**RC24131 (W0612-014) December 5, 2006**
**Computer Science**

# IBM Research Report

## Minimal Data Copy for Dense Linear Algebra Factorization

**Fred G. Gustavson, John A. Gunnels, James C. Sexton**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

**Research Division**
**Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich**

**LIMITED DISTRIBUTION NOTICE**: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

# Minimal Data Copy For Dense Linear Algebra Factorization

Fred G. Gustavson, John A. Gunnels and James C. Sexton

IBM's T. J. Watson Research Center, Yorktown Heights, NY 10598, USA,
(fg2,gunnels,sextonjc)@us.ibm.com

**Abstract.** The full format data structures of Dense Linear Algebra hurt the performance of its factorization algorithms. Full format rectangular matrices are the input and output of level the 3 BLAS. It follows that the LAPACK and Level 3 BLAS approach has a basic performance flaw. We describe a *new* result that shows that representing a matrix $A$ as a collection of square blocks will reduce the amount of data reformating required by dense linear algebra factorization algorithms from $O(n^3)$ to $O(n^2)$. On an IBM Power3 processor our implementation of Cholesky factorization achieves 92% of peak performance whereas conventional full format LAPACK `DPOTRF` achieves 77% of peak performance. All programming for our new data structures may be accomplished in standard Fortran, through the use of higher dimensional full format arrays. Thus, new compiler support may not be necessary. We also discuss the role of concatenating submatrices to facilitate hardware streaming. Finally, we discuss a *new* concept which we call the L1 / L0 cache interface.

## 1 Introduction

The current most commonly used Dense Linear Algebra (DLA) algorithms for serial and SMP processors have a performance inefficiency and hence they give sub-optimal performance. We indicate that Fortran and C two dimensional arrays are the main reason for the inefficiency. We show how to correct these performance inefficiencies by using New Data Structures (NDS) along with so-called kernel routines. These NDS generalize the current storage layouts for both the Fortran and C programming languages. One of these formats is packed format and we do not discuss it as a new result [18, 13, 19] about Rectangular Full Packed (RFP) format shows that packed format can be represented by RFP format. RFP format is full format and it and packed both use exactly the same amount of storage. However, SBP (Square Block Packed) format also replaces packed format and it is a main subject of this paper. Like RFP format it is a full format data structure and it uses only slightly more storage than RFP format.

The BLAS [22, 9, 10] (Basic Linear Algebra Subroutines) were introduced to make the algorithms of DLA performance-portable. Starting with LINPACK, [7] and progressing to LAPACK [4] the Level 1, 2, 3 BLAS were introduced. The suffix $i$ in Level $i$ refers to the number of nested "do loops" required to do the computation of a given BLAS. Almost all of the floating-point operations of

DLA algorithms are performed through the use of BLAS calls. If performance were directly proportional to operation count then performance would be truly portable. However, with today's deep memory hierarchies and other new architecural features, this is no longer the case. To understand the performance inefficiency of LAPACK algorithms, it suffices to discuss the Level 3 BLAS, `DGEMM` (Double precision GEneral Matrix Matrix). A relationship exists between the Level 3 BLAS and their usage in most of level 3 factorization routines. This relationship introduces a performance inefficiency in block based factorization algorithms and we will now discuss the Level 3 BLAS, `DGEMM` (Double precision GEneral Matrix Matrix) to illustrate this fact.

In [1, 5, 25, 14] design principles for producing a high performance Level 3 `DGEMM` BLAS are given. A key design principle for `DGEMM` is to partition its matrix operands into submatrices and then call a `DGEMM` L1 kernel routine multiple times on its submatrix operands. Another key design principle is to change the data format of the submatrix operands so that each call to the L1 kernel can operate at or near the peak Million FLoating point OPerations per Second (MFlops) rate. This format change and subsequent change back to standard data format is a cause of a performance inefficiency in `DGEMM`. The `DGEMM API` requires that its matrix operands be stored as standard Fortran or C two-dimensional arrays.

Any DLA Factorization Algorithm (DLAFA) of a matrix $A$ calls `DGEMM` multiple times with *all* its operands being submatrices of $A$. For each call data copy will be done; therefore this unit cost gets multiplied by this number of calls. However, this overall cost can be eliminated by using the NDS to create a substitute for `DGEMM`; e.g. its analogous L1 kernel routine, which does *not* require the aforementioned data copy. So, as in [15, 17], for triangular matrices, we suggest that SBP format be used in concert with kernel routines.

