Using Recursion to Boost ATLAS's Performance

Paolo D'Alberto[1] and Alexandru Nicolau[2]

[1] Department of Electrical and Computer Engineering - Carnegie Mellon University
[2] Department of Computer Science - University of California at Irvine *

Abstract. We investigate the performance benefits of a novel recursive formulation of Strassen's algorithm over highly tuned matrix-multiply (MM) routines, such as the widely used ATLAS for high-performance systems.
We combine Strassen's recursion with high-tuned version of ATLAS MM and we present a family of recursive algorithms achieving up to 15% speed-up over ATLAS alone. We show experimental results for 7 different systems.

Keywords: dense kernels, matrix-matrix product, performance optimizations.

1 Introduction

In this paper, we turn our attention to a single but fundamental basic kernel in dense and parallel linear algebra such as **matrix multiply** (MM) for matrices stored in double precision.

In practice, software packages such as LAPACK [1] or ScaLAPACK are based on a basic set of routines such as the basic linear algebra subroutines BLAS [2,3]. Moreover, The BLAS is based on an efficient implementations of the MM kernel.

In the literature, we find an abundant collection of algorithms, implementations and software packages (e.g., [4,5,6,7,8,9,10,11]), that aim at the efficient solution of this basic kernel. However, among all ATLAS [11] is one of the most widely recognized and used.

In today's high performance computing, the system performance is the result of a fine and complicated relation between the constituent parts of a processor –i.e., the hardware component, and the

* This work has been supported in part by NSF Contract Number ACI 0204028. Email the authors *pdalbert@andrew.cmu.edu* and *nicolau@ics.uci.edu*.

sequence of instructions of an application –i.e., the software component. For example, ATLAS [11] is an adaptive software package implementing BLAS that addresses the system-performance problem by careful adaptation of the software component. In practice, ATLAS generates an optimized version of MM tailored to the specific characteristics of the architecture and ATLAS does this custom installation by an combination of micro-benchmarking and an empirical search of the code solution space. In this work, we show how an implementation of Strassen's algorithm can further improve the performance of even highly-tuned MM such as ATLAS.

In the literature, other approaches have been proposed to improve the classic formulation of MM by using Strassen's strategy [12] (or Winograd's variant). In fact, Strassen's algorithm has noticeably fewer operations $O(n^{\log_2 7}) = O(n^{2.86})$ than the classic MM algorithm $O(n^3)$ and, thus, potential performance benefits. However, the execution time of data accesses dominates the MM performance and this is due to the increasing complexity of the memory hierarchy realty.

In fact, experimentally, Strassen's algorithm has found validation by several authors [13,14,5] for *simple* architectures, showing the advantages of this new algorithm starting from very small matrices or **recursion truncation point** (RP) [15]. The recursion point is the matrix size n_1 for which Strassen's algorithm yields to the original MM. Thus, for a problem of size $n = n_1$, Strassen's algorithm has the same performance of the original algorithm, and, for every matrix size $n \geq n_1$, Strassen's algorithm is faster than the original algorithm. With the evolution of the architectures and the increase of the problem sizes, the researcher community witnessed the RP increasing [16]. We now find projects and libraries implementing different version of Strassen's algorithm and considering its practical benefits [15,17,18], however with larger and larger RP, mining the practical use of Strassen's algorithm.

In this paper, we investigate recursive algorithms for an empirical RP determination and we embody our ideas so as to combine the high performance of tuned dense kernels –at the low level– with Strassen's recursive division process –at the high level– into a family of recursive algorithms. We present our experimental results for 7 systems where we tested our codes.

