RC24599 (W0807-045) July 8, 2008
Mathematics

IBM Research Report

The Relevance of New Data Structure Approaches for Dense Linear Algebra in the New Multicore/Manycore Environments

Fred G. Gustavson
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

The Relevance of New Data Structure Approaches for Dense Linear Algebra in the new Multicore / Manycore Environments

Fred G. Gustavson

IBM T.J. Watson Research Center,
Yorktown Heights, NY 10598, USA
email: fg2@us.ibm.com

Abstract. For over ten years now, Bo Kågström's Group in Umea, Sweden, Jerzy Waśniewski's Team at Danish Technical University in Lyngby, Denmark, and I at IBM Research in Yorktown Heights have been applying recursion and new data structures to increase the performance of Dense Linear Algebra (DLA) factorization algorithms. Later, John Gunnels, and later still, Jim Sexton, both now at IBM Research began working in this area. For about three years now almost all computer manufacturers have dramatically changed their computer architectures which they call Multicore (MC). The traditional designs of DLA libraries such as LAPACK and ScaLAPACK perform poorly on MC. Recent results of Jack Dongarra's group at the Innovative Computing Laboratory in Knoxville, Tennessee have shown how to obtain high performance for DLA factorization algorithms on the Cell architecture, an example of an MC processor, but only when they used new data structures. We will give some reasons why this is so.
We also present new algorithms for Blocked In-Place Rectangular Transposition of an M by N matrix A. This work adds blocking to the work by Gustavson and Swirszcz presented at Para06 on scalar in-place transposition. We emphasize the importance of RB format and also provide efficient algorithms between RB format and standard column and row major formats of 2-D arrays in the Fortran and C languages. Performance results are given. From a practical point of view, this work is very important as it will allow existing codes using LAPACK and ScaLAPACK to remain usable by new versions of LAPACK and ScaLAPACK.

1 Introduction

Multicore/Manycore (MC) can be considered a revolution in Computing. In many of my papers, we have talked about the fundamental triangle of Algorithms, Architectures and Compilers [6] or the Algorithms and Architecture approach [1, 9]. MC represent a radical change in architecture design. The fundamental triangle concept says that all three areas are inter-related. This means Compilers and Algorithms must change and probably in significant ways. The LAPACK and ScaLAPACK projects under the direction of Jim Demmel and

Jack Dongarra started an enhancement of LAPACK and ScaLAPACK projects in late 2004. In 2006 Jack Dongarra started another project, called PLASMA, which was directed at the effect MC would have on LAPACK. His initial findings were that traditional BLAS based LAPACK would need substantial changes. So, what appeared at first to be enhancements of these libraries now appears to be directed at more basic structural changes.

The essence of MC is many cores on a single chip. The `Cell Broadband Engine`$^{\text{TM}}$ `Architecture` is an example. Cell is a heterogeneous chip consisting of a single traditional PPE (Power Processing Element) and 8 SPEs (Synergistic Processing Element) and a novel memory system interconnect. Each SPE core can be thought of as a processor and a "cache memory". Because of this, "cache blocking" is still very important. Cache blocking was first invented by my group at IBM in 1985 [13] and the Cedar project at the University of Illinois.

The advent of the Cray 2 was a reason for the introduction of Level 3 BLAS [5] followed by the introduction of the LAPACK [3] library in the early 1990's. Now, according to some preliminary results of the PLASMA project, this Level 3 BLAS approach is no longer adequate to produce high performance LAPACK codes for MC. Nonetheless, it can be argued that the broad idea of "cache blocking" is still mandatory as data in the form of matrix elements must be fed to the SPEs or more generally MC so they can be processed. And, equally important, is the arrangement in memory of the matrices that are being processed. So, this is what we will call "cache blocking" here.