This paper also describes a *new* concept which we call the L1 cache / L0 cache interface. We define a L0 cache as the register file of its floating point unit. Today, many architectures possess special hardware to support the streaming of data into the L1 cache from higher levels of memory [24, 21]. In fact with a large enough floating point register file it may be possible to do, say, a L2 or L3 cache blocking for a `DGEMM` kernel; ie, completely bypass the L1 cache. This is the case in [6] where a 6 by 6 register block for the `C` matrix can be used as this processor has 64 (32 dual SIMD) floating point registers. To do L0 register blocking we can concatenate tiny submatrices to faciltate streaming by reducing the number of streams. In effect, at the L0 level we have a concatenation of tiny submatrices behaving like a single long stride one vector that passes through L1 and into L0 in an optimal way. Sections 2, 2.1 and 2.2 give details about this technique. Using this extra level of blocking does not negate the benefits of using Square Blocks (SB). It is still essential that $\texttt{NB}^2$ elements of a SB be contiguous. However, the SBs are now no longer two dimensional Fortran or C arrays. We define a SB as *simple* when it is a two dimensional Fortran or C array. Using non-simple SBs as described here and in Section 2 allows us to claim that data copy for DLAFAs using SBs can be $O(N^2)$ instead of $O(N^3)$ which occurs when using Fortran or C two dimensional arrays.

Section 3 describes SB format for symmetric and triangular arrays. In this case one gets SBP format. Section 3.1 explains that SBP format is just as easy to use and to code for as is using standard full format for the same two purposes. Section 3.2 demonstrates a typical performance improvement one gets using simple SBP format over using standard full format. Similar performance results are attainable for non-simple SB, see [6]. Section 4 contains our main result about the reduction from $O(N^3)$ to $O(N^2)$ of data copy that is possible by using NDS; ie, either SB or SBP data format. The background material for this result is developed in Sections 2 and 3.

## 2 The Need to Reorder a Contiguous Square Block

NDS represent a matrix $A$ as a collection of SB's of order `NB`. Each SB is contiguous in memory. In [23] it is shown that a contiguous block of memory maps best into L1 cache as it minimizes L1 and L2 cache misses as well as TLB misses for matrix multipy and other common row and column matrix operations. When using standard full format on a DLAFA one does an $\mathrm{O}((N/NB)^2)$ amount of data copy in calling `DGEMM` in an outer do loop: `j=0,N-1,NB`. Over the entire DLAFA this becomes $\mathrm{O}((N/NB)^3)$.

On some processors there are floating point multiple load and store instructions associated with the multiple floating point operations; see [1, 6]. A multiple load / store operation requires that its multiple operands be contiguous in memory. The multiple floating point operations require their register operands to be contiguous; eg, see [6]. So, data that enters L1 may also have to be properly ordered to be able to enter L0 in an optimal way. Unfortunately, layout of a SB in standard row / column major order may *no longer* lead to an optimal way. In some cases it is sufficient to reorder a SB into submatrices which we call register blocks. Doing this produces a new data layout that will still be contiguous in L1 but can also be loaded into L0 from L1 in an optimal manner. Of course, the order and size in which the submatrices (register blocks) are chosen will be platform dependent.

### 2.1 A `DGEMM` kernel based on Square Block Format Partitioned into Register Blocks

In this contribution register blocks can be shown to be submatrices of a SB. This fact is important as it means that one can address these blocks in Fortran and C. To see this let $A$, $B$ and $C$ be three SB's and suppose we want to apply `DGEMM` to $A$, $B$ and $C$. We partition $A$, $B$ and $C$ into conformable submatrices that are also register blocks. Let the sizes of the register blocks (submatrices) be `kb` $\times$ `mb`, `kb` $\times$ `nb` and `mb` $\times$ `nb`. Thus $A^T$, $B$ and $C$ are partitioned matrices of sizes $k_1 \times m_1$, $k_1 \times n_1$ and $m_1 \times n_1$ respectively.

