On Reducing TLB Misses in Matrix Multiplication

FLAME Working Note #9

Kazushige Goto
Robert van de Geijn

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
{kgoto,rvdg}@cs.utexas.edu

November 1, 2002

Abstract

During the last decade, a number of projects have pursued the high-performance implementation of matrix multiplication. Typically, these projects organize the computation around an "inner kernel," $C = A^T B + C$, that keeps one of the operands in the L1 cache, while streaming parts of the other operands through that cache. Variants include approaches that extend this principle to multiple levels of cache or that apply the same principle to the L2 cache while essentially ignoring the L1 cache. The intent is to optimally amortize the cost of moving data between memory layers.

The approach proposed in this paper is fundamentally different. We start by observing that for current generation architectures, much of the overhead comes from Translation Look-aside Buffer (TLB) table misses. While the importance of caches is also taken into consideration, it is the minimization of such TLB misses that drives the approach. The result is a novel approach that achieves highly competitive performance on a broad spectrum of current high-performance architectures.

1 Introduction

It is somewhat surprising that after decades of research into the optimal implementation of matrix multiplication, papers on the subject still appear with great regularity. Matrix multiplication continues to be of importance because a broad range of high-performance packages that support directly or indirectly scientific computation depend, to a large degree, on the performance of the matrix multiplication kernel [3, 13, 28, 5]. New contributions continue to be made because the gap between the performance of the CPU and the bandwidth to the memory continues to widen and new architectural features are introduced into computers, which require new techniques or refinements of old techniques, for matrix multiplication.

Two observations are fundamental to our approach:

- The ratio between the rate at which floating point computation can be performed by the floating point unit(s) and the rate at which floating point numbers can be streamed from the L2 cache is typically relatively small.
- Thus, it is the cost for starting the streaming of data from the L2 cache that represents a significant overhead.
- A large component of the startup cost of the streaming of data cames from Translation Look-aside Buffer (TLB) misses since these inherently stall the CPU.

By taking these observations into account, the contribution of this paper is that by casting the matrix multiplication in terms of an inner kernel that performs the operation $C = \hat{A}^T B + C$, where $\hat{A}$ fills most of memory addressable by the TLB table and C and B are computed a few columns at a time,

- TLB misses can be largely avoided,
- the cost of the TLB misses that do occur can be amortized of a large amount of computation, and
- the cost of transposing submatrices so that the overall matrix multiplication can be cast into this inner kernel is amortized over a large amount of computation.

In practice, these observations lead to implementations that attain extremely high performance.

It can be argued that the exact nature of the new contribution of this paper is hard to identify. Much of what we present has been incorporated in one form or another in other implementations of matrix multiplication. It can also be argued that it is already known as street-wisdom and/or is incorporated in proprietary libraries that keep the details of the implementation a trade-secret. We would like to think that at the very least this paper exposes some of the issues explicitly and thereby makes a contribution the body of knowledge in this area. The fact that the method leads to consistently higher performance than achieved by competing implementations provides some support for this view.

The structure of this paper is as follows: In Section 2 we discuss research related to the high-performance implementation of matrix multiplication. Basic architectural considerations are given in Section 3. Observations that show the importance of the TLB are given in Section 4. These observations are translated to a practical implementation in Section 5. In Section 6 we report performance results from implementations on various architectures. Concluding remarks follow in the final section.

2 Related Work

The addition of a cache memory to vector architectures required library developers to reformulate linear algebra libraries that had been written in terms of vector operations.. To obtain high performance on these new machines, both vector operations and blocking to take advantage of the cache was necessary. IBM's ESSL library included block-based vector algorithms for a number of linear equation solvers that were part of LINPACK [7], including LU and Cholesky based solvers for dense and banded matrices [21]. These implementations were based on highly optimized linear algebra routines that performed blocking together with an inner kernel that vectorized the linear algebra operation on blocks that fit in the cache memory.

It wasn't until the late 1980s that, with the introduction of the Cray 2, which also combined vector processing with a cache memory, there was a strong impetus in the linear algebra library community to standardize a new interface to a set of matrix-matrix operations, the level-3 BLAS [8]. The primary purpose of this new set of routines was to support newly proposed libraries such as LAPACK [6, 2, 3]. By casting the bulk of computation in terms of matrix-matrix operations, which perform $O(n^3)$ operations on $O(n^2)$ data, blocks of data could be moved in and out of the data cache while amortizing the cost of this movement over a large number of computations. The substantial task of providing all levels of BLAS was pushed onto the vendors. The reward was that numerically stable libraries like LAPACK then provided high-performance across a large variety of architectures.

By the early 1990s, it was recognized that as architectures were becoming increasingly complex the task of providing a complete set of (especially level-3) BLAS was becoming a substantial burden on the vendors. Fortunately, it was shown that high-performance level-3 BLAS could be coded to be portable by casting these operations in terms of matrix multiplication [23, 16, 24, 13]. This reduced the cost of implementing the level-3 BLAS to the cost of implementing matrix multiplication. Next, it was recognized that by combining a blocking strategy with a carefully crafted inner kernel, which performs matrix multiplication with blocks that are roughly of a size so that they fit in the cache memory, the cost of implementing the level-3 BLAS could be reduced to the cost of implementing this inner kernel. At IBM the idea of designing the architecture for this approach to coding the matrix multiply and other algorithms, referred to as *Algorithms and Architectures*, was both expounded and applied to the development of the IBM POWER2 architecture in conjunction with the ESSL library for that architecture [1]. By also designing compilers specifically for this combination of algorithms and architecture, the implementations of the BLAS could be coded in FORTRAN rather than in assembly code.

By the late 1990s architectures with multiple levels of cache memory were being introduced. With it came a recognition that the implementation of matrix multiplication for a given architecture had become, and would remain, a formidable task [12]. Based on the work at IBM that coded such operations in FORTRAN, the PHiPAC project at Berkeley pursued the portable implementation of matrix multiplication in a high-level language, C [4]. The idea behind that project was to automatically generate code so that in combination with an exhaustive search, the optimal blocking of the operands and optimal ordering of the loops could be discovered. The different blocking schemes were intended to automatically detect optimal blockings for the different caches while the different loop orderings would automatically detect how the movement of blocks between memory layers could be best amortized over computation. In addition, the inner kernel was automatically generated so that the number of registers and depth of pipelines could be detected. At the expense of an optimization process that often took days or even weeks to complete, remarkable performance was observed.

