Elementary multiplication Phi(A, U, V, n)

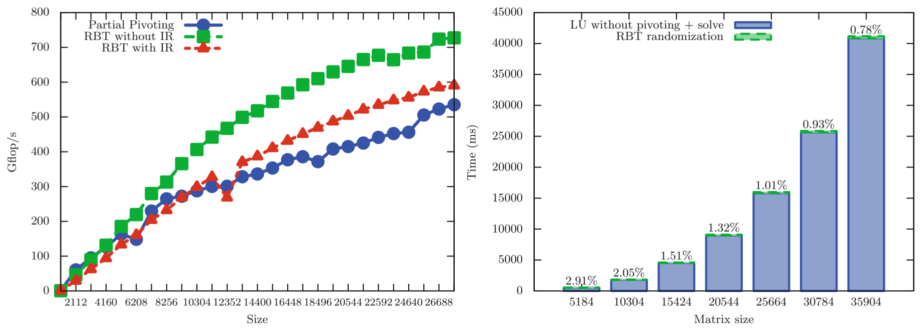
```
1: OpenMP parallel for
2: for $i = 0$ to $n/2$ do
3:    Declare $V_1$ and $V_2$ two 512-bit vector registers.
4:    Set all values of $V_1$ with $V(i)$
5:    Set all values of $V_2$ with $V(i + n/2)$
6:    for $j = 0$ to $n/2$ step 8 do
7:       Declare $a_{00}$, $a_{01}$, $a_{10}$ and $a_{11}$ four 512-bit vector registers.
8:       LOAD 8 values from $A(i, j)$ in $a_{00}$
9:       LOAD 8 values from $A(i, j + n/2)$ in $a_{01}$
10:      LOAD 8 values from $A(i + n/2, j)$ in $a_{10}$
11:      LOAD 8 values from $A(i + n/2, j + n/2)$ in $a_{11}$
12:      Declare $b_1$, $b_2$, $b_3$ and $b_4$ four 512-bit vector registers.
13:      $b_1 \leftarrow \mathbf{ADD}(a_{00}, a_{01})$
14:      $b_2 \leftarrow \mathbf{ADD}(a_{10}, a_{11})$
15:      $b_3 \leftarrow \mathbf{SUB}(a_{00}, a_{01})$
16:      $b_4 \leftarrow \mathbf{SUB}(a_{10}, a_{11})$
17:      Declare $U_1$ and $U_2$ two 512-bit vector registers.
18:      LOAD 8 values from $U(j)$ in $U_1$
19:      LOAD 8 values from $U(j + n/2)$ in $U_2$
20:      $a_{00} \leftarrow \mathbf{MUL}(U_1, \mathbf{MUL}(V_1, \mathbf{ADD}(b_1, b_2)))$
21:      $a_{01} \leftarrow \mathbf{MUL}(U_1, \mathbf{MUL}(V_2, \mathbf{ADD}(b_3, b_4)))$
22:      $a_{10} \leftarrow \mathbf{MUL}(U_2, \mathbf{MUL}(V_1, \mathbf{SUB}(b_1, b_2)))$
23:      $a_{11} \leftarrow \mathbf{MUL}(U_2, \mathbf{MUL}(V_2, \mathbf{SUB}(b_3, b_4)))$
24:      STORE 8 values from $a_{00}$ at $A(i, j)$
25:      STORE 8 values from $a_{01}$ at $A(i, j + n/2)$
26:      STORE 8 values from $a_{10}$ at $A(i + n/2, j)$
27:      STORE 8 values from $a_{11}$ at $A(i + n/2, j + n/2)$
28:    end for
29: end for
```

of instructions [10,11]. The use of the intrinsics allows the use of the assembly SIMD instructions with C style functions.

4.2 Performance Results

Here we present the performance results of our solver. The experiments were carried out using the same multicore host as described in Sect. 3.2 (two Intel Xeon X5680) with an Intel Xeon Phi coprocessor 7120 with 61 cores running at 1.238 GHz, with 16 GB of memory. The cores have 30.5 MB of combined L2 cache memory. We mention that each core can manage 4 threads by using hyper-threading. For the experiments, we use 240 threads in total. Note that we were able to perform tests on larger matrices compared to the GPU version. This is due to the larger size of the Intel Xeon Phi memory.