Our approach has the following advantages over previous approaches. First, we do not pad the original matrices so as to have even-size or, worse, power-of-two matrices [12]. Second, our codes have no requirements on the matrix layout, thus, they can be used instead of other MM routines (ATLAS) with no modifications or extra overhead to change the data layout before and after the basic computation (unlike the method proposed in [18]). In fact, we assume that the matrices are stored in row-major format and, at any time, we can yield control to a highly tuned MM such as ATLAS's $dgemm()$. Third, we propose a balanced recursive division into subproblems, thus, the codes exploit predictable performance; unlike the division process proposed by Huss-Lederma et al. [15] where for odd-matrix sizes, they divide the problem into a large even-size problem, on which Strassen can be applied, and a small, and extremely irregular, computation. Fourth, we investigate recursive algorithms that can unfold the division process more than once so to achieve further performance (in contrast to [15,18] where the unfolding is limited to one level).

The paper is organized as follows. In Section 2, we present a generalization of Strassen's algorithm better suited for recursion. In section 3, we present our techniques to determine the RP for our codes. In Section 4, we present our experimental results. Finally, Section 5, we present our concluding remarks.

2 Strassen's Algorithm for any Square-Matrix Sizes

In this section, we show that Strassen's MM algorithm can be generalized quite naturally and more efficiently than previous implementations available in the literature [12,18,15] so that it can be applied to any square-matrix size.

From here on, we identify the **size** of a matrix $\mathbf{A} \in \mathbb{M}^{m \times n}$ as $\sigma(\mathbf{A}) = m \times n$. We assume that an operand matrix $\mathbf{A}$ of size $\sigma(\mathbf{A}) = n \times n$ is logically composed by four **near square** matrices; that is, every submatrix has number of rows r and number of columns c that differ by at most one, i.e., $|r - c| \leq 1$, [9].

The classical MM of $\mathbf{C} = \mathbf{AB}$ can be expressed as the multiplication of the submatrices as follows: $\mathbf{C}_0 = \mathbf{A}_0\mathbf{B}_0 + \mathbf{A}_1\mathbf{B}_2$, $\mathbf{C}_1 =$

$\mathbf{A}_0\mathbf{B}_1 + \mathbf{A}_1\mathbf{B}_3$, $\mathbf{C}_2 = \mathbf{A}_2\mathbf{B}_0 + \mathbf{A}_3\mathbf{B}_2$ and $\mathbf{C}_3 = \mathbf{A}_2\mathbf{B}_1 + \mathbf{A}_3\mathbf{B}_3$. The computation is divided in four basic computations, one for each submatrix composing $\mathbf{C}$. Thus, for every matrix $\mathbf{C}_i$ $(0 \leq i \leq 3)$, the classical approach computes two products, for a total of 8 MMs and 4 **matrix additions** (MA).

Notice that every product is the MM of near square matrices and it computes a result that has the same size and shape of the submatrix destination $\mathbf{C}_i$. Furthermore, if we compute the products recursively, each product is divided in further four subproblems on near square matrices [9].

Strassen proposed to divide the problem into only 7 MMs and to introduce 18 matrix additions/subtractions. When the matrices have power-of-two sizes, $n = 2^k$, all multiplications and additions are among square matrices of the same sizes even if the computation is recursively carried on. We adapt Strassen's algorithm so as to compute the MM for every square matrix size as follows: $\mathbf{C}_0 = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$, $\mathbf{C}_1 = \mathbf{M}_2 + \mathbf{M}_4$, $\mathbf{C}_2 = \mathbf{M}_3 + \mathbf{M}_5$ and $\mathbf{C}_3 = \mathbf{M}_1 + \mathbf{M}_3 - \mathbf{M}_2 + \mathbf{M}_6$ where every $\mathbf{M}_i$ is defined as follow:

$\mathbf{M}_1 = \mathbf{T}_0\mathbf{T}_1$ with $\mathbf{T}_0 = \mathbf{A}_0 + \mathbf{A}_3$ of size $\sigma(\mathbf{T}_0) = \lceil n \rceil \times \lceil n \rceil$,
and with $\mathbf{T}_1 = \mathbf{B}_0 + \mathbf{B}_3$ of size $\sigma(\mathbf{T}_1) = \lceil n \rceil \times \lceil n \rceil$
thus $\sigma(\mathbf{M}_1) = \lceil n \rceil \times \lceil n \rceil$