The ATLAS [29] project at the University of Tennessee refined the techniques developed as part

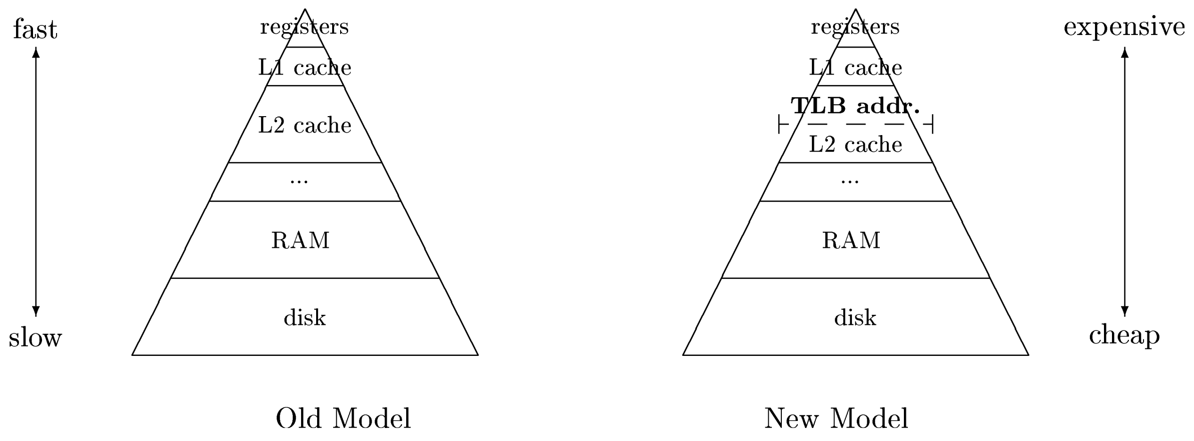


Figure 1: The hierarchical memories viewed as a pyramid. Under the new model, the memory that is addressable by the TLB is explicitly exposed.

of the PHiPAC project by constraining the number of different implementations that are generated as part of the search process. As a result, the optimization process completes more quickly, typically in a matter of hours.

In a recent paper [14] a family of algorithms based on a model of the memory hierarchy was introduced. The model predicts, and preliminary experiments with an implementation for the Intel Pentium (R) III processor show, that at a given level of the memory the blocking of the matrices and order of the loops is dictated by the shapes of the operands together with the size of memory layer one level above (in the pyramid).

Recently, algorithms that automatically block for caches by formulating the algorithms to be recursive have received a great deal of attention for matrix multiplication and many other important computations such as matrix factorizations [11, 17, 29, 22, 15, 26, 19]. Others have focused on (also) applying "recursion" to produce new data formats for matrices, instead of the traditional FORTRAN and C data structures [27]. Our view is that recursion is very powerful and excellent results are obtainable. The techniques presented in this paper are in some sense orthogonal to those addressed by recursion in data storage and algorithm implementation.

3 Basic Architectural Considerations

In this section we present, at a high level of abstraction, some of the architectural features of a typical modern microprocessor.

The memory hierarchy of a modern microprocessor is often viewed as the pyramid given in Fig. 1. At the top of the pyramid, there are the processor registers, with extremely fast access. At the bottom, there are disks and even slower media. As one goes down the pyramid, the amount of memory increases as does the time required to access that that memory, while the financial cost of memory decreases.

A second architectural consideration relates to the page management system. A typical modern architecture uses virtual memory so that the size of usable memory is not constrained by the size of the physical memory. Memory is partitioned in pages of some (often fixed) prescribed size. A table, referred to as the *page table* maps virtual addresses to physical addresses and keeps track of whether as page is in memory or on disk. The problem is that this table itself could be large (many Mbytes) which hampers speedy translation of virtual addresses to physical addresses. To overcome this, a smaller table, the Translation Look-aside Buffer (TLB), that stores information about the most recently used pages, is kept. Whenever a virtual address is found in the TLB, the translation is fast. Whenever it is not found (a TLB miss occurs), the page table is consulted and the resulting entry is moved from the page table to the TLB.

The most significant difference between a cache miss and a TLB miss is that a cache miss does not necessarily stall the CPU. A small number of cache misses can be tolerated by using algorithmic prefetching techniques as long as the data can be read fast enough from the memory where it does exist and arrives at the CPU by the time it is needed for computation. A TLB miss, by contrast, causes the CPU to stall until the TLB has been updated with the new address. In other words, prefetching can mask a cache miss but not a TLB miss.

4 Emphasizing the TLB

Consider the multiplication $C = AB + C$. Partition

$$(1)\quad C = \left(\begin{array}{c|c|c} C_{11} & \cdots & C_{1N} \\ \hline \vdots & & \vdots \\ \hline C_{M1} & \cdots & C_{MN} \end{array}\right), A = \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \\ \hline \vdots & & \vdots \\ \hline A_{M1} & \cdots & A_{MK} \end{array}\right), \text{ and } B = \left(\begin{array}{c|c|c} B_{11} & \cdots & B_{1N} \\ \hline \vdots & & \vdots \\ \hline B_{K1} & \cdots & B_{KN} \end{array}\right)$$

where the partitionings are conformal so that

$$C_{ij} = \sum_{p=1}^{K} A_{ip}B_{pj} + C_{ij}.$$

The following loop ordering will compute the multiplication:

Algorithm 1

```
for $i = 1 : M$
    for $p = 1 : K$
        for $j = 1 : N$
            $C_{ij} = A_{ip}B_{pj} + C_{ij}$
        endfor
    endfor
endfor
```

A typical approach to optimizing matrix multiplication starts by writing an inner kernel to compute $C_{ij} = A_{ip}B_{pj} + C_{ij}$. This approach has the property that the CPU attains near-optimal performance when A_{ip} remains in the L1 cache and elements of C_{ij} and B_{pj} are streamed for a lower level in the memory pyramid. The dimensions of A_{ip} are optimized so that this inner kernel attains the best performance. Finally, Some loop is created to compute all submatrices of C. Beyond this basic approach, there are some options. It is often beneficial, especially if A_{ip} is embedded in a matrix with a large leading dimension, to pack it into contiguous memory so that TLB misses are reduced. Also, it is often beneficial to transpose A_{ip} so that accesses to memory are contiguous when inner-products of columns of A_{ip}^T and B_{pj} are computed to update elements of C_{ij}.

