RC23443 (W0411-176) November 19, 2004
Computer Science

IBM Research Report

A Novel Theoretical Model Produces Matrix Multiplication Algorithms That Predict Current Practice

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

Greg M. Henry
Intel Corporation
Bldg. EY2-05
5450 NE Elam Young Parkway
Hillsboro, OR 97124-6461

Robert A. van de Geijn
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

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

A Novel Theoretical Model Produces Matrix Multiplication Algorithms That Predict Current Practice

JOHN A. GUNNELS
IBM T.J. Watson Research Center
and
FRED G. GUSTAVSON
IBM T.J. Watson Research Center
and
GREG M. HENRY
Intel Corporation
and
ROBERT A. VAN DE GEIJN
The University of Texas at Austin

We develop a simple model of hierarchical memories and we use it to determine an optimal strategy for blocking matrices. This model predicts the form of current, state-of-the-art L1 kernels. Additionally, it shows that current L1 kernels can continue to produce their high performance on operand matrices that are as large as the L2 cache. For a hierarchical memory with L memory levels (main memory and L-1 caches), our model reduces the number of potential matrix multiply algorithms from 6^L to four. We use the shape of the matrix input operands to select one of our four algorithms. Because of space limitations, we do not include perforamnce results.

Categories and Subject Descriptors: []: —

General Terms:

Additional Key Words and Phrases:

1. INTRODUCTION

In this paper, we discuss an approach to implementing matrix multiplication, $C = AB + C$, that is based on a simple model of moving data between adjacent memory

Authors' addresses: John A. Gunnels, IBM T.J. Watson Research Center P.O. Box 218 Yorktown Heights, N.Y. 10598 gunnels@us.ibm.com. Fred G. Gustavson, IBM T.J. Watson Research Center P.O. Box 218 Yorktown Heights, N.Y. 10598 gustav@watson.ibm.com. Robert A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, rvdg@cs.utexas.edu. Greg M. Henry, Intel Corp., Bldg EY2-05, 5350 NE Elam Young Pkwy, Hillsboro, OR 97124-6461, greg.henry@intel.com.

layers. This model allows us to make the following theoretical contributions: (1) At each level of the memory hierarchy most of that layer should be filled with a submatrix of one of the three operands while smaller submatrices of the other two operands are "streamed in" from the previous (slower) memory layer; (2) If a given layer is mostly filled with a submatrix of an operand, the next (faster) layer should be filled mostly with a submatrix of one of the other two operands. (3) At the L1 level, where all computations must take place, the model accurately predicts the form of two kernel routines. Embodied in points (1-3) is the general idea of using cache blocking at every memory level. The amount of work (FLOPS) done at every level is $2MNK$ on operands of size $MN + MK + KN$. We are assuming that any computer architecture, at every memory level, can feed (stream) the operands through the memory hierarchy in time less than or equal to the time for doing $2MNK$ FLOPS at maximum floating point rate (peak MFLOPs). This paper is a condensation of an amplification of [Gunnels et al. 2001] that was written in Februrary 2002.

Over the last two decades, numerical linear algebra libraries have been re-designed to accommodate multi-level memory hierarchies, thereby replacing earlier libraries such as LINPACK and EISPACK. It was demonstrated, e.g. [Gallivan et al. 1987; Corporation 1986], that on cache-based (multi-level memory) machines, block-partitioned matrix algorithms could evince dramatic performance increases, when contrasted with their non-blocked counterparts. The key insight of block partitioning research was that it allowed the cost of $O(n^2)$ data movement between memory layers to be amortized over $O(n^3)$ computations, which had to be performed by kernel routines in the L1 (lowest level) cache. At the base of our algorithms are *kernel* routines used in concert with submatrix partitioning (referred to as *cache blocking*). Our new memory model provides a mathematical "prediction" for the necessity of both. Prior work [Bilmes et al. 1997; 1998; Whaley and Dongarra 1998; Agarwal et al. 1994], produced kernel routines, first by hand and, later, via a code generator, which employed parameter variations and careful timing in order to complete the generate-and-test cycle. The principles used in these automatic code generating versions were, for the most part, the same as those used by the ESSL library. The PHiPAC [Bilmes et al. 1997; 1998] and ATLAS [Whaley and Dongarra 1998] projects each employ a code generator approach. This methodology uses a program that writes a series of programs, each differing by a set of parameters varied by the "parent" (code writing) program. Each of these produced programs is automatically compiled and carefully timed. Using the timing characteristics of these programs, the most time-efficient code is selected. The work presented here generalizes the ATLAS model. In Section 4.1 a fuller comparison of ATLAS and our model is given. It is also related to [Goto and vandeGeijn 2002] which is discussed in Section 4.2