$\mathbf{M}_2 = \mathbf{T}_2\mathbf{B}_0$ with $\mathbf{T}_2 = \mathbf{A}_2 + \mathbf{A}_3$ of size $\sigma(\mathbf{T}_2) = \sigma(\mathbf{M}_2) = \lfloor n \rfloor \times \lceil n \rceil$

$\mathbf{M}_3 = \mathbf{A}_0\mathbf{T}_3$ with $\mathbf{T}_3 = \mathbf{B}_1 + \mathbf{B}_3$ of size $\sigma(\mathbf{T}_3) = \sigma(\mathbf{M}_3) = \lceil n \rceil \times \lfloor n \rfloor$

$\mathbf{M}_4 = \mathbf{A}_3\mathbf{T}_4$ with $\mathbf{T}_4 = \mathbf{B}_2 - \mathbf{B}_0$ of size $\sigma(\mathbf{T}_4) = \sigma(\mathbf{M}_4) = \lfloor n \rfloor \times \lceil n \rceil$

$\mathbf{M}_5 = \mathbf{T}_5\mathbf{B}_3$ with $\mathbf{T}_5 = \mathbf{A}_0 + \mathbf{A}_1$ of size $\sigma(\mathbf{M}_5) = \sigma(\mathbf{T}_5) = \lceil n \rceil \times \lfloor n \rfloor$

$\mathbf{M}_6 = \mathbf{T}_6\mathbf{T}_7$ with $\mathbf{T}_6 = \mathbf{A}_2 - \mathbf{A}_0$ and $\mathbf{T}_7 = \mathbf{B}_0 + \mathbf{B}_1$
of size $\sigma(\mathbf{M}_6) = \sigma(\mathbf{T}_6) = \sigma(\mathbf{T}_7) = \lceil n \rceil \times \lceil n \rceil$

$\mathbf{M}_7 = \mathbf{T}_8\mathbf{T}_9$ with $\mathbf{T}_8 = \mathbf{A}_1 - \mathbf{A}_3$ of size $\sigma(\mathbf{T}_8) = \lceil n \rceil \times \lfloor n \rfloor$
and $\mathbf{T}_9 = \mathbf{B}_2 + \mathbf{B}_3$ of size $\sigma(\mathbf{T}_8) = \lfloor n \rfloor \times \lceil n \rceil$
and $\sigma(\mathbf{M}_7) = \lceil n \rceil \times \lceil n \rceil$

As result of the division process, the matrices $\mathbf{A}_i$, $\mathbf{B}_i$ and $\mathbf{C}_i$ are near square matrices as in the classic algorithm but MA and MMs must be re-defined.

First, we generalize **matrix addition**. Intuitively, when the resulting matrix $\mathbf{X}$ is larger than $\mathbf{Y}$ or $\mathbf{Z}$, the computation is performed

as if the matrix operands are extended and padded with zeros. Otherwise, if the result matrix is smaller than the operands, the computation is performed as the matrix operands are cropped to fit the result matrix. See a simple implementation for the addition of two generic matrices in Figure 1.

```
/* C = A+B */
void Add(Mtype *c, int McolC, int mC, int pC,
         Mtype *a, int McolA, int mA, int pA,
         Mtype *b, int McolB, int mB, int pB) {

   int i,j,x,y;

   /* minimum sizes */
   x = min(mA,mB); y = min(pA,pB);

   for (i=0; i<x; i++) {
     /* core of the computation */
     for (j=0;j<y;j++)  c[i*McolC+j] = a[i*McolA+j] + b[i*McolB+j];

     if (y<pA)      c[i*McolC+y] = a[i*McolA+y]; /* A is larger than B */
     else if (y<pB) c[i*McolC+y] = b[i*McolB+y]; /* B is larger than A */
   }

   /* last row */
   if (x<mA) { /* A is taller than B */
     for (j=0;j<y;j++) c[x*McolC+j] = a[x*McolA+j];

     if (y<pA)       c[x*McolC+y] = a[x*McolA+y];
     else if (y<pB)  c[x*McolC+y] = b[x*McolB+y];
   }
   else if (x<mB) { /* B is taller than A */
     for (j=0;j<y;j++) c[x*McolC+j] = b[x*McolB+j];

     if (y<pA)      c[x*McolC+y] = a[x*McolA+y];
     else if (y<pB) c[x*McolC+y] = b[x*McolB+y];
   }
}
```