Let us present Algorithm 1 as

Algorithm 2

```
for $i = 1 : M$
    for $p = 1 : K$
        $\left( C_{i1} \mid \cdots \mid C_{iN} \right) = A_{ip} \left( B_{p1} \mid \cdots \mid B_{pN} \right) + \left( C_{i1} \mid \cdots \mid C_{iN} \right)$
    endfor
endfor
```

Let us make the following assumptions and observations. Notice that we do not proclaim these assumptions and observations to reflect the absolute truth. They will provide a point of departure for discussion.

1. If we can optimize the individual computation

   $$\left(C_{i1} \mid \cdots \mid C_{iN} \right) = A_{ip} \left(B_{p1} \mid \cdots \mid B_{pN} \right) + \left(C_{i1} \mid \cdots \mid C_{iN} \right), \tag{2}$$

   we are in good shape.

2. In order to optimize (2) it is beneficial to transpose $\hat{A} = A_{ip}^T$ and compute

   $$\left(C_{i1} \mid \cdots \mid C_{iN} \right) = \hat{A}^T \left(B_{p1} \mid \cdots \mid B_{pN} \right) + \left(C_{i1} \mid \cdots \mid C_{iN} \right)$$

   instead. This observation comes from the fact that this allows inner products of columns of A_{ip} and B_{pj} to be computed while accessing memory contiguously. It also prevents severe thrashing of the L1 cache.

3. It is important to be able to complete a loop through all entries of $\hat{A}$ without creating a major bubble in the stream of data and computation. One way to satisfy this assumption is to store $\hat{A}$ contiguously while ensuring that accessing $\hat{A}$ does not create a TLB miss.

4. A prominent overhead comes from the cost of accessing C_{ij} and B_{pj} the first time as part of the computation $C_{ij} = \hat{A}^T B_{pj} + C_{ij}$. We will assume that this cost includes a startup (latency) cost as well as a cost proportional to the size of B_{pj}. A large part of the latency cost lies with the cost of the TLB misses associated with the first time that C_{ij} and B_{pj} are accessed. By picking B_{pj} and C_{ij} to have a relatively large row dimension, this startup cost is amortized over many elements of C_{ij} and B_{pj}. However, it is important to ensure that B_{pj} fits in the L1 cache so that the streaming of data from

5. If data is streamed so that the CPU does not stall, a second overhead that reduces performance comes from transposing (or packing) A_{ip} into $\hat{A}$.

The conclusion is that $\hat{A}$ should be relatively square and fill most of the L2 cache. Submatrices C_{ij} and B_{pj} should be relatively narrow since this means fewer entries of the TLB are devoted to those submatrices.

5 A Practical Approach

Let us examine how the above considerations affect the implementation of matrix multiplication on a current generation microprocessor like the Intel Pentium (R) 4.

We observe that on such architecture the bandwidth between the L2 cache and the registers is such that in the time it takes to load a floating point number from the L2 cache into a register, only a few floating point operations (often only a single one) can be performed once a pipeline has been established. Let us, for the sake of argument, assume that once pipelines are filled, the ratio between the cost of such a load and computation is actually one. Now, provided pipelines can be kept full, the following approach will attain high performance:

1. Partition C, A, and B as in (1), but pick C_{ij} and B_{pj} to be comprised of only a single column:

$$(3) C = \left(\begin{array}{c|c|c} c_{11} & \cdots & c_{1n} \\ \hline \vdots & & \vdots \\ \hline c_{M1} & \cdots & c_{Mn} \end{array} \right), A = \left(\begin{array}{c|c|c} A_{11} & \cdots & A_{1K} \\ \hline \vdots & & \vdots \\ \hline A_{M1} & \cdots & A_{MK} \end{array} \right), \text{ and } B = \left(\begin{array}{c|c|c} b_{11} & \cdots & b_{1n} \\ \hline \vdots & & \vdots \\ \hline b_{K1} & \cdots & b_{Kn} \end{array} \right).$$

   Notice that elements of c_{ij} and b_{pj} will be contiguous in memory.

2. Consider the computation

$$\left(\begin{array}{c|c|c} c_{i1} & \cdots & c_{in} \end{array} \right) = A_{ip} \left(\begin{array}{c|c|c} b_{p1} & \cdots & b_{pn} \end{array} \right) + \left(\begin{array}{c|c|c} c_{i1} & \cdots & c_{in} \end{array} \right).$$

   Let us implement this by first transposing $\hat{A} = A_{ip}^T$ and then computing

$$(4) \qquad \left(\begin{array}{c|c|c} c_{i1} & \cdots & c_{in} \end{array} \right) = \hat{A}^T \left(\begin{array}{c|c|c} b_{p1} & \cdots & b_{pn} \end{array} \right) + \left(\begin{array}{c|c|c} c_{i1} & \cdots & c_{in} \end{array} \right).$$

3. *If*

   (a) $\hat{A}$ is packed to be in contiguous memory,

   (b) The transposition of A_{ip}, $\hat{A} = A_{ip}^T$, is carefully ordered,

   (c) The first element of $\hat{A}$ is aligned to a page,

   (d) $\hat{A}$ and, for all j, c_{ij}, $c_{i(j+1)}$, b_{pj}, and $b_{p(j+1)}$ together do not overflow the TLB table,

   (e) $\hat{A}$ fits in the L2 cache, and

   (f) (4) is computed by the loop

```
for $j = 1 : n$
    $c_{ij} = \hat{A}^T b_{pj} + c_{ij}$
endfor
```

then, **in principle**,

- $\hat{A}$ will be loaded into the L2 cache, and the TLB, during the transposition $\hat{A} = A_{ip}^T$.
- Once the pages corresponding to $\hat{A}$ have been loaded into the L2 cache and TLB they will remain there during the duration of the computation in (4).
- *The streaming of the data should allow the computation of each individual $\hat{A}^T b_{pj} + c_{ij}$ to achieve optimal performance.*

In practice, a few modification may have be made to the approach. For example, some TLB entries may be used by data associated with indexing or the code being executed.

