# Adaptive Strassen and ATLAS's DGEMM: A Fast Square-Matrix Multiply for Modern High-Performance Systems

Paolo D'Alberto
Department of Electrical and Computer Engineering
Carnegie Mellon University

Alexandru Nicolau
Department of Computer Science
University of California at Irvine
IEEE Member

***Abstract*— Strassen's algorithm has practical performance benefits for architectures with simple memory hierarchies, because it trades computationally expensive matrix multiplications (MM) with cheaper matrix additions (MA). However, it presents no advantages for high-performance architectures with deep memory hierarchies, because MAs exploit limited data reuse.**

**We present an easy-to-use adaptive algorithm combining Strassen's recursion and high-tuned version of ATLAS MM. In fact, we introduce a last step in the ATLAS-installation process that determines whether Strassen's may achieve any speedup. We present a recursive algorithm achieving up to 30% speed-up versus ATLAS alone. We show experimental results for 14 different systems.**

## I. Introduction

In the last 30 years, the complexity of modern uniprocessor and, thus, multiprocessor systems is following accurately Moore's law. That is, the number of transistors per chip doubles every 18 months. Unfortunately, the steady increase of complexity of a processor does not always translate in a proportional increase of the system performance. In fact, the system-performance equation is the result of a fine and complicated relation between the constituent parts of a processor, the hardware component, and the sequence of instructions of an application, the software component.

This work has been support in part by NSF Contract Number ACI 0204028. Contact the authors: *pdalberto@ece.cmu.edu* and *nicolau@ics.uci.edu*.

In this work, we turn our attention to the software component of the performance equation and, specifically, to **adaptive codes**. In fact, adaptive codes are an effective solution for an efficient utilization of complex and always-changing architectures (e.g., [1], [2], [3]). In this paper, we focus on a single but fundamental kernel in dense and parallel linear algebra such as **matrix multiply** (MM) for matrices stored in double precision [4], [5], [6], [7], [8], [9].

In practice, software packages such as LAPACK [10] (or ScaLAPACK, LINPACK) are based on a basic set of routines such as BLAS [11], [12], which, in turn, is based on an efficient implementations of the MM kernel. ATLAS [2] is a clear example of an adaptive software package implementing BLAS. In this work, we show how an adaptive implementation of Strassen's algorithm can further improve the performance of even highly-tuned MM (e.g., ATLAS).

Strassen's algorithm [13] is among the first examples of algorithm engineering: in fact, Strassen discovered that the original recursive algorithm of complexity $O(n^3)$ can be reorganized so that, at a recursive step, one *computationally expensive* recursive MM can be traded in for 18 *cheaper* matrix additions (MA). As result, Strassen's algorithm has noticeably fewer operations $O(n^{\log_2 7}) = O(n^{2.86})$. (Winograd's variant requires only 15 additions and, thus, it is more efficient than Strassen's algorithm by a constant.)

Experimentally, Strassen's algorithm has found validation by several authors [14], [15], [5], showing the advantages of this new algorithm starting from

very small matrices or **cross-over sizes**.[1] With the evolution of the architectures and the increase of the problem sizes, the cross-over size started increasing [17]. We now find projects and libraries implementing different version of Strassen's algorithm and considering its practical benefits [16], [18], [19], however with larger and larger cross-over sizes and, thus, undermining the practical use of Strassen's algorithm.

None of the approaches previously proposed have really attempted to determine if there can be an automatic technique to determine whether or not such a change of strategy is effective for a particular architecture. In this paper, we propose such an automatic cross-over determination method and we embody our ideas in the installation process of ATLAS so as to combine the performance of tuned dense kernels – at the low level– with Strassen's recursive division process –at the high level– into a single adaptive algorithm. We present our experimental results for 14 systems where we tested our codes.

Our approach has four advantages over previous approaches. First, our algorithm works for any square matrices; that is, we do not need to pad the original matrices so as to have even-size or, worse, power-of-two matrices [13]. Second, the algorithm has no requirements on the matrix layout, thus, it can be used instead of other MM routines (ATLAS) with no modifications or extra overhead to change the data layout (unlike the method proposed in [19]). 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 $dgemm()$ without any overhead. Third, the recursive division produces balanced subproblems, thus, predictable performance; unlike the division process proposed by Huss-Lederma et al. [16] 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 propose a recursive algorithm that, if the problem size is large enough, can unfold the division process as deep as there is a performance advantage (in contrast to [16]), thus, the cross-over point is determined empirically by micro-benchmarking at installation time (differently as in [19]).

