Adaptive Strassen's Matrix Multiplication

Paolo D'Alberto
Dept. of Electric and Computer Engineering
Carnegie Mellon University
pdalbert@ece.cmu.edu

Alexandru Nicolau
Dept. of Computer Science
University of California Irvine
nicolau@ics.uci.edu

ABSTRACT

Strassen's matrix multiplication (MM) has benefits with respect to any (highly tuned) implementations of MM because Strassen's reduces the total number of operations. Strassen achieved this operation reduction by replacing computationally expensive MMs with matrix additions (MAs). For architectures with simple memory hierarchies, having fewer operations directly translates into an efficient utilization of the CPU and, thus, faster execution. However, for modern architectures with complex memory hierarchies, the operations introduced by the MAs have a limited in-cache data reuse and thus poor memory-hierarchy utilization, thereby overshadowing the (improved) CPU utilization, and making Strassen's algorithm (largely) useless on its own.

In this paper, we investigate the interaction between Strassen's effective performance and the memory-hierarchy organization. We show how to exploit Strassen's full potential across different architectures. We present an easy-to-use adaptive algorithm that combines a novel implementation of Strassen's idea with the MM from automatically tuned linear algebra software (ATLAS) or GotoBLAS. An additional advantage of our algorithm is that it applies to any size and shape matrices and works equally well with row or column major layout. Our implementation consists of introducing a final step in the ATLAS/GotoBLAS-installation process that estimates whether or not we can achieve any additional speedup using our Strassen's adaptation algorithm. Then we install our codes, validate our estimates, and determine the specific performance.

We show that, by the *right* combination of Strassen's with ATLAS/GotoBLAS, our approach achieves up to 30%/22% speed-up versus ATLAS/GotoBLAS alone on modern high-performance single processors. We consider and present the complexity and the numerical analysis of our algorithm, and, finally, we show performance for 17 (uniprocessor) systems.

Categories and Subject Descriptors

G.4 [**Mathematics of Computing**]: Mathematical Software; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Top-down programming*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. ICS'07 June 18-20, Seattle, WA, USA.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

General Terms

Algorithms

Keywords

matrix multiplications, fast algorithms

1. INTRODUCTION

In the last 30 years, the complexity of processors on a chip is following accurately Moore's law; that is, the number of transistors per chip doubles every 18 months. Unfortunately, the steady increase of processor integration does not always result in a proportional increase of the system performance on a given application (program); that is, the same application does not double its performance when it runs on a new state-of-the-art architecture every 18 months. In fact, the performance of an application is the result of an intricate synergy between the two constituent parts of the system: on one side, the architecture composed of processors, memory hierarchy and devices, and, on the other side, the code composed of algorithms, software packages and libraries steering the computation on the hardware. When the architecture evolves, the code must adapt so that the system can deliver peak performance.

Our main interest is the design and implementation of codes that embrace the architecture evolution. We want to write efficient and easy to maintain codes, which any developer can use for several generations of architectures. **Adaptive codes** attempt to provide just that. In fact, they are an effective solution for the efficient utilization of (and portability across) complex and always-changing architectures (e.g., [16, 11, 30, 19]). In this paper, we discuss a single but fundamental kernel in dense linear algebra: **matrix multiply** (MM) for any size and shape matrices stored in double precision and in standard row or column major layout [20, 15, 14, 33, 3, 18].

We extend Strassen's algorithm to deal with rectangular and arbitrary-size matrices so as to exploit better data locality and number of operations, and thus performance, than previously proposed versions [22]. We consider the performance effects of Strassen's applied to rectangular matrices directly (i.e., exploiting fewer operations) or, after a cache-oblivious problem division, to (almost) square matrices only (i.e., exploiting data locality). We show that for some architectures, the latter can outperform the former, (in contrast of what estimated by Knight [25]), and we show that both must build on top of highly efficient $O(n^3)$ MMs based on the state-of-the-art adaptive software packages such as ATLAS [11] and hand tuned packages such as GotoBLAS [18]. In fact, we show that choosing the right combination of our algorithm with these highly tuned MMs, we can achieve an execution-time reduction up to 30% and 22% when comparet to using alone ATLAS

and GotoBLAS respectively. We present an extensive quantitative evaluation of our algorithm performance for a large set of different architectures to demonstrate that our approach is beneficial and portable. We discuss also the numerical stability of our algorithm and its practical error evaluation.

The paper is organized as follows. In Section 2, we discuss the related work and we highlight our contributions. In Section 3, we present our algorithm and discuss its practical numerical stability and complexity. In Section 4, we present our experimental results; in particular, in Section 4.1, we discuss the performance for any matrix shape and, in Section 4.2, for square matrices only. We conclude in Section 5.

2. RELATED WORK

Strassen's algorithm is the first and the most used *fast algorithm* (i.e., breaking the $O(n^3)$ operation count). Strassen discovered that the original recursive algorithm of complexity $O(n^3)$ can be reorganized in such a way that one *computationally expensive* recursive MM step can be replaced with 18 *cheaper* **matrix additions** (MA). As a result, Strassen's algorithm has (asymptotically) fewer operations (i.e., multiplications and additions) $O(n^{2.86})$. Another variant is by Winograd; Winograd replaced one MM with 15 MAs and improved Strassen's complexity by a constant factor. Our approach can be applied to the Winograd's variant as well; however, Winograd's algorithm is beyond the scope of this paper and it will be investigated separately and in the future.