Note 1 *If the number of floating point operations that can be performed during the loading of a floating point operation (once streaming is established) is greater than two, the bandwidth between the L2 cache and the registers becomes a bottleneck. Let us assume the ratio equals the integer R. Then the above scheme must be modified so that b_{pj} and c_{ij} consist of R columns. In this case, for every element of $\hat{A}$ that is loaded, $2 * R$ flops can be performed once that element reaches the registers. Notice that as R increases, the number of TLB entries devoted to B_{pj} and C_{ij} increases, which means that the size of $\hat{A}$ may have to be reduced. More specifically, the R should be chosen so that*

$$R \geq \frac{\textit{Rate in flops per cycle}}{2 \times \textit{Bandwidth in double words per cycle between L2 and registers}} \tag{5}$$

Note 2 *The number of registers that can be used for computation and prefetching play an important role in the proposed scheme. If there aren't enough registers to support pipeline streaming the scheme breaks down.*

Note 3 *The following steps can be used to determine approximations for the various parameters:*

1. *Determine the size of the TLB table, T.*
2. *Determine R of Note 1.*
3. *The number of TLB entries used for $\hat{A}$ should not exceed $T - 4R$. The reason for this is that generally, C_{ij} and B_{pj} have, together, $2R$ columns, which typically require $2R$ TLB entries (provided a column isn't split between two pages). In order to not corrupt any TLB entries devoted to $\hat{A}$ another $2R$ entries are required for when $C_{i(j+1)}$ and $B_{p(j+1)}$ are first accessed.*
4. *The size (footprint) of $\hat{A}$ is now picked to not exceed $T - 4R$ pages of memory.*
5. *Under the constraint given in Item 4, the row and column dimensions of $\hat{A}$ are determined experimentally.*

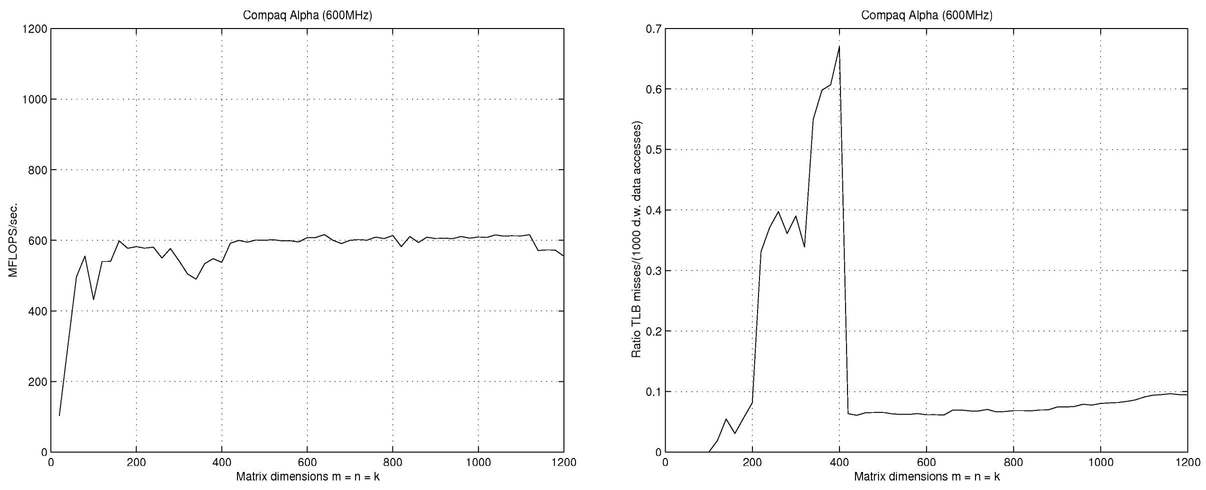


Figure 2: Performance of the ATLAS `dgemm` matrix multiplication on the Compaq Alpha 21164 (600 MHz). **Left:** Performance. **Right:** Rate of TLB misses relative to memory accesses.

6 Experimental Results

6.1 To copy or not to copy, that's the question

The results of our first experiment, reported in Fig. 2, show the importance of copying and transposing blocks of matrix A, $\hat{A} = A_{ip}$. In that figure, we show the performance of the ATLAS `dgemm` implementation (release R3.2.1), on a Compaq workstation equipped with an Alpha 21164 (600 MHz) processor. On the left, we see that once the matrix dimensions are of reasonable size, the performance is relatively smooth, independent of the matrix size. On the right, we report the number of TLB misses relative to the number of double words (d.w.) accessed during the matrix multiplication. We see that as the matrix size increases, the number of misses increases dramatically. However, once the dimensions of the matrices hit $m = n = k \approx 400$, the rate of TLB misses is dramatically reduced. We conclude that for small matrices, ATLAS neither copies nor transposes A_{ip}. Once the matrix size, or more importantly the leading dimension of A, becomes large to where TLB concerns become an issue, the implementation switches to one that copies and transposes.

6.2 Implementation on the Pentium (R) 4

Next, let us examine how the above considerations affect the implementation of a double precision real (64 bit) matrix multiplication on the Intel Pentium (R) 4 processor.