Fig. 1. Addition C-code

Second, we generalize **matrix multiplication** as follows: $\mathbf{X} = \mathbf{Y} * \mathbf{Z}$ where $\sigma(\mathbf{X}) = m \times n$, $\sigma(\mathbf{Y}) = m \times q$ and $\sigma(\mathbf{Z}) = r \times n$ so as $c_{i,j} = \sum_{k=0}^{\min(q,r)} y(i,k) * z(k,j)$.

Notice that the product $\mathbf{A}_0\mathbf{B}_0$, which is a term of $\mathbf{M}_1$, is a **necessary product** and it is required for the computation of $\mathbf{C}_0$; in contrast, $\mathbf{A}_0\mathbf{B}_3$ is an **artificial product**, computed in the same expression, and it must be reduced by MAs (e.g., $\mathbf{M}_1 + \mathbf{M}_4$). The algorithm previously defined computes correctly all necessary products and it annihilates all artificial products.

Both MA and MM, as previously defined, introduce negligible overheads. In fact, the matrices involved in the computations are always near square matrices (i.e., their sizes may differ by at most

one) and, thus, the extra control is negligible for the matrix sizes tested in this work. [3] We explain how the two approaches, that is, our version of Strassen's and tuned ATLAS routines are combined in Section 3.

In our codes, the matrix are stored in row-major format and we do not apply any recursive layout strategy as in [18], for the following three reasons. First, modern memory hierarchies use (4+ way) associative caches for which the effects of cache interferences, due to the matrix layout, is relatively minimal. Second, the MAs in the Strassen's algorithm create a smaller working space where the operands are stored dynamically, so the effect of interference can be reduced further. Third and last, non-standard layout complicates the development of correct and efficient leaf-computation routines for any square matrices; in fact, these leaf routines must be tailored to the type of layout.

The simplicity of our code in conjunction with the performance improvements achievable make our approach a good strategy addition to the already widely used software packages such as ATLAS, especially for large problems. Our pseudo code is presented in Figure 2. We also reorganized the original Strassen's computation so as to use only three temporary matrices, as already proposed in the literature [15].

3 Empirical Considerations on the Recursion Truncation Point

In this section, we propose a technique for determining when the algorithm's strategy must change so as to stop Strassen's and to yield control to the regular MM, the recursion truncation point (RP). In other words, we consider the problem of when to have a recursive call (to Strassen's MM) or a call to an highly tuned *dgemm* (e.g., such as the one offered by ATLAS). We show in Section 4 that the optimal strategy is a function of the problem size and of the underlying system.

[3] Furthermore, we use the highly tuned ATLAS *dgemm*() to reduce further the effects on the overall performance.

```
/*
 * | C0 C1 |   | A0 A1 |   | B0 B1 |
 * | C2 C3 | = | A2 A3 | * | B2 B3 |
 */
C mul(A, B) {

  if (Problem_Size < leaf_strassen)
    CC = AA atlas_dgemm  BB;
  else {
    Allocate_workspace(T1,T2,M1);

    T1 = A0 add A3;    T2 = B0 add B3;
    M1 = T1 mul T2;
    C0 = M1;           C3 = M1;

    T1 = A2 add A3;
    M2 = T1 mul B0;
    C2 = M2;           C3 = C3 sub M2;

    T1 = B1 sub B3;
    M3 = A0 mul T1;
    C1 = M3;           C3 = C3 add M3;

    T1 = B2 sub B0;
    M4 = A3 mul T1;
    C0 = C0 add M4;    C2 = C2 add M4;

    T1 = A0 add A1;
    M5 = T1 mul B3;
    C0 = C0 sub M5;    C1 = C1 add M5;

    T1 = A2 sub A0;    T2 = B0 add B1;
    M6 = T1 mul T2;
    C3 = C3 add M6;

    T1 = A1 sub A3;    T2 = B2 add B3;
    M7 = T1 mul T2;
    C0 = C0 add M7;

    Deallocate_workspace();
  }
}
```