The `DGEMM` kernel we want to compute is $C = C - A^T B$ where matrix multiply is stride one across the rows and columns of $A$ and $B$ respectively. ($A^T$ will be

stride one along rows as $A$ is stride one along its columns.) Next, consider a fundamental building block of this `DGEMM` kernel; see Figure 1 It consists of

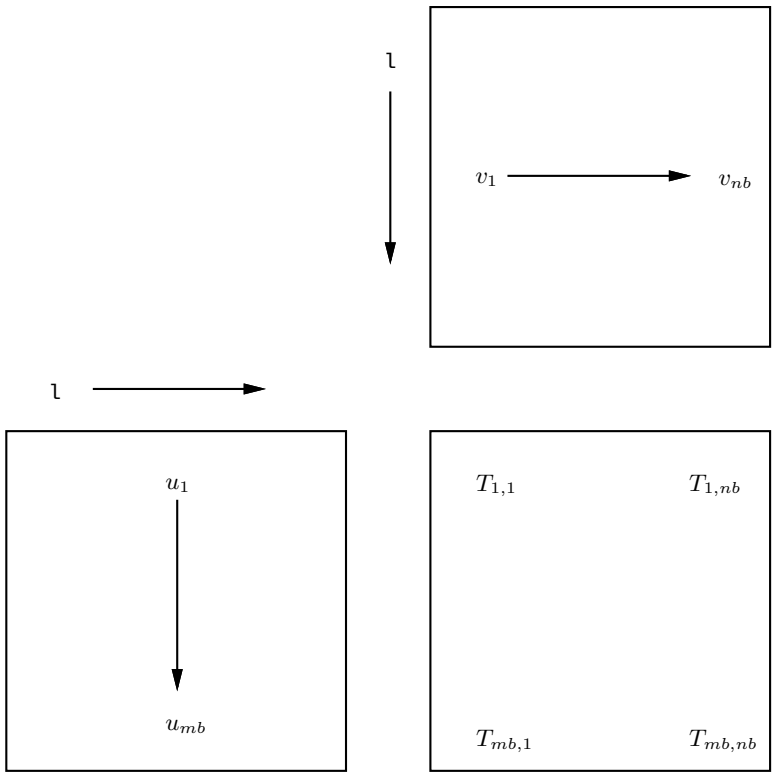


**Fig. 1.** Fundamental `GEMM` Kernel Building Block.

multiplying $k_1$ register blocks of $A^T$ by $k_1$ register blocks of $B$ and summing them to form the update of a register block of $C$. The entire kernel will consist of executing $m_1 \times n_1$ fundamental building blocks in succession to obtain a near optimal kernel for `DGEMM`.

### 2.2 A Fundamental `DGEMM` Kernel Building Block and Hardware Streaming

If we use simple SB format we would need `mb` rows of $A^T$ and `nb` columns of $B$ and $C$ to execute any fundamental building block. This would require `mb` + `2nb` stride one streams of matrix data to be present and working during the execution of a single building block. Many architectures do *n*ot possess special hardware to support this number of streams. Now the minimum number of streams is three; one each for matrix operands $A$, $B$ and $C$. Is three possible? An answer emerges if one is willing to change the data structure away from simple SB order.

In Figure 1 we describe a data layout of a fundamental register block computation [1]. Initially, a register block of $C$ is placed in $\texttt{mb} \times \texttt{nb}$ floating point registers $\texttt{T}(0:\texttt{mb}-1, 0:\texttt{nb}-1)$. An inner `do loop` on `l=0:K-1,kb` consists of performing $K/kb$ sets of $mb \times nb$ independent dot products on `T`. For a given single value of `l`, vectors `u, v` of lengths `mb, nb` from $A$ and $B$ respectively are used to update $\texttt{T} = \texttt{T} - uv^T$. This update is a `DAXPY` outer product update consisting of $mb \times nb$ independent Floating Multiply-Adds (FMAs). However, and this is important, since the `T`'s are in registers there are *no* loads and stores of the `T`'s. The entire update is $\texttt{T} = \texttt{T} - \texttt{A}^\texttt{T}(0:\texttt{K}-1, \texttt{i}:\texttt{i}+\texttt{mb}-1) \times \texttt{B}(0:\texttt{K}-1, \texttt{j}:\texttt{j}+\texttt{nb}-1)$. If $A$ and $B$ were simple SB's we would need to access vectors `u, v` with stride `NB` and also there would be $\texttt{mb}+\texttt{nb}$ streams. Luckily, if we transpose $\texttt{K} \times \texttt{mb}$ $A^T$ and $\texttt{K} \times \texttt{nb}$ $B$ we will simultaneously access `u, v` stride one, just get *t*wo streams, and still be able to address $A, B$ in the standard way. These two transpositions accomplish a matrix data rearrangement that allows for an excellent L1 / L0 interface of matrix data for the `DGEMM` kernel fundamental building block computation. We have just demonstrated that two streams are possible for $A, B$. By storing $C$ as $m_1 \times n_1$ register blocks (submatrices) contiguously in a contiguous SB in the order they are accessed by the `DGEMM` kernel we will get a single stream for $C$.