Of importance are the bandwidths between the different memory layers as well as the size of the TLB table. To verify parameters reported for the Intel Pentium (R) 4, we designed a simple experiment: An assembly-coded kernel was written that reads data into registers using software prefetch techniques to ensure that a constant stream is maintained when possible. This kernel was executed by repeatedly reading a fixed amount of data into the registers. The first time the data

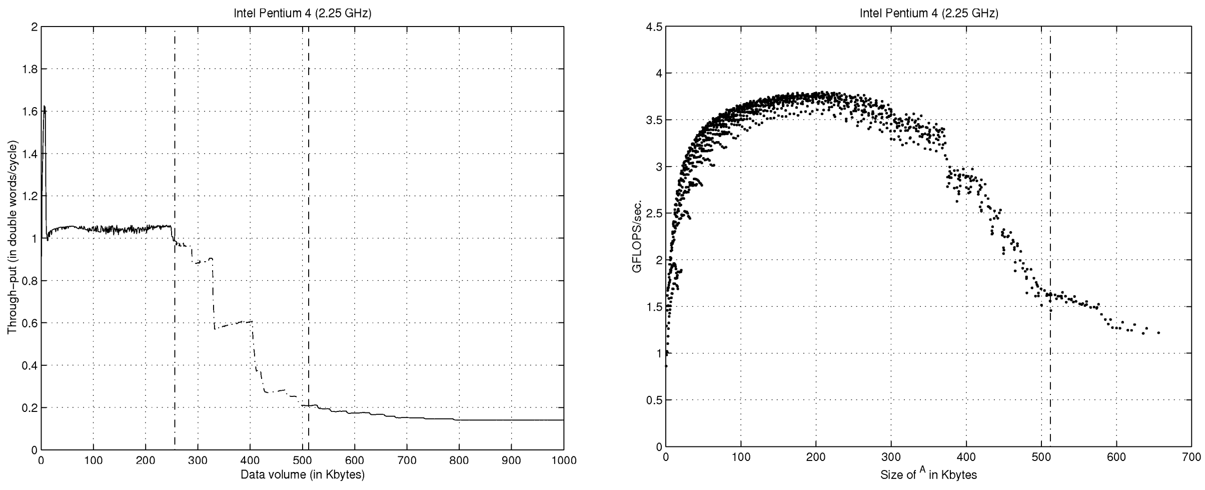


Figure 3: **Left:** Bandwidth between various parts of the memory hierarchy and the registers. **Right:** Performance when $m = n = k = 1000$ and $R = 2$, as a function of the footprint of $\hat{A}$.

was read, one would expect the data to be moved into the different layers of cache. If the amount of data being read was less than the size of a layer of cache, one would expect it to still be resident during the subsequent iterations and therefore the bandwidth attained during those subsequent iterations is an indication of the bandwidth between that layer of the cache and the registers. The results are reported in Fig. 3 (Left).

In Fig. 3 (Left), the initial spike corresponds to data residing in the L1 cache. Notice that while the advertised bandwidth between the L1 cache and registers is 2 double words per cycle (d.w./cycle), we observed a rate of 1.6 d.w./cycle with our kernel. Next, performance is steady at 1 d.w./cycle up to 256 Kbytes, which is where TLB misses can be expected to start occurring. After this, there is a decline in throughput up to 512 Kbytes, the size of the L2 cache. Notice that in this region, where the curve is marked with a dashed line, the measured throughput differed markedly from one trial run to the next. Finally, once the data is being streamed from main memory, the bandwidth is low, although steady. The pertinent details regarding the architectures are now summarized in Table 1.

Next, we report experiments to establish an optimal size for $\hat{A}$. From the table, we find that the parameter R mentioned in Note 1 should equal at least

$$R \geq \frac{\text{Rate in flops/cycle}}{2 \times \text{Bandwidth in d.w./cycle between L2 and registers}} = \frac{2}{2} = 1. \tag{6}$$

For the experiment, we fix parameter R in Note 1 to equal 2. This is greater than what we computed in (6) since in addition to reading matrix $\hat{A}$ from the L2 cache, we need to also read elements of columns of B_{pj} from the L1 cache and elements of C_{ij} from some other layer of memory which will likely keep us from attaining the optimal bandwidth from the L2 cache. In Fig. 3 (Right) we report the performance attained by our approach for different dimensions for $\hat{A}$ are chosen when the

Processor	Pentium (R) 4
Clock rate	2.25 GHz
Number of SSE2 registers	8
L1 cache	8 Kbytes
L2 cache	512 Kbytes
Bandwidth between L1 and registers	1.6 d.w./ cycle
Bandwidth between L2 and registers	1.0 d.w./ cycle
Bandwidth between RAM and registers	0.2 d.w./ cycle
Page size	4 Kbytes
TLB table size	64 entries
TLB accessible memory	256 Kbytes
Rate of computation	2 flops/cycle

Table 1: Architectural details of the test platform.

overall matrix multiply is fixed to be large ($m = n = k = 1000$). Notice that optimal performance is attained when $\hat{A}$ is 256×112 and thus occupies about 224 Kbytes (56 pages). It is interesting that the performance profiles of the two graphs in Fig. 3 in the range 256–512 Kbytes show some resemblance.

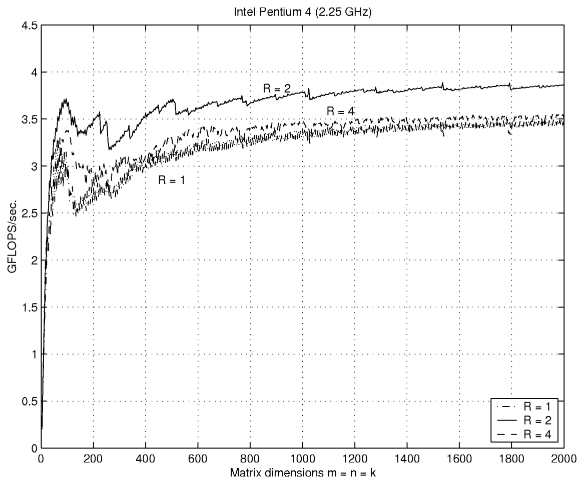
In Fig. 4 (Left) we demonstrate that picking R incorrectly affects performance. For the reported experiment, we picked $\hat{A}$ to equal 256×112. While the theory predicts that the algorithm should perform well with $R = 1$, in practice the bandwidth between the L2 cache and the registers is only achieved under idealized circumstances and thus the more conservative $R = 2$, which better amortizes the cost of bringing elements of $\hat{A}$ from the L2 cache to the registers, yields better performance.

6.3 Comparison with other libraries on assorted architectures

Finally, we compare the resulting performance against that attained by other high-performance implementations of DGEMM. Details related to the architectures and libraries tested are given in Table 2.