In practice however for small matrices, Strassen's has a significant overhead and a conventional MM results in better performance. To overcome this, several authors have shown *hybrid* algorithms; that is, deploying Strassen's MM in conjunction with conventional MM [5, 4, 20], where for a specific problem size n_1, or **recursion point** [22], Strassen's algorithm yields the computation to the conventional MM implementations. [1] With the deployment of modern and faster architectures (with fast CPUs and relatively slow memory), Strassen's has performance appeal for larger and larger problems, undermining its *practical* benefits because the we should use Strassen for even larger problems. In other words, the evolution of modern architectures —i.e., capable of solving large problems fast— presents a scenario where the recursion point has started increasing [2]. However, the demand for solving larger and larger problems has increased together with the development of modern architectures; this sheds a completely new light on what/when a problem size is actually practical, making Strassen's approach extremely powerful. Given a modern architecture, finding for what problem sizes Strassen's is beneficial has never been so compelling.

Our approach has three contributions/advantages with respect to previous approaches.

1. Our algorithm divides the MM problems into a set of balanced subproblems without any matrix padding or peeling, so it achieves balanced workload and predictable performance. This balanced division leads to: a cleaner algorithm formulation, an easier parallelization, and little or no work in combining the solutions of the subproblems (*conquer step*). This strategy differs from the division process proposed by Huss-Lederman et al. [22, 20, 1] leading to fewer operations and better data locality. In this approach, the problem division is a function of the matrix sizes such that for odd-matrix sizes, the problem is divided into a large even-size problem, on which Strassen can be applied, and into (extremely irregular) subproblems deploying matrix-by-vector and vector-by-vector computations.

2. Our algorithm applies Strassen's strategy recursively as many time as a function of the problem size. If the problem size is large enough, the algorithm has a recursion depth that goes as deep as there is any performance advantage. This is in contrast to the approach in [22] where Strassen's strategy is applied just once. Furthermore, we determine the recursion point empirically by micro-benchmarking at installation time; unlike as in [32], where Strassen's algorithm is studied in isolation without any performance comparison with high-performance MM.

3. We store matrices in standard row or column major format and, at any time, we can yield control to a highly tuned MM such as ATLAS's $DGEMM$ without any overhead. Thus, we can use our algorithm in combination with these highly tuned MM routines with no modifications or change of layout overheads (i.e., estimated as 5–10% of the total execution time [32]).

In the literature, there are *other* fast MM algorithms. For example, Pan showed a bilinar algorithm that is asymptotically faster than Strassen-Winograd [27] $O(n^{2.79})$ and he presented a survey of the topic [28] with best $O(n^{2.49})$. The practical implementation of Pan's algorithm is presented by Kaporini [23, 24]. New approaches are emerging recently, which promise to be practical and numerically stable [7, 12]. The fastest to date is by Coppersmith and Winograd [8] $O(n^{2.376})$.

3. BALANCED MATRIX MULTIPLICATION

In this section, we introduce our algorithm, which is a composition of three different layers/algorithms: on the top level (in this section), we use a cache oblivious algorithm [17] so to reduce the problem to almost square matrices; in the middle level (Section 3.1), we deploy Strassen's algorithm to reduce the computation work; in the lower level, we deploy ATLAS's [33] or GotoBLAS [18] to unleash the architecture characteristics. In the following, we introduce briefly our notations and then our algorithms.

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 $\mathbf{A}$. Matrix multiplication is defined for operands of sizes $\sigma(\mathbf{C})=m\times p$, $\sigma(\mathbf{A})=m\times n$ and $\sigma(\mathbf{B})=n\times p$, and identified as $\mathbf{C}=\mathbf{AB}$, where the component $c_{i,j}$ at row i and column j of the result matrix $\mathbf{C}$ is defined as $c_{i,j} = \sum_{k=0}^{n} a_{i,k}b_{k,j}$.

In this paper, we use a simplified notation to identify submatrices. We choose to divide logically a matrix $\mathbf{M}$ into at most four submatrices; we label them so that $\mathbf{M}_0$ is the first and the largest submatrix, $\mathbf{M}_2$ is logically beneath $\mathbf{M}_0$, $\mathbf{M}_1$ is on the right of $\mathbf{M}_0$, and $\mathbf{M}_3$ is beneath $\mathbf{M}_1$ and to the right of $\mathbf{M}_2$ (e.g., how to divide a matrix into two submatrices, and into four, see Figure 1). This notation is taken from [6].

In Table 1, we present the framework of our cache-oblivious algorithm that we identify as **Balanced** MM. This algorithm divides the problem and specifies the operands in such a way that the subproblems have balanced workload and similar operand shapes and sizes. In fact, this algorithm reduces the problem to **almost square matrices**; that is, $\mathbf{A}$ is almost square if $m \leq \gamma_m n$ or $n \leq \gamma_n m$ and γ is usually equal to 2. We aim at the efficient utilization of the

[1]Thus, for a problem of size $n \leq n_1$, this hybrid algorithm is the conventional MM; for every matrix size $n \geq n_1$, the hybrid algorithm is faster because it applies Strassen's strategy and thus it exploits all its performance benefits.