Our model partitions matrices A, B, and C into submatrices A_{ip}, B_{pj}, and C_{ij} and performs submatrix multiplication and this is illustrated in Section 2. In Section 3, we state a result on the minimal amount of times any matrix algorithm must bring matrix elements into and out of a given cache level. Our model achieves this lower bound complexity to within a constant factor. The important point here is that A, B, and C must come through the caches multiple times. With

this result in mind, Section 3 takes the L algorithmic pieces developed in Section 2 and combines them into 2^L different, composite algorithms. Furthermore, a new concept called conservation of matrix operand sizes is used to reduce the number of algorithms to just four.

Now, since the submatrices of A, B, and C must be brought through the memory hierarchy multiple times, it pays to reformat them *once* so that data re-arrangement is not done multiple times. This also aids the effort for achieving optimal L1 kernel performance. In Sections 4, 4.1 and 4.2 we briefly describe how this is done in greater detail. Our main point is that our model features loading and storing submatrices at each cache level. Clearly, loading and storing is overhead and only serves to reduce the MFLOPS value predicted by our model. By using the new data structures, `NDS`, [Gustavson 2001], we can dramatically reduce the loading and storing overhead cost, thereby increasing our MFLOP rate. Because of space limitations using `NDS` in the context of our model as well as our two kernel routines that feature streaming will not be covered.

We mention that our model does not involve automatic blocking via recursion. Recently, this area has received a great deal of attention from many important computations such as matrix factorizations, FFT, as well as matrix multiplication. See [Elmroth et al. 2004]. Some researchers have referred to such methods as cache oblivious [Frigo et al. 1999]. Others have focused on "recursion" to produce new data formats for matrices, instead of the traditional Fortran and C data structures. Our view is that recursion is very powerful and excellent results are obtainable, but *only* if one combines this approach with blocking. Nonetheless, whenever one neglects the specific features of a computer architecture, one can expect performance to suffer. We hope that readers will see our model as sufficiently simple and yet comprehensive enough to capture the salient features of matrix multiplication, and, thereby stimulate further interest in high-performance implementations of matrix-matrix multiplication.

2. OVERVIEW OF OUR MODEL

The general form of a matrix-matrix multiply is $C \leftarrow \alpha AB + \beta C$ where C is $m \times n$, A is $m \times k$, and B is $k \times n$. We will use the following terminology when referring to a matrix-matrix multiply when two dimensions are large and one is small; see Table 1.

2.1 A Cost Model for Hierarchical Memories

We will briefly model the memory hierarchy as follows:

(1) The memory hierarchy consists of $L + 1$ levels, indexed $0, \ldots, L$. Level 0 corresponds to the registers, level L as main memory and the reamining levels as caches. We will often denote the hth level by L_h.

(2) If $m_h \times n_h$ matrix $C^{(h)}$, $m_h \times k_h$ matrix $A^{(h)}$, and $k_h \times n_h$ matrix $B^{(h)}$ are all stored in level h of the memory hierarchy then forming $C^{(h)} \leftarrow A^{(h)}B^{(h)} + C^{(h)}$ costs time $2m_h n_h k_h \gamma_h$. γ_h is defined to be effective unit cost of a floating point operation over all $m_h n_h k_h$ multiply-adds at level L_h.

In an expanded version of the paper our model is fully described.

	Condition	Shape
Matrix-panel multiply	n is small	$C = A\,B + C$
Panel-matrix multiply	m is small	$C = A\,B + C$
Panel-panel multiply	k is small	$C = A\,B + C$

Table I. Three basic ways to perform matrix multiply

```
for i = 1, ..., M_h, m_h
    for j = 1, ..., N_h, n_h
        for p = 1, ..., K_h, k_h
            C_ij^(h) ← A_ip^(h) B_pj^(h) + C_ij^(h)
        endfor
    endfor
endfor
```

Fig. 1. Generic loop structure for blocked matrix-matrix multiplication.