[1] Also *recursion truncation point* [16], the cross-over size 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.

The paper is organized as follows. In Section II, we present a generalization of Strassen's algorithm better suited for adaptation. In section III, we present our technique for switching adaptively from/to different algorithm strategies. In Section IV, we present our experimental results. Finally, in Section V, we present our concluding remarks.

## II. 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 [13], [16], [19] so that it can be applied to any square-matrix size.

| C | | | A | | | B | |
|---|---|---|---|---|---|---|---|
| $C_0$ | $C_1$ | = | $A_0$ | $A_1$ | * | $B_0$ | $B_1$ |
| $C_2$ | $C_3$ | | $A_2$ | $A_3$ | | $B_2$ | $B_3$ |

Fig. 1
Logical decomposition of matrices in sub-matrices

We identify the **size** of a matrix $\mathbf{A} \in \mathbb{M}^{m \times n}$ as $\sigma(\mathbf{A}) = m \times n$, where $m$ is the number of rows and $n$ the number of columns of the matrix $\mathbf{A}$. Notice that matrices $\mathbf{C}$, $\mathbf{A}$ and $\mathbf{B}$ in the MM computation are composed by four balanced sub-matrices in an identical fashion (see Figure 1).

Now consider the operand matrix $\mathbf{A}$ with $\sigma(\mathbf{A}) = n \times n$, then $\mathbf{A}$ is logically composed by four **near-square** matrices, that is, every matrix has number of rows $r$ and number of columns $c$ that differ by at most one, i.e., $|r - c| \leq 1$, [9]. In fact, we have $\mathbf{A}_0$ with $\sigma(\mathbf{A}_0) = \lceil \frac{n}{2} \rceil \times \lceil \frac{n}{2} \rceil$, $\mathbf{A}_1$ with $\sigma(\mathbf{A}_1) = \lceil \frac{n}{2} \rceil \times \lfloor \frac{n}{2} \rfloor$, $\mathbf{A}_2$ with $\sigma(\mathbf{A}_2) = \lfloor \frac{n}{2} \rfloor \times \lceil \frac{n}{2} \rceil$ and $\mathbf{A}_3$ with $\sigma(\mathbf{A}_3) = \lfloor \frac{n}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$.

The classical MM of $\mathbf{C} = \mathbf{AB}$ can be expressed as the multiplication of the sub-matrices 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 sub-matrix 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 MAs. 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 sub-matrix destination $\mathbf{C}_i$. If we decide to compute the products recursively, that is, each product is divided in further four subproblems, then the matrices involved in the subcomputations are near-square matrices and the computation applies unchanged [9].

Strassen proposed to trade MAs in place of MMs so as to reduce the number of subproblems. He proposed to divide the MM into only 7 MMs and 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. This is Strassen's original formulation. We adapt Strassen's algorithm so as to compute the MM for every square-matrix size as follows:

$$\mathbf{C}_0 = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \quad \mathbf{C}_1 = \mathbf{M}_2 + \mathbf{M}_4$$
$$\mathbf{C}_2 = \mathbf{M}_3 + \mathbf{M}_5 \quad \mathbf{C}_3 = \mathbf{M}_1 + \mathbf{M}_3 - \mathbf{M}_2 + \mathbf{M}_6$$

And every $\mathbf{M}_i$ is defined as follow:

$$\mathbf{M}_1 = \mathbf{T}_0\mathbf{T}_1 \quad \mathbf{T}_0 = \mathbf{A}_0 + \mathbf{A}_3 \text{ and } \mathbf{T}_1 = \mathbf{B}_0 + \mathbf{B}_3$$
$$\sigma(\mathbf{T}_0) = \sigma(\mathbf{T}_1) = \sigma(\mathbf{M}_1) = \lceil n \rceil \times \lceil n \rceil$$
$$\mathbf{M}_2 = \mathbf{T}_2\mathbf{B}_0 \quad \mathbf{T}_2 = \mathbf{A}_2 + \mathbf{A}_3$$
$$\sigma(\mathbf{T}_2) = \sigma(\mathbf{M}_2) = \lfloor n \rfloor \times \lceil n \rceil$$
$$\mathbf{M}_3 = \mathbf{A}_0\mathbf{T}_3 \quad \mathbf{T}_3 = \mathbf{B}_1 + \mathbf{B}_3$$
$$\sigma(\mathbf{T}_3) = \sigma(\mathbf{M}_3) = \lceil n \rceil \times \lfloor n \rfloor$$
$$\mathbf{M}_4 = \mathbf{A}_3\mathbf{T}_4 \quad \mathbf{T}_4 = \mathbf{B}_2 - \mathbf{B}_0$$
$$\sigma(\mathbf{T}_4) = \sigma(\mathbf{M}_4) = \lfloor n \rfloor \times \lceil n \rceil$$
$$\mathbf{M}_5 = \mathbf{T}_5\mathbf{B}_3 \quad \mathbf{T}_5 = \mathbf{A}_0 + \mathbf{A}_1$$
$$\sigma(\mathbf{M}_5) = \sigma(\mathbf{T}_5) = \lceil n \rceil \times \lfloor n \rfloor$$
$$\mathbf{M}_6 = \mathbf{T}_6\mathbf{T}_7 \quad \mathbf{T}_6 = \mathbf{A}_2 - \mathbf{A}_0 \text{ and } \mathbf{T}_7 = \mathbf{B}_0 + \mathbf{B}_1$$
$$\sigma(\mathbf{M}_6) = \sigma(\mathbf{T}_6) = \sigma(\mathbf{T}_7) = \lceil n \rceil \times \lceil n \rceil$$
$$\mathbf{M}_7 = \mathbf{T}_8\mathbf{T}_9 \quad \mathbf{T}_8 = \mathbf{A}_1 - \mathbf{A}_3 \text{ and } \mathbf{T}_9 = \mathbf{B}_2 + \mathbf{B}_3$$
$$\sigma(\mathbf{T}_8) = \lceil n \rceil \times \lfloor n \rfloor, \ \sigma(\mathbf{T}_9) = \lfloor n \rfloor \times \lceil n \rceil$$
$$\sigma(\mathbf{M}_7) = \lceil n \rceil \times \lceil n \rceil$$

To reduce the total number of multiplications, the algorithm computes some **artificial products** that are not necessary for the final result. For example, 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 combining MAs (e.g., $\mathbf{M}_1 + \mathbf{M}_4$).

We notice that the matrices $\mathbf{A}_i$, $\mathbf{B}_i$ and $\mathbf{C}_i$ are near-square matrices but MAs and MMs are not *well defined*; that is, the algorithm is left with the addition of matrices that may have different number of rows/columns such as in $\mathbf{T}_0 = \mathbf{A}_0 + \mathbf{A}_3$, or MM where the left-operand column number is different from the right-operand row number, such as in $\mathbf{A}_3(\mathbf{B}_2 - \mathbf{B}_0)$.

In the following, we present our generalization of the matrix computations achieving a correct algorithm for any matrix sizes. First, we generalize **matrix addition**. Intuitively, when the resulting matrix $\mathbf{X}$ is larger than the addenda $\mathbf{Y}$ 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 if the matrix operands are cropped to fit the result matrix. Formally, $\mathbf{X} = \mathbf{Y} + \mathbf{Z}$ is defined so that $\sigma(\mathbf{X}) = m \times n$, $\sigma(\mathbf{Y}) = p \times q$ and $\sigma(\mathbf{Z}) = r \times s$ and $x_{i,j} = f(i,j) + g(i,j)$ where

$$f(i,j) = \begin{cases} y_{i,j} & \text{if } 0 \leq i < p \wedge 0 \leq j < q \\ 0 & \text{otherwise} \end{cases}$$

$$g(i,j) = \begin{cases} z_{i,j} & \text{if } 0 \leq i < r \wedge 0 \leq j < s \\ 0 & \text{otherwise} \end{cases}$$

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 $x_{i,j} = \sum_{k=0}^{\min(q,r)} y_{i,k} * z_{k,j}$.

Both MA and MM introduce negligible overhead. In fact, the matrices involved in the computations are always near-square matrices, thus their sizes may differ by at most one. Such a litte size difference introduces negligible extra control for the matrix sizes tested in this work.[2] We explain how the two approaches, that is, our version of Strassen's and tuned ATLAS routines are combined in Section III.