Table 1: Balanced Matrix Multiplication $\mathbf{C}$=$\mathbf{AB}$ with $\sigma(\mathbf{A})$=$m\times n$ and $\sigma(\mathbf{B})$=$n\times p$

Computation	Operand Sizes
if $m \geq \gamma_m \max(n,p)$ then $\mathbf{C}_0$=$\mathbf{A}_0\mathbf{B}$ $\mathbf{C}_2$=$\mathbf{A}_2\mathbf{B}$	γ_m=2 ($\mathbf{A}$ is *tall*) $\sigma(\mathbf{A}_0)$=$\lceil\frac{m}{2}\rceil\times n$, $\sigma(\mathbf{C}_0)$=$\lceil\frac{m}{2}\rceil\times p$, $\sigma(\mathbf{A}_2)$=$\lfloor\frac{m}{2}\rfloor\times n$, $\sigma(\mathbf{C}_2)$=$\lfloor\frac{m}{2}\rfloor\times p$
else if $p \geq \gamma_p \max(m,n)$ then $\mathbf{C}_0$=$\mathbf{A}\mathbf{B}_0$ $\mathbf{C}_1$=$\mathbf{A}\mathbf{B}_1$	γ_p=2 ($\mathbf{B}$ is *long*) $\sigma(\mathbf{C}_0)$=$m\times\lceil\frac{p}{2}\rceil$, $\sigma(\mathbf{B}_0)$=$n\times\lceil\frac{p}{2}\rceil$, $\sigma(\mathbf{C}_1)$=$m\times\lfloor\frac{p}{2}\rfloor$, $\sigma(\mathbf{B}_1)$=$n\times\lfloor\frac{p}{2}\rfloor$,
else if $n \geq \gamma_n \max(m,p)$ then $\mathbf{C}$=$\mathbf{A}_0\mathbf{B}_0$ $\mathbf{C}$=$\mathbf{C}+\mathbf{A}_1\mathbf{B}_2$	γ_n=2 ($\mathbf{B}$ is *tall* and $\mathbf{A}$ is *long*) $\sigma(\mathbf{A}_0)$=$m\times\lceil\frac{n}{2}\rceil$, $\sigma(\mathbf{B}_0)$=$\lceil\frac{n}{2}\rceil\times p$ $\sigma(\mathbf{A}_1)$=$m\times\lfloor\frac{n}{2}\rfloor$, $\sigma(\mathbf{B}_2)$=$\lfloor\frac{n}{2}\rfloor\times p$
else Strassen $\mathbf{C}$=$\mathbf{A}*_s\mathbf{B}$	$\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ almost square see Section 3.1

higher level of the memory hierarchy; that is, the memory pages often organized using small and fully associative cache, table look-aside buffer (TLB). This formulation has been proven (asymptotically) optimal [17] in the number of misses. We present detailed performance of this strategy for any matrix size and for three systems in Section 4.

This technique breaks down the general problem into small and regular problems to exploit better data locality in the memory hierarchy. Knight [25] presented evidence showing that this approach is not optimal in the sense of number of operations; for example, using directly Strassen decomposition to the rectangular matrices should achieve better performance. In Section 4.1, we address this issue quantitatively and show that we must exploit data locality as much as the operation reduction.

In the following section (Section 3.1), we describe our version of Strassen's algorithm for any (almost square) matrices and, thereby completing the algorithm specified in Table 1. We call it hybrid ATLAS/GotoBLAS–Strassen algorithm (HASA).

3.1 Hybrid ATLAS/GotoBLAS–Strassen algorithm (HASA)

In this section, we present our generalization of Strassen's MM algorithm. Our algorithm reduces the number of passes through the data as well as the number of computations because of a balanced division process. In practice, this algorithm is more efficient than previous approaches [31, 22, 32]. The algorithm applies to any matrix sizes (and, thus, shape) rather than only to square matrices like in [9, 10].

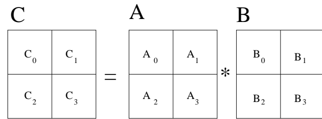

Figure 1: Logical decomposition of matrices in sub-matrices

Matrices $\mathbf{C}$, $\mathbf{A}$ and $\mathbf{B}$ in the MM computation are composed of four balanced sub-matrices (see Figure 1). Consider the operand matrix $\mathbf{A}$ with $\sigma(\mathbf{A})$=$m\times n$. Now, $\mathbf{A}$ is logically composed of four matrices: $\mathbf{A}_0$ with $\sigma(\mathbf{A}_0)$=$\lceil\frac{m}{2}\rceil\times\lceil\frac{n}{2}\rceil$, $\mathbf{A}_1$ with $\sigma(\mathbf{A}_1)$=$\lceil\frac{m}{2}\rceil\times\lfloor\frac{n}{2}\rfloor$, $\mathbf{A}_2$ with $\sigma(\mathbf{A}_2)$=$\lfloor\frac{m}{2}\rfloor\times\lceil\frac{n}{2}\rceil$ and $\mathbf{A}_3$ with $\sigma(\mathbf{A}_3)$=$\lfloor\frac{m}{2}\rfloor\times\lfloor\frac{n}{2}\rfloor$ (similarly, $\mathbf{B}$ is composed of four submatrices where $\sigma(\mathbf{B}_0)$=$\lceil\frac{n}{2}\rceil\times\lceil\frac{p}{2}\rceil$).

We generalized Strassen's algorithm in order to compute the MM *regardless of matrix size*, as shown in Table 2. Notice that, dividing a matrix into almost square submatrices as described previously, we can achieve a balanced division into subcomputations. However, now we do not have submatrices of the same sizes. In such a scenario, we defined the MA between matrices so that we have a fully functional recursive algorithm; that is, we reduce the computation to 7 *well defined* MMs, where the left-operand columns match the right operand rows, and MAs are between almost square matrices as follows.

