

Superscalar GEMM-based Level 3 BLAS – The On-going Evolution of a Portable and High-Performance Library

Fred Gustavson¹, André Henriksson², Isak Jonsson², Bo Kågström², and
Per Ling²

¹ IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598,
U.S.A.

`gustav@watson.ibm.com`

² Department of Computing Science and HPC2N, Umeå University, S-901 87 Umeå,
Sweden.

`{andropov,isak,bokg,pol}@cs.umu.se`

Abstract. Recently, a first version of our GEMM-based level 3 BLAS for superscalar type processors was announced. A new feature is the inclusion of DGEMM itself. This DGEMM routine contains inline what we call a level 3 kernel routine, which is based on register blocking. Additionally, it features level 1 cache blocking and data copying of sub-matrix operands for the level 3 kernel. Our other BLAS's which possess triangular operands, e.g., DTRSM, DSYRK use a similar level 3 kernel routine to handle the triangular blocks that appear on the diagonal of the larger input triangular operand. Like our previous GEMM-based work all other BLAS's perform the dominating part of the computations in calls to DGEMM. We are seeing the adoption of our BLAS's by several organizations, including the ATLAS and PHiPAC projects on automatic generation of fast DGEMM kernels for superscalar processors, and some computer vendors. The evolution of the superscalar GEMM-based level 3 BLAS is presented. Also, we describe new developments which include techniques that make the library applicable to symmetric multiprocessing (SMP) systems.

1 Introduction

The level 3 Basic Linear Algebra Subprograms (BLAS) [4] are a de facto standard for various matrix multiply and triangular system solving computations and are successfully used as building blocks for the development of high-performance dense linear algebra library software. Due to the complex hardware organization of state-of-the-art computer architectures the development of optimal level 3 BLAS code is costly and time consuming. However, as [8, 9] shows it is possible to develop a portable and high-performance level 3 BLAS library mainly relying on a highly optimized GEMM, the routine for the general matrix multiply and add operation, $C \leftarrow \beta C + \alpha AB$, where C , A and B are matrices of size $m \times n$, $m \times k$ and $k \times n$, respectively, and α, β are scalars. With suitable partitioning, all the

other level 3 BLAS can be defined in terms of GEMM and a small amount of level 1 and level 2 computations [8]. Whenever new high-performance architectures or extensions and modifications of existing ones are introduced, only a few underlying routines need to be optimized for the target architecture, assuming no radical changes in the architecture and memory system. Most important is the routine that implement the level 3 GEMM operation.

Before we go into any further details we outline the rest of the paper. Section 2 gives a short overview of the general GEMM-based level 3 BLAS library. In Section 3 we present the further development of the GEMM-based library targeted to superscalar processors and describe the technique of register blocking used in our kernel routines. Section 4 presents some of our design principles and techniques used for reducing delays in a multi-level memory system. Performance results from our new developments, including parallelization using threads, are also presented.

2 GEMM-based level 3 BLAS

Kågström, Ling and Van Loan developed the model implementations in Fortran 77 of the GEMM-based level 3 BLAS [9, 10]. They also developed a performance evaluation benchmark, which is a tool for evaluating and comparing different implementations of the level 3 BLAS with the GEMM-based model implementations. This software can handle all four data types and is designed to be easy to install and use on different platforms. Each of the GEMM-based routines has a few system-dependent parameters that specify internal block sizes, cache characteristics, and branch points for alternative code sections, which are given as input to a program that facilitates the tuning of these parameters. For convenience, sample values for some common architectures are also provided.

The GEMM-based model implementations are designed to reach high performance through efficient cache reuse. The level 1 and level 2 BLAS used, are not in general associated with cache reuse. However, through appropriate blocking and use of local arrays, cache reuse is achieved during multiple calls to the level 1 and level 2 BLAS routines, and thereby their use approach level 3 performance. This idea works especially well on vector machines, provided that the underlying BLAS are highly optimized. The GEMM-based approach is also adopted in [5].

3 Superscalar GEMM-based level 3 BLAS

To approach peak performance on state-of-the-art superscalar microprocessors it is necessary to attain extensive register reuse. In general, multiple calls to the level 1 and level 2 BLAS routines prohibit an efficient register reuse.