In exact arithmetic, this algorithm is correct because we are able to annihilate all components of the artificial terms and therefore $\mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 = \mathbf{A}_0\mathbf{B}_0 + \mathbf{A}_1\mathbf{B}_2$, $\mathbf{M}_2 + \mathbf{M}_4 = \mathbf{A}_0\mathbf{B}_1 + \mathbf{A}_1\mathbf{B}_3$, $\mathbf{M}_3 + \mathbf{M}_5 = \mathbf{A}_2\mathbf{B}_0 + \mathbf{A}_3\mathbf{B}_2$, and $\mathbf{M}_1 + \mathbf{M}_3 - \mathbf{M}_2 + \mathbf{M}_6 = \mathbf{A}_2\mathbf{B}_1 + \mathbf{A}_3\mathbf{B}_3$. Thus, this property can be applied recursively on the products composing the terms

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

$\mathbf{M}_i$. In the case of non-exact arithmetic, Strassen's algorithm has been proved weakly stable; that is, Brent and Higham independently, and 20 years apart, [14], [5] show that the difference between the *real* result $\mathbf{C}$ and the one obtained by Strassen, $\dot{\mathbf{C}}$, is as it follows:

$$\|\mathbf{C} - \dot{\mathbf{C}}\| \leq [(\frac{n}{n_1})^{\log 12}(n_1^2 + 5n_1) - 5n]u\|\mathbf{A}\|\|\mathbf{B}\| \quad (1)$$

where $\|A\| = \max_{ij}|a_{ij}|$, $n_1$ is the size where Strassen's algorithm yields to the usual MM, and $u$ is the inherent floating point precision. This is a weaker error bound than the one for the usual MM. As Highman noticed in his original work, the stability of the algorithm is worsening as the depth of the recursion increases. Fortunately, as we shall see in Section IV, the practical unfolding of Strassen is no larger than three; that is, $n_1 = n/8$ and, thus, the error is simply $\|\mathbf{C} - \dot{\mathbf{C}}\| \leq 27n^2u\|\mathbf{A}\|\|\mathbf{B}\| + O(u^2)$.

In the literature, divers authors have discussed the benefits of a recursive layout for the matrices in such a way to improve data reuse in cache or block transfer among different cache levels in the memory hierarchy (e.g., [20], [19], [9]). In our algorithm, we do not follow this approach for the following four reasons. First, in modern architectures, the memory hierarchy has available (4+ way) associative caches for which the effects of cache interferences, due to the matrix layout, is relatively minimal. Second, Strassen's algorithm for the computation of the MA allocates a smaller working space, dynamically and arbitrarily, where the resulting matrices are stored, so the effect of interference can be reduced further. Third, we have noticed that the data layout may improve the MM performance (i.e., making it faster) for large problem sizes, however it does not improve the MA kernels. Thus, interestingly, Strassen's algorithm may have a smaller cross-over size when the operands are stored using a row/column-major format instead of a non-standard format, such as Z-Morton (we shall discuss this topic in more detail in Section III). This means that Strassen's algorithm has practical effect for smaller cross-over sizes for MM when the operands are stored in row-major format. Fourth 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. We also reorganized the original Strassen's computation so as to use only three temporary matrices, as already proposed in the literature [16].

## III. Cross-Over Size: Empirical Considerations

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. 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 that the optimal strategy is a function of the problem size and of the underlying system.

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,[3] 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, MA have little data reuse and, thus, data-access latency time cannot be circumvented or hidden; for this kernel, 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 over the regular MM only when the saving 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 cross-over size. In fact, a few authors have found that for any problem size Strassen's (or Winograd's variation) is

[3] We use the order proposed by the authors in [21]

TABLE I

| System | Processors | $\pi^{-1}$ $\times 10^6$ | $\alpha^{-1}$ $\times 10^6$ | $\mathbf{n_1}$ | exp. $\dot{\mathbf{n}}_1$ | Figure |
|---|---|---|---|---|---|---|
| Fujitsu HAL 300 | SPARC64 100MHz | 177 | 10 | 390 | 400 | Fig. 2 |
| RX1600 | Itanium 2@1.0GHz | 3023 | 105 | 487 | 725 | Fig. 3 |
| ES40 | Alpha ev67 4@667MHz | 1240 | 41 | 665 | 700 | Fig. 4 |
| RP5470 | 8600 PA-RISC 550MHz | 763 | 21 | 772 | 1175 | Fig. 5 |
| Ultra 5 | UltraSparc2 300MHz | 407 | 9 | 984 | 1225 | Fig. 6 |
| ProLiant DL140 | Xeon 2@3.2GHz | 2395 | 53 | 995 | 1175 | Fig. 7 |
| ProLiant DL145 | Opteron 2@2.2GHz | 3888 | 93 | 918 | 1175 | Fig. 8 |
| Ultra-250 | UltraSparc2 2@300MHz | 492 | 10 | 1061 | 1300 | No |
| Sun-Fire-V210 | UltraSparc3 1GHz | 1140 | 22 | 1140 | 1150 | Fig. 9 |
| Sun Blade | UltraSparc2 500MHz | 460 | 8 | 1191 | 1884 | No |
| ASUS | AthlonXP 2800+ 2GHz | 2160 | 39 | 1218 | 1300 | Fig. 10 |
| Unknown server | Itanium 2@700MHz | 2132 | 27 | 1737 | 2150 | Fig. 11 |
| Fosa | Pentium III 800MHz | 420 | 4 | 2009 | N/A | No |
| SGI O2 | MIPS 12K 300MHz | 320 | 2 | 2816 | N/A | No |