Consider a matrix addition $\mathbf{X}$=$\mathbf{Y}+\mathbf{Z}$ (subtraction is similar). Intuitively, when the resulting matrix $\mathbf{X}$ is larger than the addenda $\mathbf{Y}$ or $\mathbf{Z}$, the computation is performed as if the smaller operands are *extended* and padded with zeros. Otherwise, if the result matrix is smaller than the operands, the computation is performed as if the larger 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:

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

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

Table 2: HASA $\mathbf{C}$=$\mathbf{A}*_s\mathbf{B}$ with $\sigma(\mathbf{A})$=$m\times n$ and $\sigma(\mathbf{B})$=$n\times p$

Computation	Operand Sizes
if RecursionPoint($\mathbf{A}$,$\mathbf{B}$) then ATLAS/GotoBLAS $\mathbf{C}$=$\mathbf{A}*_a\mathbf{B}$	(e.g., $\max(m,n,p) < 100$) (*Solve directly*)
else { $\mathbf{T}_2$=$\mathbf{B}_1-\mathbf{B}_3$ $\mathbf{M}_3$=$\mathbf{A}_0*_s\mathbf{T}_2$ $\mathbf{C}_1$=$\mathbf{M}_3$, $\mathbf{C}_3$=$\mathbf{M}_3$	(*Divide et impera*) $\sigma(\mathbf{T}_2)$=$\lceil\frac{n}{2}\rceil\times\lfloor\frac{p}{2}\rfloor$ $\sigma(\mathbf{M}_3)$=$\lceil\frac{m}{2}\rceil\times\lfloor\frac{p}{2}\rfloor$
$\mathbf{T}_1$=$\mathbf{A}_2-\mathbf{A}_0$ $\mathbf{T}_2$=$\mathbf{B}_0+\mathbf{B}_1$ $\mathbf{M}_6$=$\mathbf{T}_1*_s\mathbf{T}_2$ $\mathbf{C}_3$=$\mathbf{C}_3$+$\mathbf{M}_6$	$\sigma(\mathbf{T}_1)$=$\lceil\frac{m}{2}\rceil\times\lceil\frac{n}{2}\rceil$ $\sigma(\mathbf{T}_2)$=$\lceil\frac{n}{2}\rceil\times\lceil\frac{p}{2}\rceil$ $\sigma(\mathbf{M}_6)$=$\lceil\frac{m}{2}\rceil\times\lceil\frac{p}{2}\rceil$
$\mathbf{T}_1$=$\mathbf{A}_2+\mathbf{A}_3$ $\mathbf{M}_2$=$\mathbf{T}_1*_s\mathbf{B}_0$ $\mathbf{C}_2$=$\mathbf{M}_2$, $\mathbf{C}_3$=$\mathbf{C}_3$−$\mathbf{M}_2$	$\sigma(\mathbf{T}_1)$=$\lfloor\frac{m}{2}\rfloor\times\lceil\frac{n}{2}\rceil$ $\sigma(\mathbf{M}_2)$=$\lfloor\frac{m}{2}\rfloor\times\lceil\frac{p}{2}\rceil$
$\mathbf{T}_1$=$\mathbf{A}_0+\mathbf{A}_3$ $\mathbf{T}_2$=$\mathbf{B}_0+\mathbf{B}_3$ $\mathbf{M}_1$=$\mathbf{T}_1*_s\mathbf{T}_2$ $\mathbf{C}_0$=$\mathbf{M}_1$, $\mathbf{C}_3$=$\mathbf{C}_3$+$\mathbf{M}_1$	$\sigma(\mathbf{T}_1)$=$\lceil\frac{m}{2}\rceil\times\lceil\frac{n}{2}\rceil$ $\sigma(\mathbf{T}_2)$=$\lceil\frac{n}{2}\rceil\times\lceil\frac{p}{2}\rceil$ $\sigma(\mathbf{M}_1)$=$\lceil\frac{m}{2}\rceil\times\lceil\frac{p}{2}\rceil$
$\mathbf{T}_1$=$\mathbf{A}_0+\mathbf{A}_1$ $\mathbf{M}_5$=$\mathbf{T}_1*_s\mathbf{B}_3$ $\mathbf{C}_0$=$\mathbf{C}_0$−$\mathbf{M}_5$, $\mathbf{C}_1$=$\mathbf{C}_1$+$\mathbf{M}_5$	$\sigma(\mathbf{T}_1)$=$\lceil\frac{m}{2}\rceil\times\lfloor\frac{n}{2}\rfloor$ $\sigma(\mathbf{M}_5)$=$\lceil\frac{m}{2}\rceil\times\lfloor\frac{p}{2}\rfloor$
$\mathbf{T}_1$=$\mathbf{A}_1-\mathbf{A}_3$ $\mathbf{T}_2$=$\mathbf{B}_2+\mathbf{B}_3$ $\mathbf{M}_7$=$\mathbf{T}_1*_s\mathbf{T}_2$ $\mathbf{C}_0$=$\mathbf{C}_0$+$\mathbf{M}_7$	$\sigma(\mathbf{T}_1)$=$\lceil\frac{m}{2}\rceil\times\lfloor\frac{n}{2}\rfloor$ $\sigma(\mathbf{T}_2)$=$\lfloor\frac{n}{2}\rfloor\times\lceil\frac{p}{2}\rceil$ $\sigma(\mathbf{M}_7)$=$\lceil\frac{m}{2}\rceil\times\lceil\frac{p}{2}\rceil$
$\mathbf{T}_2$=$\mathbf{B}_2-\mathbf{B}_0$ $\mathbf{M}_4$=$\mathbf{A}_3*_s\mathbf{T}_2$ $\mathbf{C}_0$=$\mathbf{C}_0$+$\mathbf{M}_4$, $\mathbf{C}_2$=$\mathbf{C}_2$+$\mathbf{M}_4$ }	$\sigma(\mathbf{T}_2)$=$\lfloor\frac{n}{2}\rfloor\times\lceil\frac{p}{2}\rceil$ $\sigma(\mathbf{M}_4)$=$\lfloor\frac{m}{2}\rfloor\times\lceil\frac{p}{2}\rceil$