Recently, Kågström and Ling announced the first version of the superscalar GEMM-based level 3 BLAS. They have also developed a superscalar DGEMM that currently is used with the library. The superscalar library has essentially the same overall structure, with similar blocking, as the regular GEMM-based level 3 BLAS. The main difference in the design is that all calls to underlying

level 1 and level 2 BLAS have been removed. As before, the dominating part of all floating point operations take place in calls to DGEMM. The remaining computations that take care of triangular diagonal blocks are handled by “in-line” code optimized for efficient register reuse.

3.1 Unrolling for register reuse

One common technique for register reuse is loop unrolling [2]. We illustrate how loop unrolling is used for a GEMM operation, $C \leftarrow \beta C + \alpha AB$, operating on submatrices. For clarity we assume $\alpha = \beta = 1$.

Assume that we have a processor with at least 24 floating point registers for numbers and that the instruction set includes an instruction for floating point multiply and add (MAA), $z \leftarrow z + xy$, on register operands. Let LOAD and STORE denote the instructions for loading and storing data between memory and registers. The MAA instructions can be arranged so that a single MAA is completed every machine cycle after initial startup.

We use the method illustrated in Figure 1 for a blocked matrix multiplication. The two outermost loops (i - and j -) are unrolled four levels. Within the inner-

```

for  $j = 1$  to  $n$  step 4
  for  $i = 1$  to  $m$  step 4
     $r1 = c_{i,j}$ 
     $r2 = c_{i+1,j}$ 
    ...
     $r16 = c_{i+3,j+3}$ 
    for  $l = 1$  to  $k$ 
       $r1 = r2 + a_{i,l}b_{l,j}$ 
       $r2 = r2 + a_{i+1,l}b_{l,j}$ 
      ...
       $r16 = r16 + a_{i+3,l}b_{l,j+3}$ 
    end
     $c_{i,j} = r1$ 
     $c_{i+1,j} = r2$ 
    ...
     $c_{i+3,j+3} = r16$ 
  end
end

```

Fig. 1. Blocked matrix multiply and add operation using unrolling in two dimensions.

most l -loop, a 4×4 block of C , temporarily stored in the variables $r1, \dots, r16$, is performing 16 MAAs which is the result of a matrix multiplication of blocks of A ($4 \times k$) and B ($k \times 4$) (see Figure 2). The use of the local scalar variables $r1, \dots, r16$, for elements of C allow compilers to keep C in registers (and not load/store to cache/memory) inside the l -loop. Thus, in each iteration of the

$$\begin{bmatrix} r1 & r5 & r9 & r13 \\ r2 & r6 & r10 & r14 \\ r3 & r7 & r11 & r15 \\ r4 & r8 & r12 & r16 \end{bmatrix} \leftarrow \begin{bmatrix} r1 & r5 & r9 & r13 \\ r2 & r6 & r10 & r14 \\ r3 & r7 & r11 & r15 \\ r4 & r8 & r12 & r16 \end{bmatrix} + \left[\dots \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \dots \right] \begin{bmatrix} \vdots \\ [b_{.1} \ b_{.2} \ b_{.3} \ b_{.4}] \\ \vdots \end{bmatrix}$$

Fig. 2. Sequence of 16 MAAs performed within the innermost l -loop.

l -loop, 16 MAA instructions and 8 LOAD instructions are performed. The four elements of A and B in Figure 2 are loaded into registers. Most of the time spent on LOADs is overlapped by the time consumed by 16 MAAs as they execute concurrently in separate functional units. A total of 24 floating point registers are used in the innermost loop.

This technique, to keep a small square block of C in registers and replace entries of A and B between consecutive iterations of the innermost loop, maximizes the ratio between the number of MAAs and the number of load and store instructions, used to transfer data to and from registers, i.e., $\# \text{ MAAs} / (\# \text{ LOADs} + \# \text{ STOREs})$ is maximized. Notice, computing dot products at outer loop levels completely avoids the need for store instructions in the innermost loop. All of the MAAs in the innermost loop update different registers which means they execute in parallel and hence are safely pipelined. They can also be ordered arbitrarily to assist compilers in invoking multiple element load instructions, if available.

Generally, for efficient register reuse it is necessary to unroll in more than one dimension (use more than one loop) and hence matrix-matrix operations must be performed by these kernel routines. We have omitted the fix-up code required, when the dimensions m and n are not multiples of four.