2.2 Building-blocks for matrix multiplication

Consider the matrix multiplication $C^{(h+1)} \leftarrow A^{(h+1)}B^{(h+1)} + C^{(h+1)}$ where $m_{h+1} \times n_{h+1}$ matrix $C^{(h+1)}$, $m_{h+1} \times k_{h+1}$ matrix $A^{(h+1)}$, and $k_{h+1} \times n_{h+1}$ matrix $B^{(h+1)}$ are all resident in L_{h+1}. Let us assume that somehow an efficient matrix multiplication procedure exists for matrices resident in L_h. According to Table 1, we develop three distinct approaches for matrix multiplication procedures for matrices resident in L_{h+1}.

Partition

$$C^{(h+1)} = \left(\begin{array}{c|c|c} C_{11}^{(h)} & \cdots & C_{1N_h}^{(h)} \\ \hline \vdots & & \vdots \\ \hline C_{M_h 1}^{(h)} & \cdots & C_{M_h N_h}^{(h)} \end{array}\right), A^{(h+1)} = \left(\begin{array}{c|c|c} A_{11}^{(h)} & \cdots & A_{1K_h}^{(h)} \\ \hline \vdots & & \vdots \\ \hline A_{M^{(h)}1}^{(h)} & \cdots & A_{M^{(h)}K^{(h)}}^{(h)} \end{array}\right), \text{ and} \tag{1}$$

$$B^{(h+1)} = \left(\begin{array}{c|c|c} B_{11}^{(h)} & \cdots & B_{1N_h}^{(h)} \\ \hline \vdots & & \vdots \\ \hline B_{K_h 1}^{(h)} & \cdots & B_{K_h N_h}^{(h)} \end{array}\right) \tag{2}$$

where $C_{ij}^{(h)}$ is $m_h \times n_h$, $A_{ip}^{(h)}$ is $m_h \times k_h$, and $B_{pj}^{(h)}$ is $k_h \times n_h$. A blocked matrix-matrix multiplication algorithm is given in Figure 1 where the order of the loops is arbitrary. Here, M_h, N_h, and K_h are the ratios m_{h+1}/m_h, n_{h+1}/n_h, and k_{h+1}/k_h, respectively, which for clarity we assume to be integers. Further, the notation, $i = 1, \ldots, M_h, m_h$, means that the index, i iterates from 1 to M_h in steps of 1, where each block has size m_h. The unit cost γ_{h+1} will depend on the unit cost γ_h,

the cost of moving data between the two memory layers, the order of the loops, the blocking sizes m_h, n_h, and k_h, and finally the the matrix dimensions m_{h+1}, n_{h+1}, and k_{h+1}. The purpose of our analysis will be to determine optimal m_{h+1}, n_{h+1}, and k_{h+1} by taking into account the cost of all the above mentioned factors.

We can assume that m_h, n_h, and k_h are known when $h = 0$. Floating-point operations can only occur with data being transferred between the L1 cache (level 1) and the registers (level 0). This is, obviously, where all of the arithmetic of matrix-matrix multiply is done and it is a separate problem which is probably the most important part of any matrix-matrix multiply algorithm to get "right" (to optimize). Hence, via an induction-like argument we can, using our model, find optimal m_1, n_1, and k_1. Similar inductive reasoning leads to optimal blockings for the slower and larger memory layers $h = 2, \ldots, L$.

In an expanded version of this paper we analyze Figure 1 for all of its six variants, formed by permuting i, j, and p. For all six variants the central computation is: $C_{ij}^{(h)} \leftarrow A_{ip}^{(h)} B_{pj}^{(h)} + C_{ij}^{(h)}$. We make the observation that each element of $C_{ij}^{(h)}$, $A_{ip}^{(h)}$, and $B_{pj}^{(h)}$, are used k_h, n_h, and m_h times, respectively, during this matrix computation. Thus, we identify k_h, n_h, and m_h as the *re-use factors* of $C_{ij}^{(h)}$, $A_{ip}^{(h)}$, and $B_{pj}^{(h)}$, respectively. And, when we speak of *amortizing* the computation, we mean that elements of elements of $C_{ij}^{(h)}$, $A_{ip}^{(h)}$, and $B_{pj}^{(h)}$, have re-use factors, k_h, n_h, and m_h and it is advantageous to make these factors as high as possible.