We present new algorithms that require little or no extra storage to transpose a M by N rectangular (non-square) matrix A in-place. We make two assumptions on how these matrices are stored. First we assume that the matrices are stored in Rectangular Block (RB) format and we define how the M by N matrix A is represented in RB format. Square Block (SB) format is a special case of RB format when the rectangle is a square. It was first introduced in 1997, see [8], and further developed from 1999 on in [9] in conjunction with research by Bo Kagstrom's team at Umea University and Jerzy Wasniewski's team at Danish Technical University. A lot of this work culminated with a SIAM review article on recursive algorithms for DLA [6]. It turns out that our results on NDS [8, 9, 6, 2, 10] are very relevant to MC: Of all 2-D data layouts for common matrix operations SB format minimizes L1 and L2 cache misses as well as TLB misses. The essential reason for this is that a SB of order `NB` is also a contiguous 1-D array of size $\texttt{NB}^2$ and for almost all cache designs a contiguous array whose size is less than the cache size is mapped from its place in memory into the cache by the *identity* mapping. SB ormat is the same as block data layout. Block data layout is described in [15] and the authors show that this format leads to minimal L1, L2, TLB misses for matrix operations that treat rows and columns equally.

RB format has a number of other advantages. A major one is that it naturally partitions a matrix to be a matrix of sub-matrices. This allows one to view matrix transposition of a M by N matrix A where $M = m\texttt{MB}$ and $N = n\texttt{NB}$ as a block transposition of a much smaller m by n block matrix A. However, usually M and N are not multiples of `MB` and `NB`. So, RB format as we define it here, would pad the rows and columns of A so that M and N become multiples of some blocking factors `MB` and `NB`. The second format is the standard 2-D array format of the Fortran and C programming languages. This new work on in-place transformations between standard matrix formats for DLA and NDS was quite illuminating to us as it showed us why in a fundamental way why NDS are superior to the current standard data structures of DLA. From a practical point of view, this work is very important as it will allow existing codes using LAPACK and ScaLAPACK to remain usable by new versions of LAPACK and ScaLAPACK. This last aspect is the so called migration problem where existing past software is using old LAPACK or ScaLAPACK routines. By transforming the old standard API's of LAPACK and ScaLAPACK in-place to the API's that new LAPACK or ScaLAPACK will use the old existing software can be made to perform better in their LAPACK or ScaLAPACK sections of code.

"Cache blocking" will be described in Section 2. We show there how it can be automatically incorporated into dense linear algebra factorization algorithms (DLAFA) by introducing NDS in concert with using kernel routines instead of using Level 3 BLAS. In Section 3, we describe the features of In-Place Transformation between standard full layouts of matrices and the new rectangular block (RB) or square block (SB) formats of NDS. Performance results are given in Section 4 followed by a Summary and Conclusions in Section 5.

2 Cache Blocking

Here we address "cache blocking" as it relates to DLAFA. It will briefly repeat some of my early work on this subject and will sketch a proof that DLAFA could be viewed as just doing matrix multiplication (MM) by adopting the linear transformation approach of applying equivalence transformations to a set of linear equations $Ax = b$ to produce an equivalent (simpler) form of these equations $Cx = d$. Examples of C are $LU = PA$, for Gaussian elimination, $LL^T = A$, for Cholesky Factorization, and $QR = A$, for Householder's factorization. We adopt this view to show a general way to produce a whole collection of DLAFA as opposed to the commonly accepted way of describing the same collection as a set of distinct algorithms [7]. A second reason was to indicate that for each linear transformation we perform we are invoking the definition of MM. Here is the gist of the proof as it applies to $LU = PA$.

1. Perform $n = \lceil N/\texttt{NB} \rceil$ rank `NB` linear transformations on A to get U.
2. Each of these n composed `NB` linear transformations is MM by definition.
3. By the principle of equivalence we have $Ax = b$ if and only if $Ux = L^{-1}Pb$.