Fig. 2. Pseudo Strassen's Algorithm

Strassen's algorithm embodies different locality properties because its two basic computations exploit different data locality: MM has spatial and temporal locality, and MA has only spatial locality. In fact, consider that the matrix operands fit a cache level, for example L_2, but do not fit the lower cache, such as L_1. Note that the MA does not exploit data locality at the lower levels of cache and, actually, data accesses to/from the CPU during the MA will flush previous contents. In fact, MAs have little data reuse and, thus, data-access latency time cannot be circumvented or hidden; for these applications a memory hierarchy actually slows down the overall performance. In contrast, highly tuned MMs exploit temporal and spatial locality at every level of cache, thus, having fast memory

accesses and fast computations. In a hierarchical memory system, the two computations may have drastically different performance. Thus, Strassen's algorithm has a performance edge versus the regular MM only when the savings in MMs, is higher (in execution time) than the cost of the extra additions.

In the literature, we find different and, often contradicting, experimental results about the RP. In fact, a few authors have found that for any problem size Strassen's (or Winograd's variation) is always faster; a few authors have found that the RP is about 500 for some systems and implementations; and a few others, citing private communications, claim that the RP is larger than 1000 [14,16,5,15,18].

Even though the RP is machine and problem-size dependent, however it is straightforward to determine, even if tedious and time consuming. We propose to determine the RP empirically by direct measure of Strassen's MM execution and we do this for recursive Strassen's algorithm with different unfolding levels. This idea is very similar to the one applied for the solution search in ATLAS.

4 Experimental Results

We installed our codes and the software package ATLAS on 7 different architectures, Table 1. Once the installation is finished, we then determined experimentally the RP n_1 based on a simple linear search. Note that for the Fosa system, we could find no problem size

Table 1. Systems

System	**Processors**	$\mathbf{n_1}$	**Figure**
Fujitsu HAL 300	SPARC64 100MHz	400	Fig. 3
Ultra 5	UltraSparc2 300MHz	1225	Fig. 4
Ultra-250	UltraSparc2 2 @ 300MHz	1300	No
Sun-Fire-V210	UltrasparcIII 1GHz	1150	Fig. 5
ASUS	AthlonXP 2800+ 2GHz	1300	Fig. 6
Unknown server	Itanium 2 @ 700MHz	2150	Fig. 7
Fosa	Pentium III 800MHz	N/A	No

for which Strassen's is faster than ATLAS's.

In the following, we present the experimental results for five systems. We use the following terminology: **S-k-unfold** is the Strassen

algorithm for which k is the number of times the recursion unfolds before yielding to ATLAS *dgemm*. (Note we opted to omit negative relative performance and no bar is presented in the charts instead.) The performance obtained by the systems in Table 1, and presented from Figure 3 to Figure 7, are obtained by the collection of the best performance among several trials.

Note that the S-2-unfold algorithm is beneficial for very large problems and for specific systems. However, for the systems in Table 1, the performance improvements are some how limited. We have performance measures of the S-3-unfold algorithm but for the current set of systems, the algorithm has no performance advantage over ATLAS and, thus, we do not report them.

From Figure 3 to Figure 7, we present two measures of performance: relative execution time over ATLAS, and relative MFLOPS for ATLAS *dgemm* over peak performance. In fact, the execution time is what any final user cares comparing two different algorithms. However a measure of performance for ATLAS shows whether or not Strassen's algorithms improve the performance of a MM kernel which is either efficiently or poorly designed.