Also, we show in the expanded version that the inner loop repetition factors K_h, N_h or M_h will each take the value one because of the aforementioned streaming resulting in " loop fusion ". See Section 3.1 ahead. This means that the cache resident operand has an extra large re-use factor and thereby benefits from streaming.

2.3 A Corresponding Notation

By examining Figure and Table 1 the following correspondence becomes evident: (i, j, p) $\leftrightarrow$ (`RPM, RMP, RPP`). Here P stands for panel, M for matrix, and R for repeated. Think of repeated as a synomyn for streaming. For example, algorithm `RMP-RPP` is uniquely designated by its outer and inner loop indices, j and p. Similarly, algorithm `RPM-RPP` is uniquely designated by its outer and inner loop indices, i and p. For the remaining four combinations: algorithms `RPM-RMP` , `RPP-RMP` and `RMP-RPM` , `RPP-RPM` correspond to outer-inner loop pairings i, j , p, j and j, i , p, j respectively.

Floating point arithmetic must be performed in the L1 cache. Let $h = 1$. It is clear that p should be the inner loop variable as then a `DDOT` (as opposed to a `DAXPY`) kernel can be chosen. Choosing `DDOT` saves loading and storing C values for each floating point operation. In practice, at the L_1 level, almost all implementations that we are aware of are limited to these `RMP-RPP` and `RPM-RPP` algorithms. Hence we limit ourselves to only these two algorithms at the L_1 level. The "`RPP`" is redundant and will be dropped.

So, we have two L1 kernels, called `RMP` and `RPM`. When $h > 1$, we refer to the six algorithms illustrated in Table and Figure 1 as algorithmic pieces. In the next section we will see that, for an L level memory hierarchy, $L - 1$ algorithmic pieces,

Case	Dimensions	Matrix Sizes	L2-Resident	L1-Resident	"Streaming" Matrix
1	$M > N > K$	$\|C\| > \|A\| > \|B\|$	A	B	C
2	$M > K > N$	$\|A\| > \|C\| > \|B\|$	C	B	A
3	$N > M > K$	$\|C\| > \|B\| > \|A\|$	B	A	C
4	$N > K > M$	$\|B\| > \|C\| > \|A\|$	C	A	B
5	$K > M > N$	$\|A\| > \|B\| > \|C\|$	B	C	A
6	$K > N > M$	$\|B\| > \|A\| > \|C\|$	A	C	B

Table II. The six possible matrix size orderings delineated.

$h = 1, \ldots, L-1$ can be merged into a single algorithm. Because of the above-mentioned correspondence, we can label the distinct algorithms that emerge either by a string of loop indices (i, j, p) or by the analogous (`RPM, RMP, RPP`) designations. Each such string will contain $L-1$ loop indices or $L-1$ hyphenated symbol pairs from the (`RPM, RMP, RPP`) alphabet.

2.4 Matrix Operand Sizes Dictate the Streaming Values

Consider again Figure 1 for a particular value of h so that problem size is $m_{h+1} \times n_{h+1} \times k_{h+1}$. The outer loop index can be i, j or p. Thus we have three cases and for each there are two algorithms corresponding to the order of the middle and inner loops indices. Also, there are six possible size orderings for the problem dimensions, (e.g. $M > N > K$, etc.). It is self-evident that any ordering of the three sizes imposes an unambiguous ordering on the size of the three operands A, B, and C, involved in matrix multiplication. For example, $M > N > K$ implies that $C(M \times N)$ is larger than $A(M \times K)$, which is, in turn, larger than $B(K \times N)$. Consider the case where K is the largest dimension and $h = L - 1$, as follows:

(1) $K > M > N$ which becomes $k_{h+1} > m_{h+1} > n_{h+1}$ or $|A| > |B| > |C|$.

(2) $K > N > M$ which becomes $k_{h+1} > n_{h+1} > m_{h+1}$ or $|B| > |A| > |C|$.