## 3 SB Packed Formats Generalize Standard Full and Packed Formats

Square Block Packed (SBP) formats are a generalization of packed format for triangular arrays. They are also a generalization of full format for triangular arrays. A major benefit of the SBP formats is that they allow for level 3 performance while using about half the storage of the full array cases. For simple SBP formats of a triangular matrix $A$ there are two parameters `TRANS` and `NB`, where usually $n \geq \texttt{NB}$. For these formats, we first choose a block size, `NB`, and then we lay out the matrix elements in submatrices of order `NB`. Each SB can be in column-major order (`TRANS` = 'N') or row-major order (`TRANS` = 'T'). These formats support both `uplo` = 'L' or 'U'; we only cover the case `uplo` = 'L'. For `uplo` = 'L', the first vertical stripe is $n$ by `NB` and it consists of $n_1$ SBs where $n_1 = \lceil n/\texttt{NB} \rceil$. It holds the first trapezoidal $n$ by `NB` part of $L$. Here we rename matrix $A$ matrix $L$ to remind the reader that our format is lower triangular. The next stripe has $n_1 - 1$ SBs and it holds the next trapezoidal $n$ - `NB` by `NB` part of $L$, and so on, until the last stripe consisting of the last leftover triangle is reached. The total number of SBs, `nt1`, is $n_1(n_1+1)/2$ and the total storage of SBP format is $nt1 * NB^2$. In [3] we introduced Lower Hybrid SBP format in which the diagonal blocks were stored in packed format. Thus, no additional storage was required. We also provided a fast means to transform to this format using a buffer of size `n*NB`. Now we turn to full format storage. To get SBP format one simply sets $\texttt{NB} = n$; ie, SBP format gives a single block triangle which happens to be full format.

[1] Compilers require that scalars be used to designate register usage. Also, we are using origin 1 in Fig. 1 and origin 0 in the text of Section 2.2

### 3.1 Benefits of SB and SB Packed Formats

We believe a main use of SB formats is for symmetric and triangular arrays. We call these formats SB Packed (SBP). An innovation here is that one can translate, verbatim, standard packed or full factorization algorithms into a corresponding SBP format algorithm by replacing each reference to an $i, j$ element of $A$ by a reference to its corresponding SB submatrix. Because of this storage layout, the beginning of each SB is easily located. Another key feature of using SB's is that SBP format supports Level 3 BLAS. Hence, old, packed and full codes are easily converted into SBP format level 3 code. Therefore, one keeps "standard packed or full" addressing so the library writer/user can handle his own addressing in a Fortran/C environment. Figure 2 describes a RLA for block Cholesky factorization and illustrates what we have just said. For clarity, we assume that `n` is a multiple of `nb`. Lines 2, 4, 7 and 9 of Figure 2 are calls to kernel routines.

```
do j = 0, n-nb, nb
  factor a(j:j+nb-1,j:j+nb-1) ! kernel routine for potrf
  do i = j + nb, n-nb, nb
    a(i:i+nb-1,j:j+nb-1) =
      a(i:i+nb-1,j:j+nb-1)*aT(j:j+nb-1,j:j+nb-1) ! kernel trsm
  end do
  do i = j +nb, n-nb, nb  ! THE UPDATE PHASE
    a(i:i+nb-1,i:i+nb-1) = a(i:i+nb-1,i:i+nb-1) -
      a(i:i+nb-1,j:j+nb-1)*aT(i:i+nb-1,j:j+nb-1) ! kernel syrk
    do k = i + nb, n-nb, nb ! The Schur Complement update phase
      a(k:k+nb-1,i:i+nb-1) = a(k:k+nb-1,i:i+nb-1) -
        a(k:k+nb-1,j:j+nb-1)*aT(i:i+nb-1,j:j+nb-1) ! kernel gemm
    end do
  end do
end do
```