MM clearly involves "cache blocking". Around the mid 1990's we noticed, see page 739 of [8], that the API for Level 3 BLAS `GEMM` could hurt performance. In fact, this 1-D API is also the API for 2-D arrays in Fortran and C. An explanation of dimension is given in [16]. One can prove that it is impossible to layout a matrix in 1-D fashion and maintain closeness of its elements. LAPACK and ScaLAPACK also use this API for full arrays. On the other hand, high performance implementations of GEMM do not use this API as doing so would lead to sub-optimal performance. In fact, some amount of data copy is usually done by most high performance GEMM implementations. Now, Level 3 BLAS are called multiple times by DLAFA. This means that multiple data copy will usually occur in DLAFA that are use standard Level 3 BLAS. The NDS for full matrices are good for GEMM. Their layouts are essentially 2-D. DLAFA algorithms can be expressed in terms of scalar elements $a_{i,j}$ which are one by one block matrices. Alternatively, they can be expressed in terms of partitioned submatrices, $A(I : I+\texttt{NB}-1, J : J : \texttt{NB}-1)$ of order NB. See [7] for a definition of colon notation. The algorithms are almost identical. However, the latter description automatically incorporates "cache blocking" into a DLAFA. Take the scalar statement $c_{i,j} = c_{i,j} - a_{i,k}b_{k,j}$ representing MM as a fused multiply-add. The corresponding statement for partitioned submatrices becomes a kernel routine for Level 3 BLAS GEMM. However, it is imperative to store the order NB SB's as contiguous blocks of matrix data, as this is what many Level 3 BLAS GEMM implementations do internally. We remark that this is not possible with the standard Fortran and C API. This fact emphasizes the importance of storing the submatrices of DLAFA as contiguous blocks of storage. An essence of NDS for full matrices is to store their submatrices as contiguous blocks of storage. The simple format of full NDS has each RB or SB as being in standard column major (CM) or standard row major (RM) format; see [9] for more details.

Recent results of the PLASMA project as it related to the Linpack benchmark $LU = PA$ when running on the Cell processor emphasize their use of SB format. According to Dongarra's Team it was crucial that NDS be used as the matrix format. In particular, using the standard API of Fortran and C did not yield good performance results. Also, earlier results obtained by considering the IBM new Blue Gene/L computers [4] emphasize the same thing. However, the simple format of full NDS needs to be re-arranged internally to take into account "cache blocking" for the L0 cache. The L0 cache is a new term defined in [10] and it refers to the register file of the FPU that is attached to the L1 cache. Full details are given in [10].

3 Inplace Transposition between Standard Full Layouts and RB Format

The algorithms in [11], although fast, or very fast compared to the existing algorithms for the same problem, are relatively slow on today's processors. This is because A in [11] is viewed as a permutation P of length $q = MN - 1$ and A is moved in its array A on top of itself randomly one element at a time. Today's processors layout memories in chunks of size LS called lines and when an element is accessed the entire line containing the element is brought into the L1 cache. To obtain high performance it is therefore imperative to utilize or process all elements in a line once the element enters the L1 and L0 caches. The L0 cache is the register file of a processor; eg, see [10]. However, for the algorithms in [11] this is impossible because the P governing in-place transposition of A is essentially random for almost all M and N when $M \neq N$. We speculate that the reason in-place transposition has not been used for DLA algorithms is because one can prove that in-place transposition is impossible for sub-matrices of a matrix A stored in standard format. Also, these algorithms are slow relative to out-of-place transposition algorithms which are almost universally used instead.

Usually, one uses SB format. Here we use its generalization, RB format. It will be evident that our results also hold for SB format. In the RB format version of our new Block In-Place Xpose (`BIPX`) algorithm our M by N matrix A, usually padded, can be considered a m by n block matrix where each submatrix has $r = \texttt{MB}$ rows and $s = \texttt{NB}$ columns. Padding occurs when either $M < mr$ or $N < ns$. Now the governing permutation P has length $q = mn - 1$. So, each element moved is a submatrix of size `MB` by `NB` whose elements are contiguous and hence consist of $\lceil rs/\texttt{LS} \rceil$ contiguous lines. Hence, the problem of the previous paragraph disappears and our `BIPX` algorithm will perform about the same as current out-of-place algorithms. In this case one, the `BIPX` algorithm has one stage and hence it is more efficient than the other two cases which we now describe.