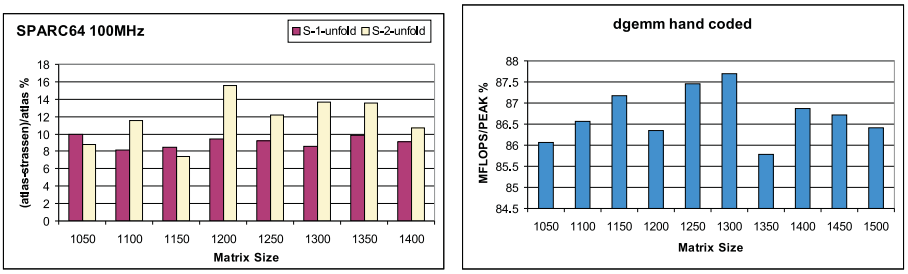

Fig. 3. Fujitsu HAL 300.

5 Conclusions

We have presented a practical implementation of Strassen's algorithm, which applies a recursive algorithm to exploit highly tuned MMs, such as ATLAS's. We differ from previous approaches because

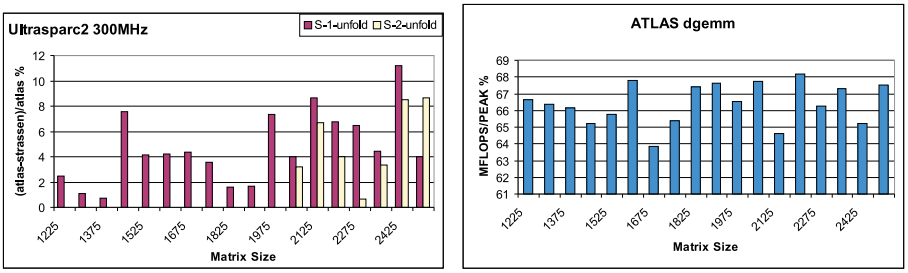

Fig. 4. Ultra 5.

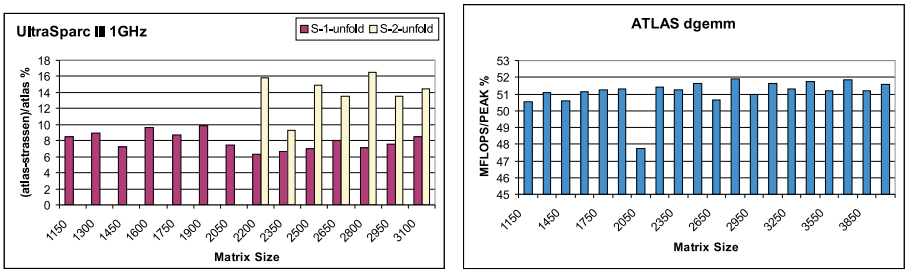

Fig. 5. Sun-Fire-V210.

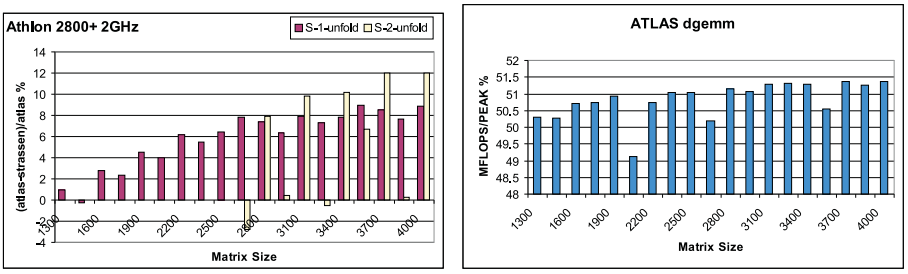

Fig. 6. ASUS A7N8X.

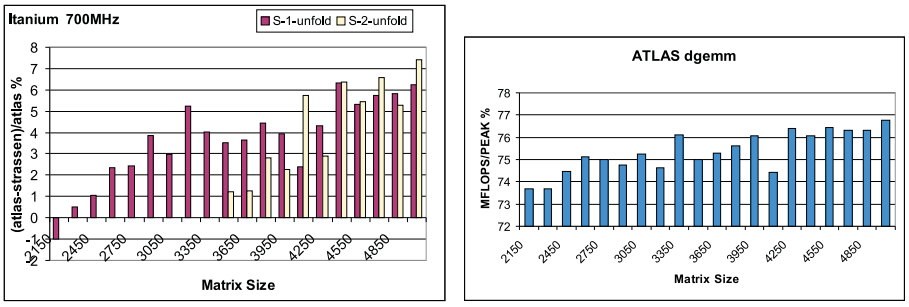

Fig. 7. Linux Itanium 2 700 MHz.

we investigate a family of recursive algorithms with a balanced division process, which, in turn, makes the algorithm performance more predictable.