In Fig. 3a, we notice that the Intel Xeon Phi version is up to 50 % faster than the solver using partial pivoting without iterative refinement, and only 25 % faster with iterative refinement, which is not yet optimized for Intel Xeon Phi.



(a) Performance.

(b) Time breakdown.

Fig. 3. Randomized LU-based solver on Intel Xeon Phi

In Fig. 3b, we observe that the randomization requires less than 3 % of the total time and even less than 1 % for larger matrices. We recall that the randomization performed on the Intel Xeon Phi has been optimized using SIMD instructions and OpenMP.

5 Conclusion

In this paper, we have presented two implementations of the RBT solver using accelerators based respectively on GPU and Intel Xeon Phi, resulting in routines that are significantly faster than the reference solver based on the LU factorization with partial pivoting. Thanks to an efficient implementation of the randomization, the overhead for randomizing the original system is negligible compared to the computational cost of the whole solver. Ongoing work include optimizing the iterative refinement on Intel Xeon Phi and solving multiple small systems at the same time using batched solvers [7].

Acknowledgments. The authors would like to thank Stanimire Tomov and Ichitaro Yamasaki from University of Tennessee for the support with the MAGMA library.

References

1. Baboulin, M., Becker, D., Bosilca, G., Danalis, A., Dongarra, J.: An efficient distributed randomized algorithm for solving large dense symmetric indefinite linear systems. Parallel Comput. **40**(7), 213–223 (2014)
2. Baboulin, M., Becker, D., Dongarra, J.: A parallel tiled solver for dense symmetric indefinite systems on multicore architectures. In: 2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS), pp. 14–24. IEEE (2012)
3. Baboulin, M., Dongarra, J., Herrmann, J., Tomov, S.: Accelerating linear system solutions using randomization techniques. ACM Trans. Math. Softw. **39**(2), 1–13 (2013)

4. Baboulin, M., Dongarra, J., Tomov, S.: Some issues in dense linear algebra for multicore and special purpose architectures. In: 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA'08). Lecture Notes in Computer Science, vol. 6126–6127. Springer-Verlag (2008)
5. Diefendorff, K., Dubey, P.K., Hochsprung, R., Scale, H.: Altivec extension to PowerPC accelerates media processing. IEEE Micro. **20**(2), 85–95 (2000)
6. Grigori, L., Demmel, J.W., Xiang, H.: CALU: a communication optimal LU factorization algorithm. SIAM J. Matrix Anal. Appl. **32**(4), 1317–1350 (2011)
7. Haidar, A., Dong, T., Luszczek, P., Tomov, S., Dongarra, J.: Batched matrix computations on hardware accelerators based on GPUs. IJHPCA 29(2), 193–208 (2015). http://dx.doi.org/10.1177/1094342014567546
8. Haidar, A., Luszczek, P., Tomov, S., Dongarra, J.: Heterogenous acceleration for linear algebra in multi-coprocessor environments. In: Daydé, M., Marques, O., Nakajima, K. (eds.) VECPAR 2014. LNCS, vol. 8969, pp. 31–42. Springer, Heidelberg (2015)
9. Intel: Math Kernel Library (MKL). http://www.intel.com/software/products/mkl/
10. Intel: Intel Xeon Phi$^{\mathrm{TM}}$ Coprocessor System Software Developers Guide (2012). http://software.intel.com/en-us/articles/
11. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High-Performance Programming. Morgan Kaufmann, Newnes (2013)
12. Khabou, A., Demmel, J.W., Grigori, L., Gu, M.: Lu factorization with panel rank revealing pivoting and its communication avoiding version. SIAM J. Matrix Anal. Appl. **34**(3), 1401–1429 (2013)
13. Nvidia, C.: Compute Unified Device Architecture programming guide (2007)
14. Parker, D.S.: Random butterfly transformations with applications in computational linear algebra. Technical Report CSD-950023, Computer Science Department, UCLA (1995)
15. Parker, D.S., Pierce, B.: The randomizing FFT: an aternative to pivoting in Gaussian elimination. Technical Report CSD-950037, Computer Science Department, UCLA (1995)
16. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput. **36**(5&6), 232–240 (2010)