Correctness. To reduce the total number of multiplications, Strassen's algorithm computes *implicitly* some products that are necessary and some that are artificial and must be carefully re-

moved from the final result. For example, the product $\mathbf{A}_0\mathbf{B}_0$, which is a term of $\mathbf{M}_1$, is a **singular product** and it is required for the computation of $\mathbf{C}_0$; in contrast, $\mathbf{A}_0\mathbf{B}_3$ is an **artificial product**,[2] computed in the same expression, because of the way the algorithm adds submatrices of **A** and **B** together in preparation of the recursive MM. Every artificial product must be removed (i.e., subtracted) by combining the different MAs (e.g., $\mathbf{M}_1 + \mathbf{M}_5$) so as to achieve the final correct result. By construction, Strassen's algorithm is correct; here, the only concern is our adaptation to any matrix sizes: First, we note that all MM are well defined —i.e., the number of columns of the left operands is equal to the number of rows of the right operand. Second, all singular products are correctly computed and added. Third, all artificial products are correctly computed and removed.

3.2 Numerical considerations

The classic MM has component-wise and norm-wise error bounds as follows:

$$\text{Component-wise: } |\mathbf{C} - \dot{\mathbf{C}}| \leq nu|\mathbf{A}| * |\mathbf{B}| + O(u^2),$$
$$\text{Norm-wise: } \|\mathbf{C} - \dot{\mathbf{C}}\| \leq n^2u\|\mathbf{A}\|\|\mathbf{B}\| + O(u^2)\ .$$

where we identify the *exact* result with $\mathbf{C}$ and the computed solution with $\dot{\mathbf{C}}$, $|\mathbf{A}|$ has components $|a_{i,j}|$, and $\|A\| = \max_{ij} |a_{ij}|$.

For this algorithm, we can apply the same numerical analysis used for Strassen's algorithm. Strassen's has been proved weakly stable. Brent and Higham [5, 20] showed that the stability of the algorithm is worsening as the depth of the recursion increases.

$$\begin{aligned} \|\mathbf{C} - \dot{\mathbf{C}}\| &\leq [(\tfrac{n}{n_1})^{\log 12}(n_1^2 + 5n_1) - 5n]u\|\mathbf{A}\|\|\mathbf{B}\| + O(u^2) \\ &\leq 3^\ell n^2 u\|\mathbf{A}\|\|\mathbf{B}\| + O(u^2) \end{aligned} \tag{1}$$

Again, n_1 is the size where Strassen's algorithm yields to the usual MM, and u is the inherent floating point precision. We denote with ℓ the **recursion depth**; that is, the number of times we divide the problem. This is a weaker error bound (a norm-wise bound) than the one for the classic algorithm (a component-wise bound) and it is also a pessimistic estimation (i.e., in practice). Demmel and Higman [13] have shown that, in practice, fast MMs can lead to fast and accurate results when they are used in blocked computation of the LAPACK routines.

Here, we follows the same procedure used by Higham [21] to quantify the error for our algorithm and for large problems sizes such as to investigate the effect of error in double precision (Figure 2).

Input. We restrict the input matrix values to a specific range or intervals: $[-1, 1]$ and $[0, 1]$. We then initialize the input matrices using a uniformly distributed random number generator.

Reference Doubly Compensated Summation. We consider the output of the computation $\mathbf{C} = \mathbf{AB}$. We compute every element c_{ij} independently and by performing first a *dot* product of the row vector a_{i*} by the column vector b_{*j} and we store it into a temporary vector $\mathbf{z}$. Then, we sort the vector in decreasing order such that $|z_i| \geq |z_j|$ with $i < j$ using the adaptive sorting library [26]. Then we compute the reference output using Priest's **doubly compensated summation** (DCS) procedure [29] in extended double precision as described in [21]. This is our baseline or ultimate reference.

Comparison. We compare the output-value difference (w.r.t of the DCS based MM) of ATLAS algorithm, HASA (Strassen algorithm using ATLAS's MM) and the naive **row-by-column** algorithm (RBC) using an accumulator register, for which the summation is not compensated and the values are in the original order.

[2]The product $\mathbf{A}_0\mathbf{B}_3$ cannot be computed directly because is not defined in general: the $columns(\mathbf{A}_0) > rows(\mathbf{B}_3)$ and thus we should pad $\mathbf{B}_3$ or crop $\mathbf{A}_0$ properly.