We have tested the performance of our approach on 7 systems with different level of recursion unfolding, and we have shown that not always Strassen is applicable. We have also shown that for modern systems the RP can be quite different and quite large.

As future work, we will investigate the implementation of a single adaptive recursive algorithm. In fact, the ideas implemented in our codes yield to a natural approach for the automatic determination of the RP for a recursive Strassen's algorithm for different systems.

References

1. Anderson, E., Bai, Z., Bischof, C., Dongarra, J.D.J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., Sorensen, D.: LAPACK User' Guide, Release 2.0. 2 edn. SIAM (1995)
2. Kagstrom, B., Ling, P., van Loan, C.: Algorithm 784: GEMM-based level 3 BLAS: portability and optimization issues. ACM Transactions on Mathematical Software **24** (1998) 303–316
3. Kagstrom, B., Ling, P., van Loan, C.: GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. ACM Transactions on Mathematical Software **24** (1998) 268–302
4. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. In: Proceedings of the 19-th annual ACM conference on Theory of computing. (1987) 1–6
5. Higham, N.J.: Exploiting fast matrix multiplication within the level 3 BLAS. ACM Trans. Math. Softw. **16** (1990) 352–368
6. Frens, J., Wise, D.: Auto-Blocking matrix-multiplication or tracking BLAS3 performance from source code. Proc. 1997 ACM Symp. on Principles and Practice of Parallel Programming **32** (1997) 206–216
7. Eiron, N., Rodeh, M., Steinwarts, I.: Matrix multiplication: a case study of algorithm engineering. In: Proceedings WAE'98, Saarbrücken, Germany (1998)
8. Whaley, R., Dongarra, J.: Automatically tuned linear algebra software. In: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM), IEEE Computer Society (1998) 1–27
9. Bilardi, G., D'Alberto, P., Nicolau, A.: Fractal matrix multiplication: a case study on portability of cache performance. In: Workshop on Algorithm Engineering 2001, Aarhus, Denmark (2001)
10. Goto, K., van de Geijn, R.: On reducing tlb misses in matrix multiplication. Technical Report Technical Report TR-2002-55, The University of Texas at Austin, Department of Computer Sciences (2002)
11. Demmel, J., Dongarra, J., Eijkhout, E., Fuentes, E., Petitet, E., Vuduc, V., Whaley, R., Yelick, K.: Self-Adapting linear algebra algorithms and software. Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" **93** (2005)

12. Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik **14** (1969) 354–356
13. Brent, R.P.: Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity. Numerische Mathematik **16** (1970) 145–156
14. Brent, R.P.: Algorithms for matrix multiplication. Technical Report TR-CS-70-157, Stanford University (1970)
15. Huss-Lederman, S., Jacobson, E., Tsao, A., Turnbull, T., Johnson, J.: Implementation of Strassen's algorithm for matrix multiplication. In: Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM), ACM Press (1996) 32
16. Bailey, D.H., Gerguson, H.R.P.: A Strassen-Newton algorithm for high-speed parallelizable matrix inversion. In: Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing, IEEE Computer Society Press (1988) 419–424
17. Bilmes, J., Asanovic, K., Chin, C., Demmel, J.: Optimizing matrix multiply using PHiPAC: a portable, high-performance, Ansi C coding methodology. In: International Conference on Supercomputing. (1997)
18. Thottethodi, M., Chatterjee, S., Lebeck, A.: Tuning Strassen's matrix multiplication for memory efficiency. In: Proc. Supercomputing, Orlando, FL (1998)