Standard Fortran and C two dimensional arrays are the other format we consider for our in-place transpose algorithms. We need to describe how one can move from a standard CM or RM format to a RB format. This is done by using a vector version of the `IPX` or `MIPT` algorithm [11] , the `VIPX` algorithm, which has similar features to our `BIPX` algorithm. Our `VIPX` algorithm maps in-place a mr by `NB` submatrix of A in standard CM format with $\texttt{LDA} = m\texttt{MB}$ to a RB format matrix consisting of m RB concatenated together each of size `MB` rows by `NB` columns. This submatrix of A is called a column swath of A. Repeating algorithm `VIPX` n times on the n concatenated column swaths that make up CM A converts CM A to RB format A. Then, algorithm `BIPX` computes RB A^T. Next, the inverse of the `VIPX` algorithm, applied m times on the m column swaths of A^T, computes CM A^T. This algorithm is called case two. It assumes CM A has a certain layout space in terms of standard 2-D layout terminology. Here, CM A and RB format A will be $M \leq mr$ by $N \leq ns$ matrices with $\texttt{LDA} = m\texttt{MB}$ where $m = \lceil M/\texttt{MB} \rceil$. Also, the array `A` holding CM A and RB format A will have space for $mnrs$ elements where $n = \lceil N/\texttt{NB} \rceil$. It can be seen that the case two in-place transpose algorithm has three stages consisting of two `VIPX` algorithm stages and one `BIPX` algorithm stage. Hence, it will be less efficient than case one by approximately a factor of three.

Clearly, matrix A rarely has its M a multiple of `MB` and its N a multiple of `NB` or A is contained in an array `A` whose size is equal to $mnrs$ elements. This third case is where A is in standard format and has CM A being M by N with $\texttt{LDA} \geq M$ and case two is not holding. In case three, we set $m_1 = \lfloor M/\texttt{MB} \rfloor$ and $n_1 = \lfloor N/\texttt{NB} \rfloor$ to define the space for a $M_1 = m_1\texttt{MB}$ by $N_1 = n_1\texttt{NB}$ smaller A_1 submatrix of A inside the original array space of A. This requires that we save the leftover $M - M_1$ rows and $N - N_1$ columns of A in a buffer. We fill this buffer using out-of-place transpose operations on the these leftover rows and columns of A. Then we move the M_1 by N_1 matrix A_1 to $\mathtt{A}(0:\mathtt{M}_1-1, 0:\mathtt{N}_1-1)$ of its array space `A` using standard move operations. Note that matrix A_1 is now in standard CM format with `LDA` $= M_1$ and is a case two matrix. We apply the case two algorithm to A_1 to get CM A_1^T. Next, we expand A_1^T in the array space of `A` using standard move operations thereby making "holes" in array `A` for the submatrices of A in the buffer. Finally, we transfer the buffer with the its saved leftover rows and columns to the "holes" in `A` using standard out-of-place transpose and copy algorithms to get the final CM A^T matrix. The case three algorithm contains four additional stages of save $A - A_1$, contract A_1, expand A_1 and restore $A - A_1$ over the case two algorithm and hence is the least efficient of the three algorithms. Since, it passes over A about five times it will perform about five times slower than the case one algorithm when A is large; see [12].

We only describe the `BIPX`, `VIPX` and case two algorithms here.

There is some literature on this subject in the form of a patent disclosure [14] which we discovered after we finished this work. This disclosure is incomplete, and furthermore its algorithms are not really in-place.

3.1 The `VIPX(MB,m,NB,A)` Column Swath Algorithm

We briefly describe how one gets from standard CM format to RB format. Let $A1$ have $M = mr$ rows and $N = ns$ columns. Let $A1$ have its `LDA` $= M$. Thus, $A1$ consists of n column swaths that are concatenated together. Denote any such swath as a submatrix $A3$ and note that $A3$ consists of `NB` contiguous columns of CM matrix $A1$. So, $A3$ has M rows and $s =$ `NB` columns. Think of $A3$ as an m by NB matrix whose elements are column vectors of length $r =$ `MB`. Now apply algorithm `MIPT` or `IPT` of [11] to this m by s matrix $A3$ of vectors of length r. The result is that now $A3$ has been replaced (over-written) as $A3^T$ which is a size s by m matrix of vectors of length r. It turns out, as a little reflection will indicate, that $A3^T$ can also be viewed as consisting of m RB matrices of size r by s concatenated together. For $A1$ one can do n parallel operations for each of the $n = N/s$ different submatrices $A3$ of $A1$. After completion of these n parallel steps one has transformed CM $A1$ in-place to be matrix $A2$ as a RB matrix consisting of m block rows by n block columns stored in standard CM block order. Of course, $A1$ and $A2$ are different representations of the same matrix. The `VIPX` algorithm here is either algorithm `MIPT` or `IPT` of [11] modified to move in-place vectors of length r as opposed of scalars of length one.