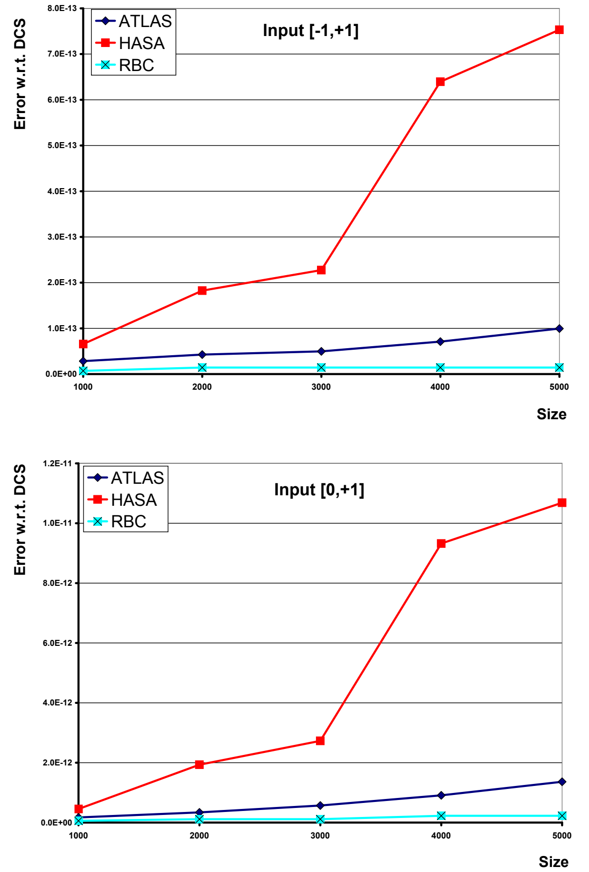

Figure 2: Pentium 4 3.2GHz Maximum error estimation: Recursion point n_1=900, 1 recursion $900 \leq n \leq 1800$, 2 recursions $1800 \leq n \leq 3600$, and 3 recursions $3600 \leq n$

Considerations. In Figure 2, we show the error evaluation w.r.t. the DCS MM for square matrices only. For Strassen's algorithm the error ratio of HASA over ATLAS is no larger than 10 for both ranges $[0, 1]$ and $[-1, 1]$. These error ratios are less dramatic than what an upper bound analysis would suggest. That is, each recursion could increases the error as 2^ℓ instead of 3^ℓ, and in practice, no more than 10 (one precision digit w.r.t. the 16 available).

3.3 Recursion Point and Complexity

Strassen's algorithm embodies two different locality properties because its two basic computations exploit different data locality: matrix multiply MM has spatial *and* temporal locality, and matrix addition MA has only spatial locality. In this section, we describe how these data reuses affect the algorithm performance and thus our strategy.

The 7 MMs take $2\pi(\lceil\frac{m}{2}\rceil\lceil\frac{n}{2}\rceil p + mn\lceil\frac{p}{2}\rceil + \lceil\frac{m}{2}\rceil\lfloor\frac{n}{2}\rfloor\lceil\frac{p}{2}\rceil)$ seconds, where π is the efficiency of MM —i.e., *pi* for product— and $\frac{1}{\pi}$ is simply the **floating point operation per second** (FLOPS) of the computation.

The 22 MAs (18 matrix additions and 4 matrix copies) take $\alpha[5\lceil\frac{n}{2}\rceil(\lceil\frac{m}{2}\rceil + \lceil\frac{p}{2}\rceil) + 3mp]$ seconds (see Table 2), where α is the efficiency of MA —i.e., *alpha* for addition. For example, we

perform five MAs of submatrices of $\mathbf{A}$: we perform three additions with $\mathbf{A}_0$, $3\lceil\frac{m}{2}\rceil\lceil\frac{n}{2}\rceil$, one with $\mathbf{A}_1$, $\lceil\frac{m}{2}\rceil\lfloor\frac{n}{2}\rfloor$, and one with $\mathbf{A}_2$, $\lfloor\frac{m}{2}\rfloor\lceil\frac{n}{2}\rceil$. Similarly, we perform MAs with submatrices of $\mathbf{B}$ and $\mathbf{C}$.

Thus, we find that the problem size $[m, n, p]$ to yield control to the ATLAS/GotoBLAS's algorithm is when Equation 2 is satisfied:

$$\lfloor\frac{m}{2}\rfloor\lceil\frac{n}{2}\rceil\lceil\frac{p}{2}\rceil \leq \frac{\alpha}{2\pi}[5\lceil\frac{n}{2}\rceil(\lceil\frac{m}{2}\rceil + \lceil\frac{p}{2}\rceil) + 3mp]. \quad (2)$$

We assume that π and α are functions of the matrix size only, as we explain in the following.

Layout effects. The performance of MA is not affected by a specific matrix layout (i.e., row-major format or Z-Morton) or shape as long as we can exploit the only viable reuse: spatial data reuse. We know that data reuse (spatial/temporal) is crucial for matrix multiply. In practice, ATLAS and GotoBLAS cope rather well with the effects of a (limited) row/column major format reaching often 90% of peak performance. Thus, we can assume for practical purpose that π and α are functions of the matrix size only.

Square Matrix MM. Combining the performance properties of both matrix multiplications and matrix additions with a more specific analysis for only square matrices; that is, $n = m = p$. We can simplify Equation 2 and we find that the recursion point n_1 is as it follows:

$$n_1 = 22\frac{\alpha}{\pi}. \quad (3)$$

For example, if we assume a ratio α/π=50 (this is common for the systems tested in this work), we find that the recursion point corresponds to the problem (matrix) size $n_1 > 1100$. For problems of size $(\ell)n_1 \leq n < (\ell + 1)n_1$, we may apply Strassen's ℓ times. In fact, the factors π and α are easy to estimate by benchmarking and we can determine the specific recursion point n_1 by a linear search.

