Minimal Data Copy For Dense Linear Algebra Factorization

Fred G. Gustavson

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

Abstract. 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 $O(n^2)$.

1 Description of a Fortran and C Inefficiency for Dense Linear Agebra Factorizations

The current Most Commonly Used (MCU) Dense Linear Algebra (DLA) algorithms for serial and SMP processors have a performance inefficiency and hence they give sub-optimal performance. We show that standard 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. The NDS generalizes the current storage layouts for both the Fortran and C programming languages.

The BLAS(Basic Linear Algebra Subroutines) were introduced to make the algorithms of DLA performance-portable. However, a relationship exists between the Level 3 BLAS used 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 [5, 2] 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 an 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` interface definition 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; the principle inefficiency is therefore multiplied by this number of calls. However, this inefficiency 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.

This paper describes a *new* concept which we call the L1 cache / L0 cache interface [1]. The L0 cache is the register file of a Floating Point Unit. Today, many architectures possess special hardware to support the streaming of data into the L1 cache from higher levels of memory [4]. 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 [1] where a 6 by 6 register block for the `C` matrix can be used as this processor has 32 dual SIMD floating point registers. To do L0 register blocking we can concatenate tiny submatrices to faciltate streaming; ie, to reduce 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 and 2.1 gives details about this technique. Using this extra level of blocking does not negate the benefits of using square blocks SB's. It is still essential that $\mathtt{NB}^2$ elements of a SB be contiguous. However, the SB's are now no longer Fortran or C arrays which we define as simple.

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 [3] 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=1,N,NB`. Over the entire DLAFA this becomes $\mathrm{O}(N^3)$.

On some RISC processors there are floating point multiple load and store instructions associated with the multiple floating point operations; see [1]. A multiple load / store operation usually requires that its multiple operands be contiguous in memory. Some newer processors with multiple floating point operations require their operands to be contiguous; eg, [1]. 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 contrbution register blocks can be considered as submatrices of a SB. This fact is very important as it means 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 register blocks. Let the sizes of the register blocks be $\mathtt{kb}\times\mathtt{mb}$, $\mathtt{kb}\times\mathtt{nb}$ and $\mathtt{mb}\times\mathtt{nb}$. Thus A^T, B and C are matrices of register blocks 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 $C = C - A^T B$ as matrix multiply is stride one across the rows and columns of A and B respectively. Next, consider a fundamental building block of this `DGEMM` kernel which consists of 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 therefore consist of executing $m_1 \times n_1$ fundamental building blocks in succession to obtain a near optimal kernel for `DGEMM`. If we use simple SB format we would need `mb` rows of A^T and `nb` columns of B and C to execute the fundamental building block. This would require $\mathtt{mb} + 2\mathtt{nb}$ 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. Initially, a register block of C is placed in $\mathtt{mb} \times \mathtt{nb}$ floating point registers $\mathtt{T}(\mathtt{0} : \mathtt{mb} - \mathtt{1}, \mathtt{0} : \mathtt{nb} - \mathtt{1})$. An inner `do loop` on `l=0:K-1,kb` consists of performing kb sets of $mb \times nb$ independent dot products on `T`. For a given single value of `l` vectors $\mathtt{u}, \mathtt{v}$ of lengths $\mathtt{mb}, \mathtt{nb}$ from A and B respectively are used to update $T = T - uv^T$. This update is a `DAXPY` outer product update. 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 $\mathtt{T} = \mathtt{T} - \mathtt{A}^{\mathtt{T}}(\mathtt{0} : \mathtt{K} - \mathtt{1}, \mathtt{i} : \mathtt{i} + \mathtt{mb} - \mathtt{1}) \times \mathtt{B}(\mathtt{0} : \mathtt{K} - \mathtt{1}, \mathtt{j} : \mathtt{j} + \mathtt{nb} - \mathtt{1})$. If A and B were simple SB's we would need to access vectors $\mathtt{u}, \mathtt{v}$ with stride `NB` and also there would be $\mathtt{mb} + \mathtt{nb}$ streams. Luckily, if we transpose $\mathtt{K} \times \mathtt{mb}$ A^T and $\mathtt{K} \times \mathtt{nb}$ B we will simultaneously access $\mathtt{u}, \mathtt{v}$ stride one, just get two streams and be able to address A, B in the standard way. These two transpositions accomplishes a matrix data rearrangement that allows for an excellent L1 / L0 interface of matrix data for the `DGEMM` kernel fundamental building block computation.

3 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. 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 square blocked, packed, 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.

3.1 Data Copy of DLAFA can be O(N^2)

Our proof sketch is for Cholesky factorization. However, what we say about Cholesky factorization applies to many other DLAFA's. There are many Cholesky DLAFA's. We only mention left and right looking which do the least, most amount of computation in the outer `do loop` stage `j`, respectively. The result we now give holds generally for Right Looking Algorithm (RLA)s 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`. 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 $n_1(n_1-1)(n_1-2)/2$ where $n_1 = \lceil \mathtt{N/NB} \rceil$ and `N` is the order of A. It is therfore clear that O(N^3) data copies will occur.

In Section 2.1 we indicated that it is now usually necessary to reformat each SB every time `DGEMM` is called if 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 Section 2.1. 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. 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 $\mathtt{j}=\mathtt{n_1}$ we find just $\mathtt{n_1}(\mathtt{n_1}-\mathtt{1})/2$ SB's in all that need reformatting (data copying) over the course of this entire RLA. 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).

References

1. 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.
2. 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, 2004.
3. 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.
4. 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
5. 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.