3.2 Improved performance for the superscalar library

In the current release of the superscalar GEMM-based level 3 BLAS, 4×4 unrolling is used for the C matrix in DGEMM and 4×2 unrolling is used in the remaining routines. As for the GEMM-based model implementations all references are stride one which is implemented using work arrays and data copying prearranged so that the DGEMM kernel will run close to peak performance. The extra data copying allows the superscalar library to handle so called “critical” leading dimensions as well [9, 10]. The Fortran source code is publically available from netlib, see ‘www.netlib.org/blas/gemm_based/ssgemmbased.tgz’.

Performance results from the GEMM-based level 3 BLAS performance benchmark on an IBM PowerPC 604 processor (112 MHz, IBM SP, SMP node) show substantial improvements for the current release of the superscalar library:

DSYMM	DSYRK	DSYR2K	DTRMM	DTRSM
+3%	+28%	+2%	+23%	+25%

These percentage numbers are for square matrices of size 500×500 . We obtain up to 80% improvement for small matrices (32×32). The improvements are mainly

for the routines that called level 2 routines in the model implementations [9, 10]. The GEMM-based algorithms for DSYMM and DSUR2K do not call any level 2 routines. The calculations are transformed to level 3 GEMM operations by copying the symmetric subblocks stored in triangular format to general full format subblocks in work arrays [11].

The ATLAS [12] and PHiPAC projects [3] use the superscalar GEMM-based level 3 BLAS together with their own automatically tuned DGEMM to provide a complete set of level 3 BLAS in double precision. The ATLAS project reports impressive performance results for several different machines where the combination of the superscalar GEMM-based level 3 BLAS and ATLAS DGEMM is often faster than the vendor supplied level 3 BLAS, see 'www.netlib.org/atlas'.

4 Techniques for More Complex Memory Hierarchies

In our on-going development of the superscalar GEMM-based level 3 BLAS we are investigating different techniques to reduce delays in cache and memory accesses. We focus on ways to handle memory hierarchies with multiple levels of cache memories efficiently. Furthermore, we are using threads for developing parallel versions of our routines. In the following we survey some of these techniques.

In this case study, all experiments are carried out on an SMP node of the IBM SP parallel system. Presumably, this SMP node has several characteristics which will be typical in future high performance computing systems. For example, several processors (currently four IBM PowerPC 604 with an MAA instruction), a memory hierarchy with more than one level of cache (currently L1 and L2 cache), and a single shared main memory. We sometimes use assembly language in these experiments since some of the techniques we investigate cannot be invoked via current Fortran or C compilers. One goal with this case study is to find strong evidence for introducing support for these techniques in future compilers.

4.1 Prefetching

Generally, when data does not reside in the top levels of a memory hierarchy (registers or L1 cache), significant delays occur when the data is referenced. A well known technique to overcome these delays, commonly used in compilers, is prefetching. This technique enables data to be brought to registers or cache, ahead of its use, so that when the data is needed, it is immediately available. Unfortunately, today's compilers do not perform prefetching as well as one would desire, especially for complex memory hierarchies. A software technique to include prefetching directly into Fortran programs called *algorithmic prefetching* is suggested by Agarwal, Gustavson and Zubair [1]. They used this technique successfully for the level 2 BLAS of the IBM ESSL library. However, the usefulness of this technique depends much on the optimization procedures performed by compilers. In our assembly programs we use a "touch" instruction, 'dcbt' to

perform prefetching. It is currently not possible to invoke this instruction explicitly from Fortran or C. This instruction brings data up into L1 cache, essentially without disturbing the ongoing computations and data references.

4.2 Algorithmic preloading

We also use a technique that we call *algorithmic preloading* for prefetching data. We distinguish this technique from algorithmic prefetching, since the latter is characterized by statements or instructions in a program that can be removed without affecting the correctness of the program, while statements or instructions for algorithmic preloading cannot.

In Figure 3 we have modified the algorithm of Figure 1 to illustrate algorithmic preloading. Apart from loading entries of C into registers, via $r1, \dots, r16$, we also load four elements of A and four elements of B into registers, via $r17, \dots, r24$, before the l -loop starts. These elements are operands to the MAAs in the first iteration of the l -loop. Immediately after the last reading of each of $r17, \dots, r24$ within the l -loop, they are preloaded with the entries from A and B that take part in the next iteration. This way, the operands of the MAAs are loaded into registers as early as possible, and thereby more likely to be available when the MAAs are ready to execute. Possibly all of the MAAs can execute without any delays, enabling for the l -loop to approach peak performance. The MAAs of the k -th iteration execute after the l -loop is completed.