4. EXPERIMENTAL RESULTS

We split this experimental section. We present experimental results for rectangular matrices and square matrices separately. For rectangular matrices, we investigate the effects of the problem sizes and shapes w.r.t. the algorithm choice and the architecture. For square matrices, we show how effective our approach is for a large set of different architectures and odd matrix sizes, and we show detailed performance for a representative set of architectures.

4.1 Rectangular matrices

Given a matrix multiply $\mathbf{C} = \mathbf{AB}$ with $\sigma(\mathbf{A})$=$m\times n$ and $\sigma(\mathbf{B})$=$n\times p$, we characterize this problem size by a triplet $\mathbf{s} = [m, n, p]$. For presentation purpose, we opted to represent one matrix multiplication by its number of operations x=$2\prod_{i=0}^{2}\mathbf{s}_i$ and plot it on the abscissa (otherwise the performance plot must be a 4-dimension graph, $[time, m, n, p]$).

4.1.1 *HP zv6000, Athlon-64 2GHz using ATLAS, data locality vs. operations*

In this section, we turn our attention to a general performance comparison of the balanced MM (Table 1) and HASA (Table 2) with respect to the routine *cblas_dgemm* —i.e., ATLAS's MM for matrices in double precision— for dense but rectangular matrices. In Figure 3, we present the relative performance, we then discuss the process used to collect these results and offer an interpretation.

We investigated the input space $\mathbf{s}\in\mathbb{T}\times\mathbb{T}\times\mathbb{T}$ with $\mathbb{T}$={100, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 6000, 8000, 10000, 12000}; however, we ran only the problems that have a working set

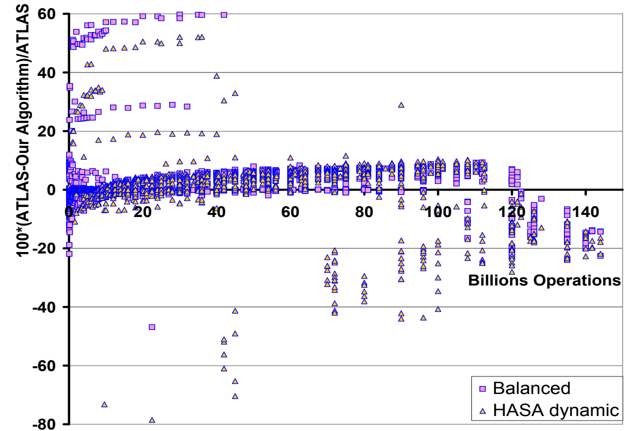

Figure 3: HP ZV6005: Relative performance with respect to ATLAS's *cblas_dgemm*.

(i.e., operands, $\mathbf{C}$, $\mathbf{A}$ and $\mathbf{B}$, and one-recursion-step temporaries, $\mathbf{M}$, $\mathbf{T}_1$ and $\mathbf{T}_2$) smaller than the main-memory size (i.e., 512MB). We present relative performance results for two algorithms with respect to *cblas_dgemm*: *Balanced* and *HASA dynamic*.

Balanced is the algorithm where we determine the recursion point for HASA when we install our codes into this architecture. First, we found the experimental recursion point for square matrices, which is $\dot{n}_1$=1500. We then set the HASA (Table 2) to stop the recursion when at least one matrix size is smaller than 1500.

HASA dynamic is the algorithm where we determine the recursion point for every specific problem size at runtime (no cache-oblivious strategy). To achieve the same performance of the *Balanced* algorithm for square matrices, we set a coefficient ϵ=20 as an additive contribution to the ratio $\frac{\alpha_\mathbf{s}}{\pi_\mathbf{s}} \sim \frac{\alpha_\mathbf{s}}{\pi_\mathbf{s}} + \epsilon$. At run time and for each problem $\mathbf{s}$, we measure the performance of MAs (i.e., $\alpha_\mathbf{s}$) and the performance of *cblas_dgemm* (i.e., $\pi_\mathbf{s}$). We set the recursion point as the problem size satisfying $2\lfloor\frac{m}{2}\rfloor\lceil\frac{n}{2}\rceil\lceil\frac{p}{2}\rceil \leq (\frac{\alpha_\mathbf{s}}{\pi_\mathbf{s}} + \epsilon)[5\lceil\frac{n}{2}\rceil(\lceil\frac{m}{2}\rceil + \lceil\frac{p}{2}\rceil) + 3mp]$.

Performance observations. ATLAS's peak performance is 3.52 GFLOPS; Balanced and HASA dynamic peak performance is normalized to 3.8 GFLOPS (i.e., $2 * m * n * p/ExecutionTime$, it is an overestimate of the actual number of operations per second but it is still a valid comparison measure). The erratic performance of both algorithms (i.e., from 50% speedup to 20% slowdown and especially for small sizes and very large sizes w.r.t. ATLAS's *cblas_dgemm*) is not a problem of the recursion point determination, which may cause either the recursion to improve performance unexpectedly or the recursion overhead to *chocke* the execution time. The main reason for the *sudden* speedups is the *poor* performance of *cblas_dgemm*; the main reason for the slowdowns is the access of the hard-disk memory space by our algorithm.