**Fig. 2.** Block Version of Right Looking Algorithm for Cholesky Factorization

### 3.2 Performance for SBP Cholesky

Performance results of SBP format for Cholesky factorization were taken from [17]. We only include one of the two graphical plots. To be fair we show the curves for Block Hybrid Cholesky which includes the cost of doing a data transformation from packed format to SBP format. For small $N$ this cost is large, so we reduced this cost to zero by writing a Cholesky factor kernel for packed format; to distinguish this fact we call the resulting code with change over to SBP format BHC code. In Figure 3 the graphs plot MFlops versus matrix order $N$. Note that the x-axis is log scale; we let $N$ range from 10 to 2000. In the comparison

for BHC versus LAPACK we give four graphs: BHC, BHC + data transformation, `DPOTRF` and `DPPTRF`; we name these curves 1, 2, 3, 4. Data for the graphs were obtained on a 200 MHz IBM Power 3 with a peak performance of 800 MFlops. The performance of the BHC Cholesky algorithm of Figure 3 shows the data transformation does cost something. The actual crossover between the packed kernel and SBP format plus data transformation occurred at $N = 230$. For $N \leq 230$ curves 1 and 2 are identical. For $N \geq 230$ it pays to do the data transformation and the curves 1 and 2 separate. Curve 2 is faster than curve 3 for small $N$ (up to four times faster) and more than 10 % faster for large $N$.

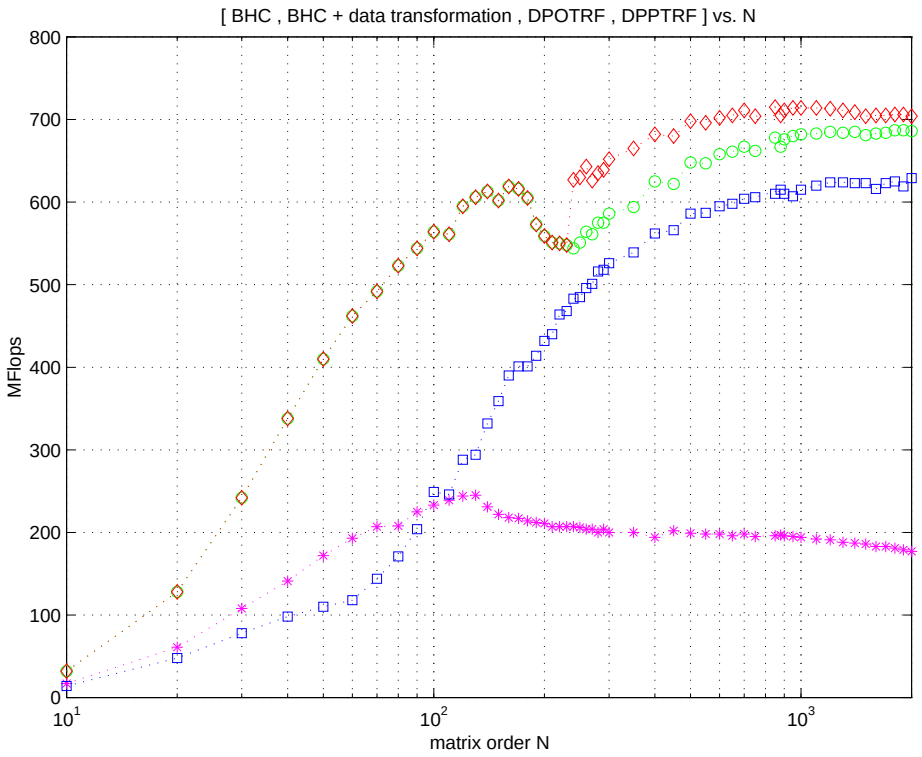


**Fig. 3.** Four Performance curves: diamond, circle, square, +

## 4 DLAFA's using SB Format require $O(n^2)$ Data Copy

We show this result by demonstrating it for a symmetric factorization as our focus is on Cholesky factorization and what we say about these factorizations applies to many other DLAFA's [8, 15, 12, 11]. There are many Cholesky DLAFA's. We only mention left and right looking as well as hybrid and recursive [7, 4, 15,