In both cases, the algorithms used should have p as the inner loop index. Streaming provides us with the reason for this assertion. To be more specific, operand C benefits from the large streaming value of k_{h+1}. Since C is the smallest matrix in these cases, C will tend to be composed of fewer submatices making up all of C. In other words, M_h and N_h, the repetition values, will tend to be the two smallest integers, as the $m_{h+1} \times n_{h+1}$ matrix, C, is the smallest operand. Refer to case (1) above which depicts Figure 1 with loop order j, i, p. The cache resident matrix in this case is the A operand and hence N is the streaming value. In case (2) above the loop order is i, j, p and B is the cache resident matrix and M is the streaming value. Finally, in the remaining four cases we have n_{h+1} as the largest value for two cases corresponding to inner loop index j and m_{h+1} as the largest value for two cases corresponding to inner loop index i.

In Table II we list the six possible size-orderings for problem dimensions, M, N, and K. In cases where two or more dimensions are equal, $>$ can be viewed as $\geq$.

Row entries 5 and 6 of Table II suggest that the C operand should be L1 cache-resident. However, for reasons given in Section 2.3, this situation leads to an inefficient inner kernel. Our solution to this dilemma is to reverse the L2/L1 residence roles of the smaller two operands.

The case $L = 2$ suggests a strategy for algorithmic choices when $L > 2$. Choose the smaller two operands to be cache-resident with the large operand supplying the streaming values. This choice is based on maximizing the streaming feature exhibited by L1 cache algorithms on current architectures and machine designs. The smaller two operands will alternate in the role of cache-resident matrix.

3. THEORETICAL RESULTS ON BLOCKING FOR MATRIX MULTIPLICATION

Our model features block-based (submatrix) matrix multiplication. One might object to our research on the grounds that there exists some other, as yet unknown, superior method for performing the standard matrix multiply. We now state that this cannot happen as our model is optimal in the following sense:

THEOREM 3.1. *Any algorithm that computes $a_{ik}b_{kj}$ for all $1 \leq i, j, k \leq n$ must transfer between memory and D word cache $O(n^3/\sqrt{D})$ words if $D < n^2/5$.*

This theorem, was first demonstrated by Hong and Kung [Hong and Kung 1981] in 1981. Toledo [Toledo 1999] gives another, simplified, proof of their result. Our model of block-based matrix multiplication transfers $O(n^3/\sqrt{D})$ words for an L1 cache of size D words. Furthermore, we can apply this theorem to every cache level residing "below" L1 in the memory pyramid: L2, L3, ..., in the same way. Our model evinces the $\Omega(n^3/\sqrt{D_h})$ word movement, where D_h is the size of L_h, $h = 1, 2, \ldots$. To illustrate this, let us review Figure 1 and Table I. In Figure 1 with inner loop index p, matrix C is brought into L_h once while matrices A and B are brought through L_h, N_h and M_h times, respectively. Similarly, in Figure 1 with inner loop index j, matrix A is brought into L_h once while matrices B and C are brought through L_h, M_h and K_h times, respectively. Finally, in Figure 1 with inner loop index i, matrix B is brought into L_h once while matrices A and C are brought through L_h, N_h and K_h times, respectively. Thus, we can see that the streaming matrices employed in our algorithms have the repetitive factors, N_h, M_h, and K_h.

3.1 Bridging the Memory Layers

Overall, the number of single choices is bounded by 6^{L+1} as each instance of Figure 1 has six choices and h takes on $L+1$ values. In Section 2.3, with $h = 1$ we restricted our choice to two L1 kernels; i.e. we set the inner loop index to be p. For $h > 1$ we shall "fuse" the inner loop index at level $h + 1$ to be equal to the outer loop index at level h. This amounts to using streaming at every memory level. This means the number of algorithmic choices reduces to 2^L. Thus, Figure 1 will have only 2 loops (instead of 3) when they are combined into a composite algorithm for a processor with L caches, where cache L is memory. When L is one, we have a flat (uniform) memory and a call to either of our kernel routines, `RMP` (j_0, i_0, p_0) or `RPM` (i_0, j_0, p_0), will conform to our model. For arbitrary L, we get a nested looping structure consisting of $2(L-1)$ loop variables, 2 each from levels 2 to L. When one considers the L1 kernel as well, there are $2L + 1$ variables (as we must add i_0, j_0, and p_0).