In Figs. 4 (Right), 5 (Left), 5 (Right), and 6 (Left), we report the performance attained on the Intel Pentium (R) 4, Compaq Alpha, IBM POWER 3, and Intel Pentium (R) III, respectively. In those graphs we compare against `dgemm` implementations provided by ATLAS and by the vendor. Notice that while for very small matrices the performance of our implementation suffers due to the copying of the submatrices, in general the performance is very competitive and smooth (insensitive to small changes in the dimension sizes).

In Fig. 6 (Right), we compare the performance of the presented approach against the ITXGEMM `dgemm` implementation developed jointly by researchers at UT-Austin and Intel [14, 20]. It is important to realize that ITXGEMM actually represents a family of algorithms. The specific implementation reported in [14] is such that $\hat{A}$ is chosen to be L1 cache resident, while B_{pj} and C_{ij}



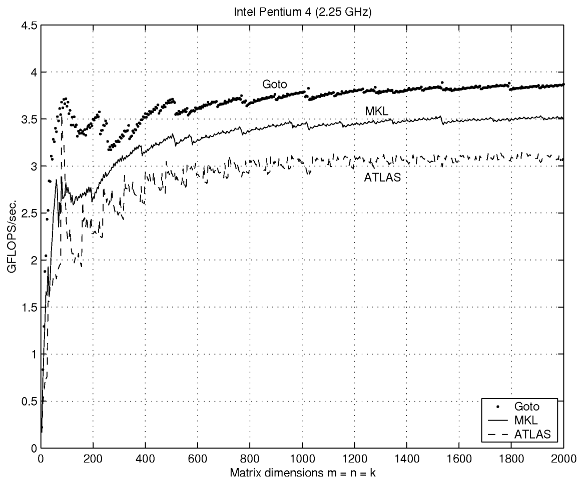
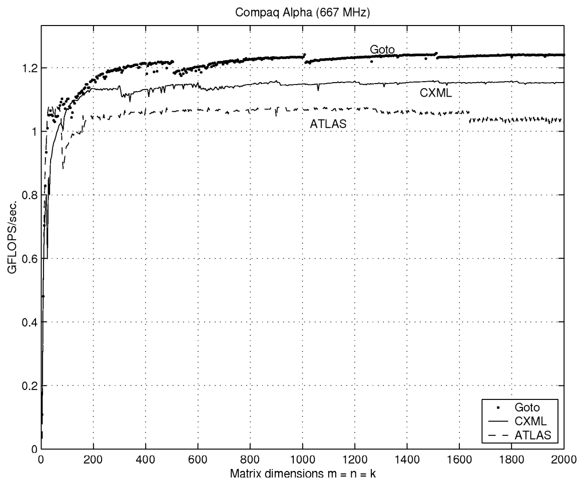


Figure 4: **Left:** Performance when $m = n = k$ for different choices of R. **Right:** Performance as a function of the matrix dimensions ($m = n = k$) of the presented approach (`Goto`), Intel's MKL library (`MKL`), and UT-Knoxville's ATLAS library (`ATLAS`).



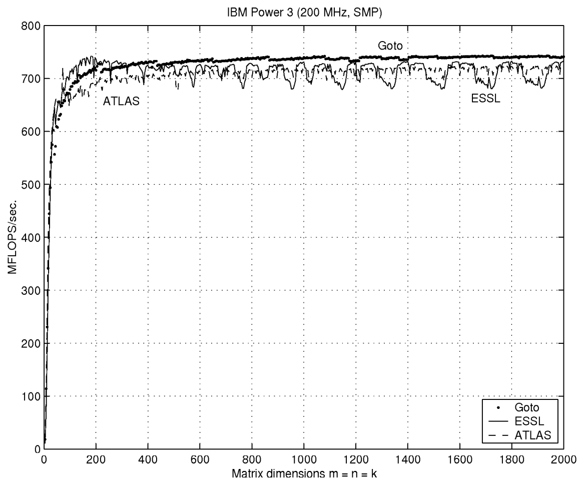


Figure 5: Performance as a function of the matrix dimensions ($m = n = k$) of the presented approach (`Goto`) and ATLAS on the Compaq Alpha (LEFT) and IBM POWER 3 (RIGHT). We also compare against Compaq's CXML and IBM's ESSL libraries.

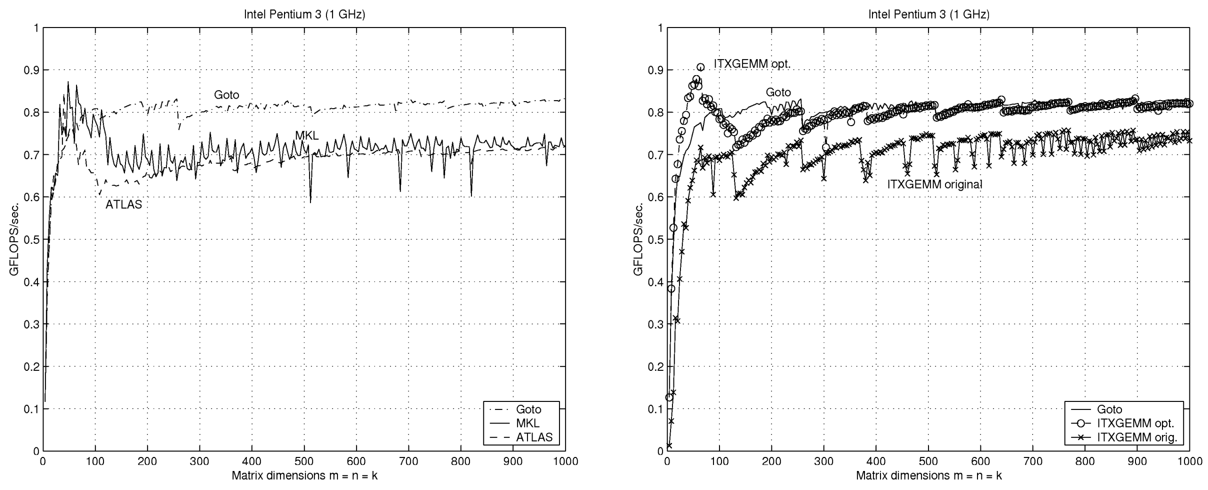


Figure 6: Performance as a function of the matrix dimensions ($m = n = k$) of the presented approach on the Intel Pentium (R) III. Left: Comparison with ATLAS and MKL. Right: Comparison with ITXGEMM.