4.3 Hierarchical blocking

The purpose of the blocking structure of the model implementations and the current version of the superscalar GEMM-based level 3 BLAS is to reuse the most frequently referenced matrix blocks in L1 cache as much as possible. However, for machines with several levels of cache memories a need for more advanced blocking strategies arises.

Explicit multi-level blocking: One way to handle the memory hierarchy explicitly is to reorganize loops such that each loop set matches a specific level of the memory hierarchy. However, this is a tedious work and requires a lot of knowledge about the architecture characteristics. For example, a blocking parameter is required for each level of the memory hierarchy. Some results of a two-level blocked matrix multiply for the PowerPC 604 are presented in Section 4.4. The two blocking parameters are tuned for L1 and L2 cache.

Automatic blocking via recursion: Recursion is a key concept for matching an algorithm and its data structure. In [6] we introduced recursive blocked data formats and recursive blocked algorithms for level 3 BLAS. The recursive algorithms lead to automatic blocking which is variable and “squarish”. The blocking is variable and hierarchical and allow for maintaining the two-dimensional data locality at every level of the one-dimensional tiered memory structure. The only tuning parameter is the size of L1 cache. The size of each matrix block is constrained so that a few of them will simultaneously fit in L1 cache. For more details see [6].

```

for j = 1 to n step 4
  for i = 1 to m step 4
    Preload  $C_{i:i+3,j:j+3}$  to  $r_1, \dots, r_{16}$ 
    Preload  $A_{i:i+3,1}$  to  $r_{17}, \dots, r_{20}$ 
    Preload  $B_{1,j:j+3}$  to  $r_{21}, \dots, r_{24}$ 
    for l = 1 to k - 1
       $r_1 = r_1 + r_{17}r_{21}$ 
      ...
      Preload:  $r_{17} = a_{i,l+1}$ 
      ...
      Preload:  $r_{21} = b_{l+1,j}$ 
      ...
       $r_{16} = r_{16} + r_{20}r_{24}$ 
      Preload:  $r_{20} = a_{i+3,l+1}$ 
      Preload:  $r_{24} = b_{l+1,j+3}$ 
    end
     $r_1 = r_1 + r_{17}r_{21}$ 
     $c_{i,j} = r_1$ 
    ...
     $r_{16} = r_{16} + r_{20}r_{24}$ 
     $c_{i+3,j+3} = r_{16}$ 
  end
end
end

```

Fig. 3. Blocked matrix multiply and add operation using unrolling in two dimensions and preloading of matrix entries one iteration ahead of their use.

Our assembly implementation consists of a recursively called routine that creates a list of calls to a computational kernel and then executes these calls. The list constitutes a precalculated *recursive task tree* with actual calculations at the leaf nodes. The computational kernel is designed to hold six matrix blocks in L1 cache simultaneously. Three of them are involved in the current computation while the other three blocks are being prefetched using the ‘dcbt’ instruction. When the current block computation is completed, a new computation starts involving the blocks just prefetched, and three new blocks are prefetched for the subsequent computations, and so on.

4.4 Some uniprocessor performance results

In Figure 4 we show the performance of an explicitly tuned two-level blocked algorithm and the recursive GEMM algorithm [6] executing on an IBM PowerPC 604. Both routines use the same atomic kernel that prefetches, register preloads, does 4×4 unrolling, and works on 16×16 submatrices. The performance results, which are very good, do not account for any data copying, i.e., the blocks are initially stored in column block order and recursive block order, respectively. The recursive algorithm performs around 7% better than the multi-level blocked