3.2 The `BIPX(MB,NB,m,n,A)` Block Transpose Algorithm

We briefly describe how one gets from RB format to the transpose of RB format in-place. Let $A2$ have m block rows and n block columns where each block element is a standard CM matrix having `MB` rows, `NB` columns and `LDA=MB`. These mn block matrices are laid out in standard CM block order. Now apply algorithm `MIPT` or `IPT` of [11] to this m by n matrix $A2$ of RB matrices. The result is that now $A2$ has been replaced (over-written) by an n by m matrix $A2^T$ of RB matrices of size `NB` rows by `MB` columns. Each block element is a standard

CM matrix having NB rows, MB columns and LDA=NB; ie, each new RB matrix is the transpose of an old RB matrix. The BIPX algorithm is either algorithm MIPT or IPT of [11] modified to move in-place RB standard CM matrices of size MB rows by NB columns to be transposes of these RB matrices according to a permutation cycle of the BIPX algorithm.

3.3 The Case Two In-Place Transpose Algorithm

The case two Algorithm was described in the previous section 3. When $M = N$ one calls a standard inplace transpose algorithm. We present the $M \neq N$ case now:

```
m1=m/nb ! A is a tectangular m by n matrix where m ^= n
n1=n/nb ! A will become a SB matrix of size m1 by n1
nb2=nb*nb ! each SB holds nb^2 matrix elements
if(m1.gt.1)then ! Stage 1 of CM to SB
  call VIPX1(nb,m1,nb,A,temp,L,nL)
  do k=1,n1-1
    call VIPX2(nb,m1,nb,A(k*m*nb),temp,L,nL)
  enddo
endif
if(m1.eq.1.or.n1.eq.1)then ! A is a block vector matrix
  if(m1.eq.1)then ! A is a 1 by n1 block vector; xpose n1 blocks
    do i=0,n1-1
      call DXPI(A(i*nb2),nb,nb)
    enddo
  else ! n1 = 1; A is a m1 by 1 block vector; xpose m1 blocks
    do i=0,m1-1
      call DXPI(A(i*nb2),nb,nb)
    enddo
  endif
else ! A is a SB matrix of size m1 by n1
  call BIPX(m,n,A,m,nb,temp) ! Stage 2 of SB to SB^T
endif
if(n1.gt.1)then ! Stage 3 of SB^T to CM
  call VIPX1(nb,nb,n1,A,temp,L,nL)
  do k=1,m1-1
    call VIPX2(nb,nb,n1,A(k*n*nb),temp,L,nL)
  enddo
endif
```

We have broken stages one and three into an initial call to VIPX1 and the remaining calls to VIPX2. VIPX1 is VIPX of Section 3.1 where we save the leaders [11] of VIPX in vector L of length NL. Hence, further calls to VIPX can be handled by the more efficient VIPX2 which receives its leaders in L as input. Routine DXPI is a simple vanilla inplace transpose routine. Array temp will hold a vector of length NB or NB^2. DGETMI is the ESSL inplace transpose routine [13].

4 Performance Studies of VIPX,BIPX versus ACM Algorithm 467 and ESSL DGETMO

We assume case two of the previous Section. Our routines VIPX,BIPX are modifications routine IPT. Hence our results could be slightly better as MIPT is a more efficient routine. In the case two algorithm we pass through the matrix A three times while ACM Algorithm 467 (Brenner's Algorithm) and the ESSL DGETMO (Double Precision GEneral Transpose Matrix Out-of-place) Algorithm passes through A only once. Hence, one might expect that the new Algorithm would perform about three times slower than ACM Algorithm 467 and ESSL's DGETMO. We will see this false for ACM Algorithm 467 and roughly true for DGETMO. 210 matrices of row and column sizes from 50 to 1000 in steps of 50 were generated. This was our test set. We excluded square matrices as they are easy to transpose in place. Hence our test set was 19*20/2 = 190 matrices. We let MB=NB so we dealt with SB. We wanted to see the effect of choosing the block size NB. Thus, for matrices whose sizes were multiples of 100 we let NB=100. Otherwise, we let NB=50.