Furthermore, for streaming to fully work one must have matrix operands to stream from. We call this conservation of matrix operands. M, N, and K are inputs to `DGEMM`. In Table 2 and Section 2.3 we saw that were four algorithms as cases 5 and 6 were mapped to cases 2 and 4 respectively. Thus, the 2^L choices map to just four

algorithms as the choice of the outer and inner indices becomes fixed for each value of h. The reason to do this is based on the conservation of matrix operand sizes for the given `DGEMM` problem. The two largest of the three input dimensions M, N, and K determine the streaming matrix as the A, B, or C operand of largest size. The four patterns that emerge for cache residency are $A, B, A, \ldots$, $B, A, B, \ldots$, $A, C, A, \ldots$, and $B, C, B, \ldots$, for $h = 1, 2, 3, \ldots$ The associated streaming values come from the two dimensions of the matrix operands C, C, B, and A respectively.

4. PRACTICAL CONSIDERATIONS

In the previous section we developed a model for the implementation of matrix-matrix multiplication that amortizes movement of data between memory hierarchies from a local point of view. However, there are many issues associated with actual implementation that are ignored by the analysis and the heuristic. In this section we briefly discuss some implementation details that do take some of those issues into account. We do so by noting certain machine characteristics that to our knowledge hold for a wide variety of architectures. While in the previous sections we argued from the bottom of the pyramid to the top (L_{h+1} in terms of L_h), we now start our argument at the top of the pyramid after providing some general guidelines and background.

4.1 Background and Related Work

In the Introduction, we mentioned both the PHiPAC and the ATLAS research efforts. We now describe ATLAS in more detail as it more closely relates to our contributions. In ATLAS, the code generation technique is only applied to the generation of a *single* L1 kernel. The ATLAS L1 kernel has the same form as the ESSL `DGEMM` kernel outlined in [Agarwal et al. 1994] and their code generator uses many of the architectural and coding principles described in that paper. Our model predicts two types of L1 kernel and, for IBM platforms, we have an efficient implementation of each. ATLAS literature does mention this second kernel, stating that either kernel could be used and it was an arbitrary choice on their part to generate the one so selected. However, they did not pursue including this second kernel, nor did they justify their conclusion, that both kernel routines were basically the same.

Most practical matrix-matrix multiply L1 kernel routines have the form that our model predicts. For example, ESSL's kernel for the RISC-based RS/6000 processors, since their inception in 1990, have used routines that conform to this model. The same is true of the kernel for Intel's CISC Pentium III processor, which is described in an expanded version of this paper. Since ATLAS's code generator for its L1 kernel also fits our model and has shown cross-platform success, we can expect that our model will work on other platforms as well.

For the L1 level of memory, our model predicts that one should load most of the L1 cache with either the A or the B matrix operand. The other operands, C, and B or A, respectively, are streamed into and out of (through) the remainder of the L1 cache while the large A or B operand remains consistently cache-resident. Another theory predicts that each operand should be square and occupy one-third of the L1 cache. In this regard, we mention that ATLAS only uses its L1 kernel on square matrix operands. Hence the maximum operation count (multiply-add)

that an invocation of the ATLAS kernel can achieve is NB^3. Our (first) model places the A operand, of size $MB \times KB$, into the L1 cache, filling most of available memory at that level. However, we can stream N, nb-sized blocks of the remaining operands through the L1 cache. By selecting nb based on the register sizes, we can allow N to be, essentially, infinite. Thus, the streamed form of our L1 kernel can potentially support $2 \times MB \times KB \times N$ flops per invocation. We observe two practical benefits from our form of kernel construction and usage:

(1) A rectangular blocking where $MB < KB$ leads to a higher FLOP rate, due to the inherent asymmetry that results from having to load and store C.
(2) The streaming feature allows a factor of N/NB fewer invocations of the kernel routine.