Table 2: Details regarding the architectures and libraries targeted in Section 6.3.

	Architectures			
Processor	Pentium (R) 4	Alpha 21264	POWER 3	Pentium (R) III
Clock rate (MHz)	2,250	667	200	1,000
L1 cache	8 Kbytes	64 Kbytes	64 Kbytes	16 Kbytes
L2 cache	512 Kbytes	4 Mbytes	4 Mbytes	256 Kbytes
TLB size	64	128	256	64
page size	4 Kbytes	8 Kbytes	4 Kbytes	4 Kbytes
Cache line size	64 bytes	64 bytes	128 bytes	32 bytes
flops/cycle	2	2	4	1
Native library version	MKL 5.2	CXML 5.0.0	ESSL 3.30	MKL 5.2
ATLAS version	3.4.1	3.3.14	3.4.1	3.4.1

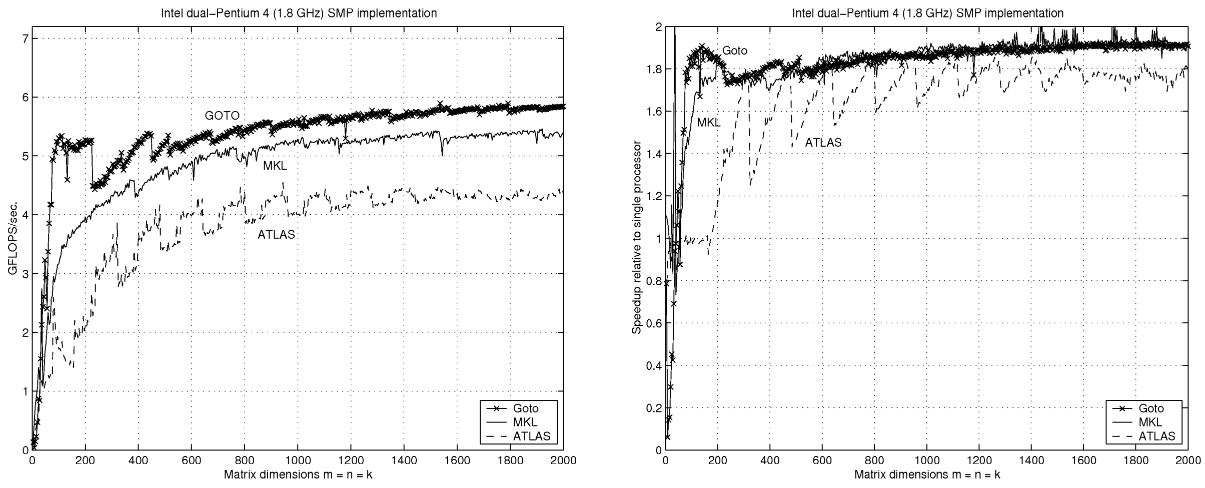


Figure 7: Performance of dual processor implementations of the different libraries on a dual-Pentium (R) 4 processor based system. Left: Performance attained. Right: Speedup relative to the performance of a single processor.

are streamed through the L1 cache.

For the Pentium (R) III processor, making $\hat{A}$ L1 cache resident leads to an inner kernel that achieves higher performance, as is evident from the spike in performance in Fig. 6 (Right) when the matrix sizes are relatively small. The optimal dimensions of $\hat{A}$ are 64×8. This implies that eight elements of C are computed before moving on to the next column of C. This in turn means that, once the leading dimension of C becomes large, the TLB miss that is likely incurred when accessing a new column of C is not amortizes over much computation. The solution is to make the current part of C being computed contiguous, which in turn requires this current part to be added to the appropriate part of C upon completion of its computation. This additional operation creates considerable overhead that decreases the overall performance that can be attained. In addition, the transposition of $\hat{A}$ is not amortized over as much computation, which translates to a higher overhead for this transposition. Interestingly enough, for larger matrices the two approaches attain nearly identical performance.

Note 4 *In Fig. 6 (Right), the curve labeled* `ITXGEMM original` *represents the implementation that was used to collect data for the paper on that method [14]. The curve labeled* `ITXGEMM opt.` *represents an optimized version of that method, where the only optimization came from an improved routine for copying and/or transposing data.*

Our attempts to match the performance of the presented approach on a Pentium (R) 4 processor with an implementation of ITXGEMM for that processor was not successful. We believe that is due to the very small L1 data cache on that processor, in combination with the relatively small difference between the bandwidth to the registers from the L2 cache and L1 cache.

Table 3: Performance attained by the HPL implementation of the Massively-Parallel LINPACK benchmark.

	Matrix Dimension	Performance
HPL + our DGEMM	42200	1.00 TFLOPS
(`NB=104, P=20, Q=30`)	253400	2.00 TFLOPS
HPL + ATLAS	42200	0.41 TFLOPS
(`NB=80, P=20, Q=30`)	253400	1.47 TFLOPS

6.4 SMP implementation

The described method can easily be used to construct an SMP implementation of DGEMM. For reference, in Fig. 7 we compare the resulting performance with that of other libraries on a dual-Pentium (R) 4 processor based system. In that picture, we show both the raw performance attained as well as the speedup of the dual processor implementation relative to the single processor implementation. For each curve, the ratio between the dual- and single-processor implementations of the same library are given.

6.5 Impact

One easily measurable impact of the described approach can be summarized by looking at its effect on the performance attained by the Massively Parallel LINPACK Benchmark (MP-LINPACK) [10]. The LINPACK benchmark measures the performance attained by a given architecture when solving a linear system of equations in 64-bit arithmetic via an LU factorization with partial pivoting. High-performance implementations cast the LU factorization in terms of matrix multiplication [9]. The HPL implementation of this benchmark was used for this experiment [25].

Our approach to implementing DGEMM was used to benchmark the 300 compute node, dual-Pentium (R) 4 processor (2.4 GHz) based, cluster at the Center for Computational Research at the University at Buffalo, SUNY. This machine has a theoretical peak performance of around 2.9 TFLOPS/sec. (2.9×10^{12} floating point operations per second). Table 3 summarizes the benchmark results. The indicated block (`NB`) and grid (`P` and `Q`) sizes were obtained through extensive experimentation. For details on the meaning of these parameters, see [25].