In the first experiment, run on a IBM Power 5 and IBM 604E, ACM Algorithm 467 is compared to Algorithms VIPX and BIPX. We made a single run where IWORK = 0; see [11]. This means ACM Algorithm 467 was truly in place. Our performance results as functions of m and n are essentially random for ACM Algorithm 467. To first approximation VIPX and BIPX have equal execution times. Because of space limitations and time considerations we can only summarize our results. On the IBM 604E, the new three pass Algorithm was faster except for one matrix where the time ratio was .859. For (68, 115, 4, 2) test matrices it was between (1:2, 2:3, 3:4, 4:5) times faster respectively. We remark that IBM604E was not a scientific machine and that it has a slow memory system. Also, its line size of 4 double words is four time smaller than the Power 5 line size. The performance numbers really should be multiplied by three for a fairer comparison. On IBM Power 5, the new three pass Algorithm was always faster. For (144, 39, 4, 2, 1) matrices it was (1:5, 5:10, 10:15, 15:20, 25) times faster respectively. These Power 5 numbers are even more impressive when these ratios are multiplied by three. The following integer vector v of length 25 gives the complete breakdown. Its i-th component gives the number of matrices whose time ratio, (three pass) / (ACM Algorithm 467), was greater equal i but less than i+1. Here is the vector: v = 11 49 25 29 30 19 12 4 0 4 0 3 0 1 0 0 0 2 0 0 0 0 0 0 1. The test matrix of size 1000 100 corresponded to $v(25) = 1$. The performance ratio was 25.434, the time for ACM Algorithm 463 was 9411 μ-secs and the time for three pass algorithm was 370 μ-secs.

We now discuss the second experiment. We compare ESSL DGETMO routine versus Algorithms VIPX and BIPX. We consider first the IBM 604E platform. The BIPX Algorithm is faster than DGETMO in all 190 cases. For (3 10 19 31 66 38 15 4 1 2 1) cases it was between (6:10, 10:20, 20:30, 30:40, 40:50, 50:60, 60:70, 70:80, 80:90, 90:100, 101)% faster. The VIPX Algorithm runs slighty slower than the BIPX Algorithm. The value of NB being 100 instead of 50 has little effect on both the VIPX and BIPX Algorithms. The percent time ratios of DGETMO / (

three stage) is (38, 40:50, 50:60, 60:70, 70:80, 80:85)% for the (1, 150, 12, 15, 9, 3) cases. Secondly, for the Power 5 platform, the BIPX Algorithm is faster than the DGETMO routine in 102 of the 190 cases. For (22, 37, 18, 17, 8) cases BIPX was between (0:10, 10:20, 30:40, 40:49)% faster than DGETMO. For (35, 35, 17, 1) cases DGETMO was between (0:10, 10:20, 20:30, 32)% faster than BIPX. The performance of the VIPX Algorithm is usually slower by about 20% than the BIPX Algorithm when $\mathtt{NB} = 100$. For $\mathtt{NB} = 50$ this amount is about 50%. Also, the BIPX Algorithm is slower by about 10% when $\mathtt{NB} = 50$ instead of being 100. And VIPX is slower by about 33% when $\mathtt{NB} = 50$ instead of being 100. The memory system of Power 5 is much better than the memory system of 604E. I believe this fact accounts for what we observed above about VIPX and BIPX regarding their performances resulting from the use of a larger NB value. The reason is that larger NB better supports use of automatic streaming or prefetching which is a feature of Power 5. In this regard, a reason for BIPX better performance over VIPX is that BIPX moves larger chunks of memory per invocation; its block length is $\mathtt{NB}^2$ whereas the vector length of VIPX is NB. The percent time ratios of DGETMO / (three stage) is (18:20, 20:30, 30:40, 40:49)% for the (7, 101, 75, 7) cases.