always faster; a few authors have found that the cross-over size is about 500 for some systems and implementations; and a few others, citing private communications, claim that the cross-over size is larger than 1000 [15], [17], [5], [16], [19].

In the following, we present our approach to determine the cross-over size. In practice, our implementation requires 22 matrix updates (4 copies and 18 additions) and 7 recursive calls. Thus, a tentative cross-over size is the size for which the work of the 22 MAs is equal to the one (avoided) of a single MM: $2(\frac{m}{2})^3 = 22(\frac{m}{2})^2$, that is $m = 22$. This cross-over estimation is based on a bottom up approach and is not really accurate. In fact, as the matrix size increases, the MAs (at the higher levels of the division process) are more expensive because they involve data stored in slower levels of the memory hierarchy with no reuse in the levels below. In practice, a more precise analysis would suggest to stop recursion at level $\ell \geq 1$ so that

$$\max_{\ell>0} \sum_{k=0}^{\ell-1} (\frac{7}{4})^k [\frac{n}{2^{k+1}} \pi_{\frac{n}{2^{k+1}}} - 11\alpha_{\frac{n}{2^k}}] \qquad (2)$$

where $\pi_m$ is the **efficiency coefficient** for MM for matrices of size $m \times m$ and $\alpha_m$ is for MA. Notice that $\pi_m^{-1}$ and $\alpha_m^{-1}$ are the performance of MM and MA represented as FLOPS.

That is, $\ell$ is the number of times we perform the division of the problem, or function unfolding, and for which the difference between the cost of saved multiplications and the cost of the extra additions is maximum. In practice, to save enough multiplications, we need to handle large $n$, but then the matrix will lie on slow memories and, therefore, small $\alpha$ are common. If we assume that $\pi$ and $\alpha$ are constant, we can use the simplified formula: $n > 11\frac{\alpha}{\pi}2^{\ell}$. Moreover, If we assume a ratio $\alpha/\pi = 50$ (i.e., common for the systems adopted in this work when the matrices lie in memory only), we find that the cross-over size is $n_1 > 1100$. Thus, for problems of size smaller than $n_1$, Strassen's algorithm should be avoided; for problems of size $(\ell)n_1 \leq n < (\ell+1)n_1$, we may apply Strassen's $\ell$ times.

Of course, the ratio $\pi/\alpha$ is machine and problem-size dependent, however it is straightforward to determine, even if tedious and time consuming. In fact, the factors $\pi$ and $\alpha$ are easy to estimate by benchmarking and they summarize, clearly and concisely, the characteristics of the underlying architecture so as to easily adapt the algorithm to the ever changing system.

## IV. Experimental Results

We installed our codes and the software package ATLAS on 14 different architectures, Table I. Once

the installation is finished, we measure $\alpha$ and $\pi$, and we determine the cross-over size $n_1 = 22\frac{\pi}{\alpha}$. We then determined experimentally the cross-over size $\dot{n}_1$ based on a simple linear search. The two cross-over sizes may differ as we shall discuss shortly.

We determined the factor $\pi$ and $\alpha$ by measuring the performance of ATLAS $dgemm()$ and our MA for matrices of size $1000 \times 1000$. Though in practice, $\pi$ represents the performance of MM and $\alpha$ represents the performance of MA in isolation, however, these routines interact and share data with other routines. Because MM exploits data locality, our measure of $\pi$ is a lower bound of the effective $\dot{\pi}$, and because MA exploits little data locality, our $\alpha$ is an upper bound of the effective $\dot{\alpha}$. Hence, the experimental cross-over size $\dot{n}_1$ is always larger than the estimated $n_1$.