16, 2, 3] ones. A (left, right) looking algorithm does the (least, most) amount of computation in the outer `do loop` of stage `j`, respectively; see Figure 2 where we use a right looking algorithm. A recursive algorithm uses the divide-and-conquer paradigm. A hybrid algorithm is a combination of left and right looking algorithms. The current version of LAPACK [4] uses a hybrid algorithm. The paper [3] examines some of these algorithm types using a variant of SBP format, packed recursive and standard full and packed formats. Performance studies on six platforms, Alpha, IBM P4, Intel x86, Itanium, SGI and SUN were made. Overall, the hybrid algorithm, using a variant of SB format, was best. However, it was not a clear winner. In [3], we did *not* call BLAS kernel routines. Instead, we either called the vendor or Atlas BLAS [25]. So, these BLAS probably did $O(N^3)$ data copy during Cholesky factorization.

### 4.1 Data Copy of a DLAFA can be O($N^2$)

The result we now give holds generally for Right Looking Algorithms (RLAs) for DLAFAs. And similar results hold for Left Looking Algorithms (LLAs). Here we shall be content with demonstrating that the Cholesky RLA on SBP format can be done by only using O($N^2$) data copies. The O($N^3$) part of the block Cholesky RLA has to do with the Schur Complement Update (SCU); ie, the inner `DGEMM do loop` over variable `k`; see Figure 2. We assume each call to `DGEMM` will do data copy on each of its three operands $A$, $B$ and $C$. Now the number of $C$ SB's that get SCUed over the entire RLA is $n_1(n_1+1)(n_1-1)/6$ where $n_1 = \lceil \mathtt{N/NB} \rceil$ and `N` is the order of $A$. It is therefore clear that O($N^3$) data copies will occur.

In the case of simple SBs our result is obvious as no data copy occurs during execution of the RLA algorithm in Figure 2 because kernel routines of the BLAS are being called. So, there is only an initial reformating cost of full format $A$ to SBP format of $A$, which is clearly $O(N^2)$. Also, as mentioned in Section 3 and 3.2 this initial reformating is optional. Now, consider non-simple SBs. In Sections 2.1 and 2.2 we indicated that it is now usually necessary to reformat each SB every time `DGEMM` is called if non-simple SB's are used. We now demonstrate that we can reduce this data copy cost to O($N^2$). What we intend to do is to store the $C$ operands of `DGEMM` in the register block format that was indicated in Sections 2.1 and 2.2. Hence, the format of these $C$ operands is then fixed throughout this algorithm and no additional data copy occurs for them during the entire execution of this RLA; see Figure 2. And clearly, an initial formatting cost, if necessary, is only O($N^2$). Now we examine the $A$ and $B$ operands of the SCU for the outer loop variable `j`. SB's $A(\mathtt{j}:\mathtt{n_1},\mathtt{j})$ whose total is $\mathtt{n_1}-\mathtt{j}$ are needed for the SCU as they constitute all the $A, B$ operands of the SCU at iteration `j`. Summing from `j=1` to $\mathtt{j}=\mathtt{n_1}-\mathtt{1}$ we find just $\mathtt{n_1}(\mathtt{n_1}-\mathtt{1})/\mathtt{2}$ SB's in all that need reformatting ( data copying ) over the course of this entire RLA; see Figure 2. And since there are both $A$ and $B$ operands we may have to double this amount to $n_1(n_1-1)$ SB's. However, in either case this amount of data copy is clearly O($N^2$).

## 5 Summary and Conclusions

This paper demonstrates that the standard data structures of DLA can hurt the perfomance of its factorization algorithms. It indicates that by using NDS this performance loss can be lessened. Specifically, it describes SB and SBP format as a replacement of these standard data structures. SB and SBP data structures are shown to be easy to use and to code for. These two features are strong features of the standard data structures of DLA. SB and SBP formats have two desirable features that the standard data structures lack. SBP format uses near minimal storage for symmetric and triangular matrices whereas standard full format storage uses nearly double the minimal storage. Secondly, SB and SBP formats give DLAFAs better performance than standard full format does. Our main result, that DLAFAs require only $O(N^2)$ data copy, indicates partly why this is so. The use of standard full format requires $O(N^3)$ data copy by the level 3 BLAS being used by the DLAFA. We assumed these BLAS always did data copy on their submatrix operands. We discussed a new concept called the L1 / L0 cache interface. The existence of this interface showed one the necessity of introducing non-simple SBs in order to maintain high performance of `DGEMM` kernels on several new platforms. These non-simple SBs were able to fully exploit hardware streaming which is a feature of several new platforms.