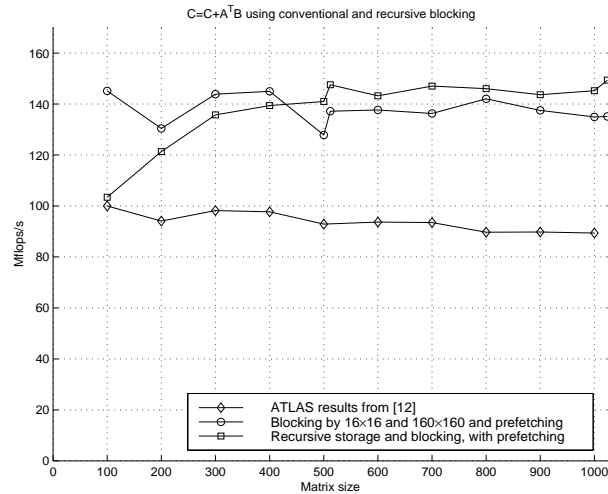


Fig. 4. Performance of two hierarchical blocking strategies.

algorithm for large enough problems (500×500), even though the tuning is much less explicit (only L1 cache versus L1 and L2 cache tuning for the multi-blocked algorithm).

4.5 Parallelization

The recursive task tree facilitates parallelization on shared memory machines. We simply divide the tree into subtrees and let different processes or threads (lightweight processes) execute on different subtrees. We use a single thread for each processor and divide the tree into one subtree more than the number of threads. The left-over subtree is divided among the threads when they become ready for additional work. This way of scheduling enables good load balancing on non-dedicated machines.

For multiple threads, we show performance results with the recursive task tree approach in Figure 5 [7]. The algorithm scales well. One reason is that the recursive task tree automatically keeps data references local to each thread.

This research was conducted using the resources of the High Performance Computing Center North (HPC2N).

References

1. R. C. Agarwal, F. G. Gustavson, and M. Zubair. Improving performance of linear algebra algorithms for dense matrices, using algorithmic prefetch. *IBM J. Res. Develop*, 38(3):265–275, May 1994.
2. R. C. Agarwal, F. G. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM J. Res. Develop*, 38(5):563–576, September 1994.

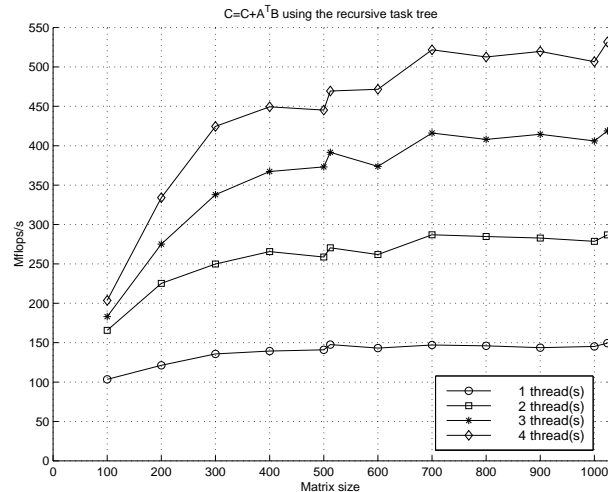


Fig. 5. Scheduling the matrix multiply problem on several threads.

3. J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high performance, ANSI C coding methodology. In *Proceedings of the 11th International Conference on Supercomputing (ICS-97)*, pages 340–347, New York, July 7–11 1997. ACM Press.
4. J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 18–28, March 1990.
5. M. J. Dayde, I. S. Duff, and A. Petitot. A parallel block implementation of level-3 BLAS for MIMD vector processors. *ACM Trans. Math. Softw.*, 20(2):178–193, June 1994.
6. F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström and P. Ling. Recursive Blocked Data Formats and BLAS's for Dense Linear Algebra Algorithms. *This Proceedings*, Springer Verlag, 1998.
7. A. Henriksson and I. Jonsson. High-Performance Matrix Multiplication on the IBM SP High Node. *Master Thesis*, UMNAD 98.235, Department of Computing Science, Umeå University, S-901 87 Umeå, June 1998.
8. B. Kågström and C. Van Loan. GEMM-Based Level-3 BLAS. Technical Report CTC91TR47, Department of Computer Science, Cornell University, Dec. 1989.
9. B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Software*, 1997. To appear.
10. B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: Portability and optimization issues. *ACM Trans. Math. Software*, 1997. To appear.
11. P. Ling. A set of high-performance level 3 BLAS structured and tuned for the IBM 3090 VF and implemented in Fortran 77. *The Journal of Supercomputing*, 7(3):323–355, September 1993.
12. R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. Tech. Report TN 37996-1301, Computer Science Dept., Univ. of Tennessee, 1997.