In this section, we present two measures of performance (Figure 2 - 11): 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, such as MFLOPS, shows whether or not Strassen's algorithm improves the performance of a MM kernel that is either efficiently or poorly designed. This basic measure has been omitted in previous investigations but we consider it important; in fact, this measure shows the performance effects of Strassen's algorithm over the classic MM algorithm.

In the following, we present the experimental results for ten systems. We use the following terminology: **S-adaptive** is the Strassen's algorithm for which the unfolding of the recursion is based on the parameter $n_1 = 22\frac{\pi}{\alpha}$; **S-$k$-unfold** is the Strassen algorithm for which $k$ is the number of times the recursion unfolds before yielding to ATLAS $dgemm$. (Note that we did not report negative relative performance and we omitted the correspondent bar in the charts.) The performance obtained by the systems in Table I, and presented from Figure 2 to Figure 11, are obtained by the collection of the best performance among several trials.

Though, six systems have two or more processors, however, our codes are single threaded and we collected performance results using only one processor at anytime. This is because most of the multiprocessor systems used are a common resource among different users making the system workload unpredictable, thus, we decided to investigate the more reproducible performance (of our codes) on single processor.

## V. Conclusions

We have presented a practical implementation of Strassen's algorithm, which applies an adaptive algorithm to exploit highly tuned MMs, such as ATLAS's. We differ from previous approaches because we use an adaptive recursive algorithm with a balanced division process, which, in turn, makes the algorithm performance more predictable.

We have tested extensively the performance of our approach on 14 systems and we have shown that not always Strassen is applicable. We have also shown that for modern systems the cross-over size can be quite large.

## References


[1] M. Frigo and S. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation*, vol. 93, no. 2, pp. 216–231, 2005.

[2] J. Demmel, J. Dongarra, E. Eijkhout, E. Fuentes, E. Petitet, V. Vuduc, R. Whaley, and K. Yelick, "Self-Adapting linear algebra algorithms and software," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, 2005.

[3] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, 2005.

[4] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," in *Proceedings of the 19-th annual ACM conference on Theory of computing*, 1987, pp. 1–6.

[5] N. J. Higham, "Exploiting fast matrix multiplication within the level 3 BLAS," *ACM Trans. Math. Softw.*, vol. 16, no. 4, pp. 352–368, 1990.

[6] J. Frens and D. Wise, "Auto-Blocking matrix-multiplication or tracking BLAS3 performance from source code," *Proc. 1997 ACM Symp. on Principles and Practice of Parallel Programming*, vol. 32, no. 7, pp. 206–216, July 1997.

[7] N. Eiron, M. Rodeh, and I. Steinwarts, "Matrix multiplication: a case study of algorithm engineering," in *Proceedings WAE'98*, Saarbrücken, Germany, Aug 1998.

[8] R. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, 1998, pp. 1–27.

[9] G. Bilardi, P. D'Alberto, and A. Nicolau, "Fractal matrix multiplication: a case study on portability of cache performance," in *Workshop on Algorithm Engineering 2001*, Aarhus, Denmark, 2001.

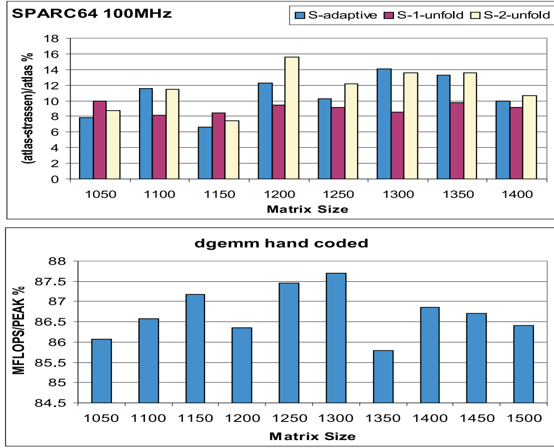


Fig. 2
FUJITSU HAL 300.

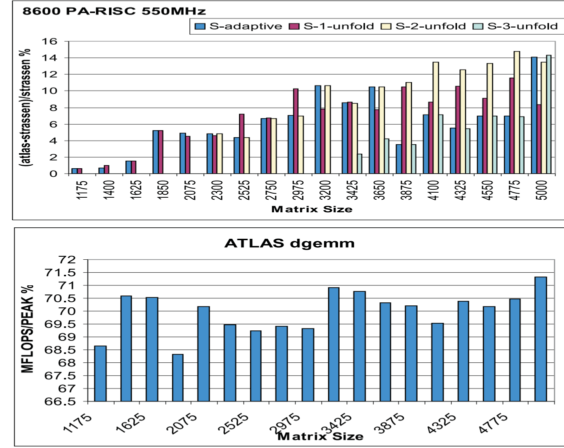


Fig. 5
RP5470.