Now we turn to partitioning the matrices, A, B, and C, into comforming submatrix blocks. ATLAS's model advises only two such partitioning strategies: (1) J, I, L and (2) I, J, L, where the outer loop increments are the smallest and the inner loop, the largest. Further, for both partitionings, the ATLAS model only allows a single square blocking factor of size NB. By having only two such partitionings ATLAS documentation states that it can only block for one other memory level, for example, L2, and that their method for doing so is only approximate. Our model differs from the one employed by ATLAS in that our model has three potential blocking factors, MB, NB, and KB, at *every* cache level of the memory hierarchy.

Strangely, our model does **not** predict the two partitionings selected by ATLAS. The reason for this is that ATLAS's partitionings use K as the "streaming" parameter. In our model, the blocking parameter, KB, would be tiny. This would lead to a `DAXPY`-like kernel which is known to be inferior to a `DDOT`-like kernel because the former continually loads and stores the C operand, whereas the latter keeps the C operand in registers.

4.2 Goto BLAS

Presently the `DGEMM` provided by [Goto] gives very high performance on a variety of platforms. We think our model encompasses all the principles espoused in [Goto and vandeGeijn 2002]. TLB blocking is automatically handled when one performs data copy via the use of `NDS`. Our `RPM` B kernel is used in [Goto and vandeGeijn 2002]. And our `RMP` A kernel is probably not used in [Goto and vandeGeijn 2002] because then data copy of C is required to avoid `TLB` misses. The third and second last paragraphs of the Conclusion of [Goto and vandeGeijn 2002] corroborates the statements made above.

5. SUMMARY AND CONCLUSION

This paper extends the results of [Gunnels et al. 2001] by introducing the concept of conservation of matrix operand sizes. By doing so, we show that the number of algorithms reduces from 2^L to four. It emphasizes the importance of streaming and generalizes it from L2 to L1 caches to caches $h+1$ to h for all $h > 1$. Because of space limitations the role of `NDS` via data copy is not covered as well as our two kernel routines that feature streaming. Finally, our model is claimed to encompass the principles of Goto BLAS as described in [Goto and vandeGeijn 2002].

REFERENCES

Agarwal, R. C., Gustavson, F., and Zubair, M. 1994. Exploiting functional parallelism on Power2 to design high-performance numerical algorithms. *IBM Journal of Research and Development 38,* 5, 563–576.

Bilmes, J., Asanović, K., whye Chin, C., and Demmel, J. 1997. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*. Vienna, Austria.

Bilmes, J., Asanović, K., whye Chin, C., and Demmel, J. 1998. The PHiPACv1.0 matrix-multiply distribution. Tech. Rep. 98-35, Int'l Computer Science Institute. October.

Corporation, I. 1986. Essl guide and reference for ibm es/3090 vector multiprocessors. order no. sa22-7220, feb. 1986.

Elmroth, E., Gustavson, F., Jonsson, I., and Kagstrom, B. 2004. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review 46,* 1, 3–45.

Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. 1999. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 285.

Gallivan, K., Jalby, W., Meier, U., and Sameh, A. 1987. The impact of hierarchical memory systems on linear algebra algorithm design. CSRD Report 625, Center for Supercomputing Research and Development, University of Illinois. Sept.

Goto, K. Goto blas.

Goto, K. and vandeGeijn, R. 2002. On reducing tlb misses in matrix multiplication. Flame working note # 9, Univ. of Texas. November.

Gunnels, J. A., Henry, G. M., and van de Geijn, R. A. 2001. A family of high-performance matrix multiplication algorithms. In *Computational Science - ICCS 2001, Part I*, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. K. Tan, Eds. Lecture Notes in Computer Science 2073. Springer-Verlag, 51–60.

Gustavson, F. G. 2001. New generalized matrix data structures lead to a variety of high-performance algorithms. In *The Architecture of Scientific Software*, R. F. Boisvert and P. T. P. Tang, Eds. Kluwer Academic Press.

Hong, J. and Kung, H. 1981. complexity: the red-blue pebble game.

Toledo, S. 1999. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms and Visualization*, J. Abello and J. S. Vitter, Eds. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 161–180.

Whaley, R. C. and Dongarra, J. J. 1998. Automatically tuned linear algebra software. In *Proceedings of SC'98*.

Received Month Year; revised Month Year; accepted Month Year