## 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, F. G. Gustavson, and J. Wasnieski. A Recursive Formulation of Cholesky Factorization of a Matrix in Packed Storage. *ACM TOMS*, Vol. 27, No. 2 June 2001, pp. 214-244.
3. B. S. Andersen, J. A. Gunnels, F. G. Gustavson, J. K. Reid and J. Wasnieski. A Fully Portable High Performance Minimal Storage Hybrid Cholesky Algorithm. *ACM TOMS*, Vol. 31, No. 2 June 2005, pp. 201-227.
4. 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, (http://www.netlib.org/ lapack/lug/lapack_lug.html).
5. J. Bilmes, K. Asanovic, C. Whye Chin, J. Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. *Proceedings of International Conference on Supercomputing*, Vienna, Austria, 1997.
6. 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.
7. J. J. Dongarra, C. B. Moler, J. R. Bunch, G. W. Stewart. *LINPACK Users' Guide Release 2.0*, SIAM, Philadelphia, 1979.
8. J. J. Dongarra, F. G. Gustavson, A. Karp. Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine. *SIAM Review*, Vol. 26, No. 1, Jan. 1984, pp. 91,112.

9. J. J. Dongarra and J. Du Croz, S. Hammarling and R. J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms, *TOMS*, Vol. 14, No. 1, Mar. 1988, pp. 1-17.
10. 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.
11. E. Elmroth, F. G. Gustavson, B. Kagstrom, and I. Jonsson. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. *SIAM Review*, Vol. 46, No. 1, Mar. 2004, pp. 3,45.
12. J. Gunnels, F. G. Gustavson, G. Henry, and R. van de Geijn. Formal linear algebra methods environment (FLAME). *ACM TOMS*, Vol. 27, No. 4, Dec. 2001, pp. 422-455.
13. J. A. Gunnels, F. G. Gustavson. A New Array Format for Symmetric and Triangular Matrices. *Computational Science - Para 2004*, J. J. Dongarra, K. A. Madsen, J. Wasniewski, eds., Lecture Notes in Computer Science 3732. Springer-Verlag, pp. 247-255, 2006.
14. J. A. Gunnels, F. G. Gustavson, G. M. Henry, R. A. van de Geijn. A Family of High-Performance Matrix Multiplication Algorithms. *Computational Science - Para 2004*, J. J. Dongarra, K. A. Madsen, J. Wasniewski, eds., Lecture Notes in Computer Science 3732. Springer-Verlag, pp. 256-265, 2006.
15. 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.
16. F. G. Gustavson and I. Jonsson. Minimal Storage High Performance Cholesky via Blocking and Recursion *IBM Journal of Research and Development*, Vol. 44, No. 6, Nov. 2000, pp. 823,849.
17. 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.
18. F. G. Gustavson. New Generalized Data Structures for Matrices Lead to a Variety of High performance Dense Linear Algorithms. *Computational Science - Para 2004*, J. J. Dongarra, K. A. Madsen, J. Wasniewski, eds., Lecture Notes in Computer Science 3732. Springer-Verlag, pp. 11-20, 2006.
19. F. G. Gustavson, J. Wasniewski. LAPACK Cholesky routines in rectangular full packed format. *Computational Science - Para 2006*, Bo Kagstom, E. Elmroth, eds., Lecture Notes in Computer Science xxxx. Springer-Verlag, pp. aaa-bbb, 200y.
20. IBM. IBM Engineering and Scientific Subroutine Library for AIX Version 3, Release 3. *IBM Pub. No. SA22-7272-04* Dec. 2001.
21. R. Kalla, B. Sinharoy, J. Tendler. Power 5 *HotChips-15*, August 17-19, 2003. Stanford, CA
22. C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage, *TOMS*, Vol. 5, No. 3, Sep. 1979, pp. 308-323.
23. N. Park, B. Hong, V. K. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Trans. Parallel and Distributed Systems*, 14(7):640-654, 2003.
24. B. Sinharoy, R.N. Kalla, J.M Tendler, R.G. Kovacs, R.J. Eickemeyer, J.B. Joyner. POWER5 System Microarchitecture *IBM Journal of Research and Development*, Vol. 49, No. 4/5, July/Sep. 2005, pp. 505-521.
25. R. C. Whaley, A. Petitet, J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 2001(1-2), pp. 3-35.