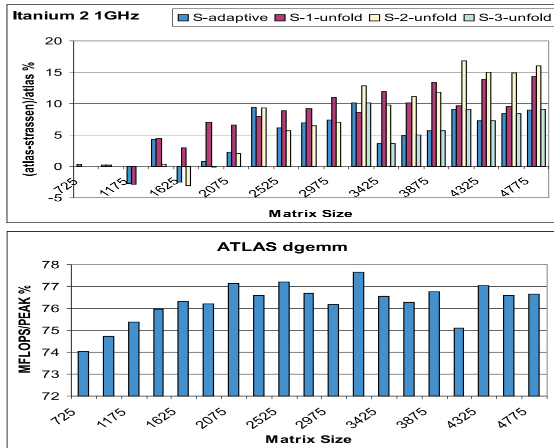


Fig. 3
RX1600.

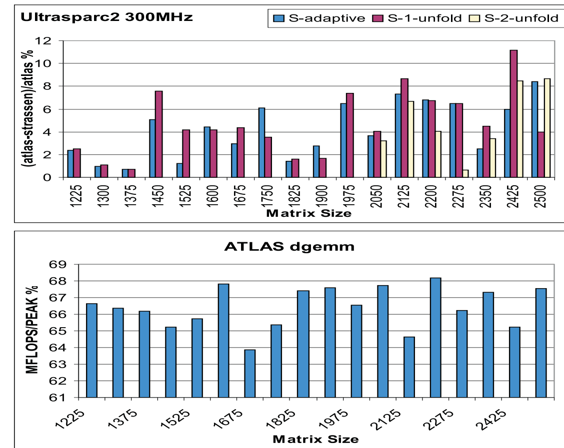


Fig. 6
ULTRA 5.

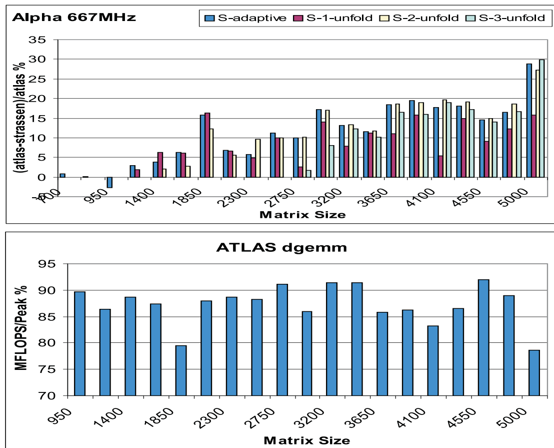


Fig. 4
ES40.

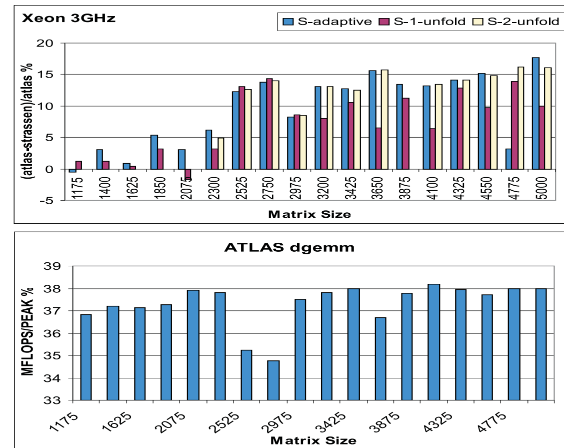


Fig. 7
PROLIANT DL140-

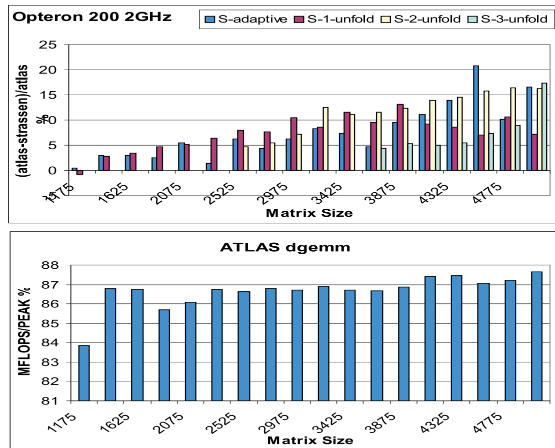


Fig. 8
PROLIANT DL145.