5 Conclusions and Summary

We showed that DLAFA are mainly MM algorithms. The standard API for matrices use arrays. All array layouts are *one* dimensional. It is *impossible* to maintain locality of reference in a matrix or any higher than 1-D object using a 1-D layout; see [16]. MM requires row and column operations and thus requires matrix transposition (MT). Our results on inplace MT show that performance suffers greatly if one uses a 1-D layout. Using NDS for matrices approximates a 2-D layout; thus, one can dramatically improve inplace MT performance. Our message is that DLAFA are all MM. MM requires MT and both require NDS. Thus, DLAFA can and do perform well on Cell if one uses NDS.

References

1. R. C. Agarwal, F. G. Gustavson, M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, Vol. 38, No. 5, Sep. 1994, pp. 563,576.
2. B. S. Andersen, J. A. Gunnels, F. G. Gustavson, J. K. Reid and J. Waśniewski. A Fully Portable High Performance Minimal Storage Hybrid Cholesky Algorithm. *ACM TOMS*, Vol. 31, No. 2 June 2005, pp. 201-227.
3. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide Release 3.0*, SIAM, Philadelphia, 1999.
4. S. Chatterjee et. al. Design and Exploitation of a High-performance SIMD Floating-point Unit for Blue Gene/L. *IBM Journal of Research and Development*, Vol. 49, No. 2-3, March-May 2005, pp. 377-391.

5. J. J. Dongarra and J. Du Croz, S. Hammarling and I. Duff. A Set of Level 3 Basic Linear Algebra Subprograms, *TOMS*, Vol. 16, No. 1, Mar. 1990, pp. 1-17.
6. E. Elmroth, F. G. Gustavson, I. Jonsson, and B. Kågström. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. *SIAM Review*, Vol. 46, No. 1, Mar. 2004, pp. 3,45.
7. G. Golub, C. VanLoan. Matrix Computations. Book, *John Hopkins Press*, Baltimore and London, 3rd. 1996.
8. F. G. Gustavson. Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. *IBM Journal of Research and Development*, Vol. 41, No. 6, Nov. 1997, pp. 737,755.
9. F. G. Gustavson High Performance Linear Algebra Algorithms using New Generalized Data Structures for Matrices. *IBM Journal of Research and Development*, Vol. 47, No. 1, Jan. 2003, pp. 31,55.
10. F. G. Gustavson, J. Gunnels, J. Sexton. Minimal Data Copy For Dense Linear Algebra Factorization. *Computational Science - Para 2006*, B. Kågström, E. Elmroth, J. Dongarra, J. Waśniewski eds., Lecture Notes in Computer Science 4699. Springer-Verlag, pp. 540-549, 2007.
11. F. G. Gustavson, T. Swirszcz. In-Place Transposition of Rectangular Matrices. *Computational Science - Para 2006*, B. Kågström, E. Elmroth, J. Dongarra, J. Waśniewski eds., Lecture Notes in Computer Science 4699. Springer-Verlag, pp. 560-569, 2007.
12. F. G. Gustavson, J. Gunnels, J. Sexton. Method and Structure for Fast In-Place Transformation OF Standard Full and Packed Matrix Data Formats. *United State Patent Office Submission YOR920070021US1 and Submission YOR920070021US1(YOR.699CIP)* US Patent Office, 35 pages, Sep. 1, 2007, 58 pages March 2008.
13. IBM. IBM Engineering and Scientific Subroutine Library for AIX Version 3, Release 3. *IBM Pub. No. SA22-7272-00* Feb. 1986.
14. S. Lao, B. R. Lewis, M. L. Boucher In-place Transpose *United State Patent No. US 7,031,994 B2.* US Patent Office. Apr. 18, 2006
15. N. Park, B. Hong, V. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Trans. Parallel and Distributed Systems*, 14(7):640-654, 2003.
16. H. Tietze. Three Dimensions–Higher Dimensions. *Famous Problems of Mathematics*, book, 1965, Graylock Press, pp. 106-120.