7 Conclusion

There are at least two ways that the insights in this paper can be interpreted.

If one views the kernel that computes the operation in (2) as the inner kernel, then one can view the contribution of this paper to be that by considering architectural features such as the TLB, and bandwidth between the different memory layers and the registers, that the level of cache in which data resides on which the kernel operates (the level of cache in which A_{ip} is chosen to reside) should be as low in the memory hierarchy pyramid as possible in an effort to optimally amortize the cost of copy and transpose operations required to make data contiguous in memory.

While one can argue that automated systems like PHiPAC and ATLAS should detect the presented approach as part of their optimization effort. Indeed, ATLAS appears to yield an inner-kernel for the Intel Pentium (R) 4 architecture that "by-passes" the L1 cache by placing making $\hat{A}$ L2 cache resident. Nonetheless, the resulting implementation is considerably slower than the presented approach. It should be possible to use our insights to refine these automated systems.

A second way to interpret the method is to view the operation $C_{ij} = A_{ip}B_{pj} + C_{ij}$ as the inner kernel, where B_{pj} is taken to be L1 cache resident and A_{ip} is accessed by rows (or as columns of $\hat{A}$) from the L2 cache. This interpretation makes the method a member of the family of matrix multiplication algorithms proposed in [14]. Notice that that paper underlies the ITXGEMM implementation discussed in the experimental section. However, for those experiments, a different member of the family was implemented for the Pentium (R) III processor. When viewing the presented method like this, the importance of adding TLB considerations and pipelining explicitly to the model that underlies ITXGEMM becomes apparent.

An observation in the paper that is relatively subtle is the importance of optimizing the matrix copy and transpose routines. This is not generally discussed in papers about optimizing matrix multiplication and we believe that in some cases incorrect conclusions have been made as a result of a failure to properly optimize these routines. This is particularly obvious for the ITXGEMM results in Fig. 6 (Right). It also supports the view that alternative schemes for storing matrices have merit as also observed in [18].

Regardless of how one interprets the results, the fact that the implementations attain better performance than those attained by other high-quality efforts demonstrates that there is value in the approach.

Additional Information

For additional information visit `http://www.cs.utexas.edu/users/flame/goto/`

Acknowledgments

We are indebted to Dr. Fred Gustavson and Dr. John Gunnels for their comments on an earlier draft. Performance numbers for the Massively Parallel LINPACK Benchmark were gathered with the 300 compute node Dell Cluster at the Center for Computational Research at the University at Buffalo, SUNY. We thank Dr. Matthew D. Jones for his efforts in obtaining these numbers.

References

[1] R.C. Agarwal, F.G. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, 38(5), Sept. 1994.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings Supercomputing '90*, pages 2–11. IEEE Computer Society Press, Los Alamitos, California, 1990.

[3] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.

[4] Jeff Bilmes, Krste Asanović, Chee Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[5] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Comput. Soc. Press, 1992.

[6] James Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen. Prospectus for the development of a linear algebra library for high-performance computers. Technical Report MCS-TM-97, Argonne National Laboratory, Sept. 1987.

[7] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, 1979.

[8] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[9] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA, 1991.

[10] J.J. Dongarra. Performance of various computers using standard linear equations software, (LINPACK benchmark report). University of Tennessee Computer Science Technical Report CS-89-85, Oct. 2002.

[11] E. Elmroth and F.G. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. Develop.*, 44(4):605–624, 2000.

[12] Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed Sameh. The impact of hierarchical memory systems on linear algebra algorithm design. CSRD Report 625, Center for Supercomputing Research and Development, University of Illinois, Sept. 1987.

[13] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

[14] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C.J. Kenneth Tan, editors, *Computational Science - ICCS 2001, Part I*, Lecture Notes in Computer Science 2073, pages 51–60. Springer-Verlag, 2001.

[15] F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, 1997.

[16] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Superscalar GEMM-based level 3 BLAS – the on-going evolution of a portable and high-performance library. In B. Kågström et al., editor, *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science 1541, pages 207–215. Springer-Verlag, 1998.

[17] F.G. Gustavson and I. Jonsson. Minimal storage high-performance Cholesky factorization via blocking and recursion. *IBM J. Res. Develop.*, 44(6):823–850, November 2000.

[18] Fred G. Gustavson. New generalized matrix data structures lead to a variety of high-performance algorithms. In Ronald F. Boisvert and Ping Tak Peter Tang, editors, *The Architecture of Scientific Software*. Kluwer Academic Press, 2001.

[19] Greg Henry. BLAS based on block data structures. Theory Center Technical Report CTC92TR89, Cornell University, Feb. 1992.

[20] Greg M. Henry. Flexible high-performance matrix multiply via self-modifying runtime code. Technical Report CS-TR-01-44, Department of Computer Sciences, The University of Texas at Austin, December 2001. `http://www.cs.utexas.edu/users/flame/pubs/`.

[21] IBM. *Engineering and Scientific Subroutine Library, Guide and Reference, Release 3.* Fourth Edition (Program Number 5668-863), 1988.

[22] I. Jonsson and B. Kågström. Recursive blocked algorithms for solving triangular matrix equations - part I: One-sided and coupled Sylvester equations. Department of Computing Science and HPC2N Report UMINF-01.05, Umeå University, 2001.

[23] B. Kågström, P. Ling, and C. Van Loan. Gemm-based level 3 blas: High-performance model, implementations and performance evaluation benchmark. LAPACK Working Note #107 CS-95-315, Univ. of Tennessee, Nov. 1995.

[24] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft.*, 24(3):268–302, 1998.

[25] A. Petitet, R.C. Whaley, J. Dongarra, and A. Cleary. Hpl – a portable implementation of the high-performance LINPACK benchmark for distributed–memory computers. `www.netlib.org/benchmark/hpl/`.

[26] S. Toledo. Locality of reference in lu decomposition with partial pivoting. *SIAM Journal on Matrix. Anal. Appl.*, 18(4), 1997.

[27] Vinod Valsalam and Anthony Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. Concurrency and Computation: Practice and Experience, 2002.

[28] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* The MIT Press, 1997.

[29] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.