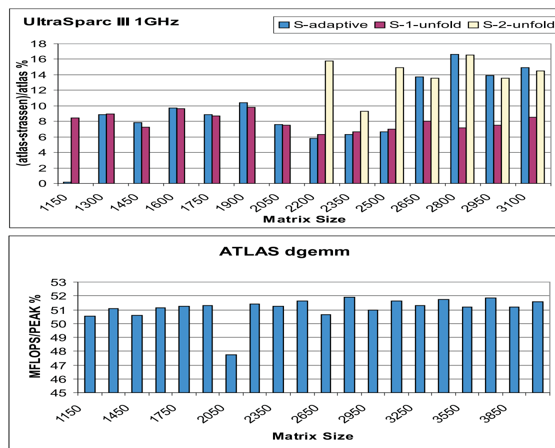


Fig. 9
SUN-FIRE-V210.

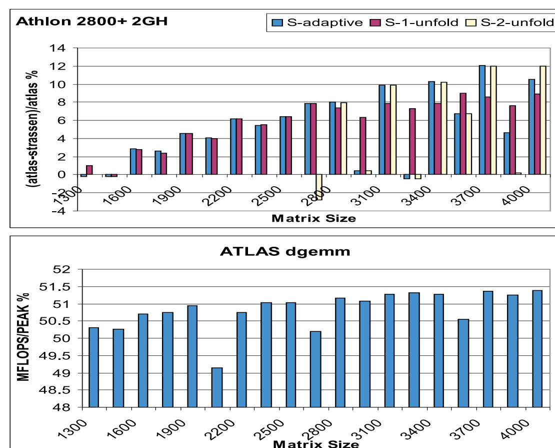


Fig. 10
ASUS A7N8X.

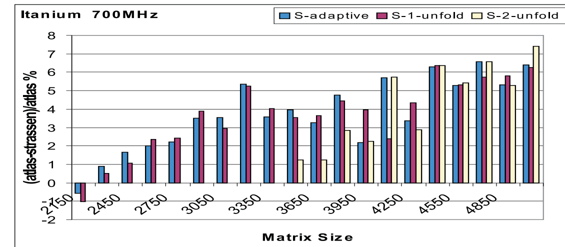


Fig. 11
LINUX ITANIUM 2 700 MHZ.

[10] E. Anderson, Z. Bai, C. Bischof, J. D. J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK User' Guide, Release 2.0*, 2nd ed. SIAM, 1995.

[11] B. Kagstrom, P. Ling, and C. van Loan, "Algorithm 784: GEMM-based level 3 BLAS: portability and optimization issues," *ACM Transactions on Mathematical Software*, vol. 24, no. 3, pp. 303–316, Sept 1998.

[12] ——, "GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark," *ACM Transactions on Mathematical Software*, vol. 24, no. 3, pp. 268–302, Sept 1998.

[13] V. Strassen, "Gaussian elimination is not optimal." *Numerische Mathematik*, vol. 14, no. 3, pp. 354–356, 1969.

[14] R. P. Brent, "Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity," *Numerische Mathematik*, vol. 16, pp. 145–156, 1970.

[15] ——, "Algorithms for matrix multiplication," Stanford University, Tech. Rep. TR-CS-70-157, Mar 1970.

[16] S. Huss-Lederman, E. Jacobson, A. Tsao, T. Turnbull, and J. Johnson, "Implementation of Strassen's algorithm for matrix multiplication," in *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*. ACM Press, 1996, p. 32.

[17] D. H. Bailey and H. R. P. Gerguson, "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, pp. 419–424.

[18] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel, "Optimizing matrix multiply using PHiPAC: a portable, high-performance, Ansi C coding methodology," in *International Conference on Supercomputing*, July 1997.

[19] M. Thottethodi, S. Chatterjee, and A. Lebeck, "Tuning Strassen's matrix multiplication for memory efficiency." in *Proc. Supercomputing*, Orlando, FL, nov 1998.

[20] A. Aggarwal, A. Chandra, and M. Snir, "Hierarchical memory with block transfer," in *28th Annual Symposium on Foundations of Computer Science*, Los Angeles, California, October 1987, pp. 204–216.

[21] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir, "A model for hierarchical memory," in *Proceedings of 19th Annual ACM Symposium on the Theory of Computing*, New York, 1987, pp. 305–314.