We conclude that Strassen's algorithm can be applied successfully to rectangular matrices and we can achieve significant improvements in performance in doing so. However, the improvements are often the result of a better data-locality utilization than just a reduction of operations. For example, *HASA dynamic* algorithm is bound to have fewer floating point operations than *Balanced*, because the former apply Strassen's division more times than the latter especially for non-square matrices; however, *Balanced* algorithm achieves on average 1.3% execution-time reduction instead *HASA dynamic* achieves on average 0.5%. [3] In general,

[3]The input set has mostly small problems, thus the average time

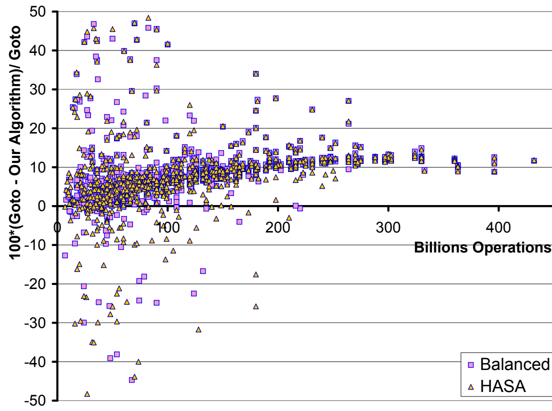

Figure 4: Optiplex GX280, Pentium 4 3.2GHz, using GotoBLAS's DGEMM.

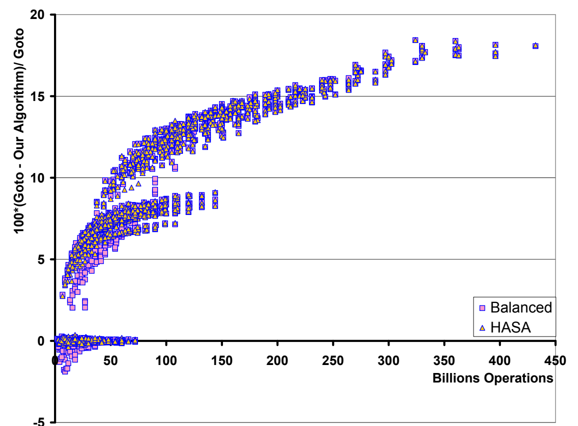

Figure 5: Altura 939, Athlon 64 2.5GHz, using GotoBLAS's DGEMM.

Balanced presents very predictable performance with often better peak performance than *HASA dynamic*.

4.1.2 GotoBLAS, Strassen vs. Faster MM

Recently, GotoBLAS MM has replaced ATLAS MM and the former has become the fastest MM for most of state-of-the-art architectures. In this section, we show that our algorithm is almost unaffected by the change of the leaf implementation (i.e., when the recursion yields to ATLAS/GotoBLAS), leading to comparable and even better improvements.

We investigated the input space $\mathbf{s}\in\mathbb{T}\times\mathbb{T}\times\mathbb{T}$ with $\mathbb{T}=\{1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 6000\}$. We present relative performance results for two algorithms with respect to GotoBLAS *DGEMM* (resp. *Balanced* and *HASA*) and for two architectures Athlon 64 2.4GHz and Pentium 4 3.2 GHz. Balanced and HASA are tuned offline (i.e., recursion point determined once for all).

Optiplex GX280, Pentium 4 3.2GHz using GotoBLAS. GotoBLAS peak performance is 5.5 GFLOPS; Balanced and HASA dynamic peak performance is normalized to 6.2 GFLOPS (for comparison purpose). For this architecture, the recursion point is empirically found at $n_1 = 1000$ and we stop the recursion when a matrix size is smaller than n_1. By construction, HASA algorithm has fewer instructions than Balanced because it applies Strasssen's division to larger matrices; however, HASA achieves smaller relative improvement w.r.t. GotoBLAS MM than what the Balance algorithm achieves (on average Balanced achieves 7.2% speedup and HASA achieves 5.8%), see Figure 4. This suggests that the algorithm with better data locality delivers better performance than the algorithm with fewer operations (for this architecture).

Notice that the performance plot has an erratic behavior, which is similar to the previous scenario (i.e., Figure 3); however, in this case, the performance is affected by the architecture as we show in the following using the same library but a different system.

Altura 939, Athlon-64 2.5GHz using GotoBLAS. GotoBLAS peak performance is 4.5 GFLOPS, Balanced and HASA dynamic peak performance is normalized to 5.4 GFLOPS (as comparison). For this architecture (with a faster memory hierarchy and processor than in Section 4.1), the recursion point is empirically found at $n_1 = 900$ and we stop the recursion when a matrix size is smaller than n_1. Both algorithms Balanced and HASA have similar performance; that is, on average HASA achieves 7.7% speedup and Balanced achieves 7.5% speedup. Also, for this architectures, the performance is very predictable and the performance plot shows the levels of recursion applied clearly, see Figure 5 (clearly 3 levels and a fourth for very large problems).

reduction is biased towards small values.

4.2 Square Matrices

For square matrices, we measure only the performance of HASA because the Balanced algorithm will call directly HASA. For each machine, we had three basic kernels: the C-code implementation *cblas_dgemm* of MM in double precision from ATLAS, hybrid ATLAS/Strassen algorithm (HASA) —in Table 2 where the leaf computation is the *cblas_dgemm*— and our hand-coded MA, which is tailored to each architecture. In Table 3, we present a summary of the experimental results (for rectangular matrices as well) but, in this section, we present details for only four architectures (see [9, 10] for more results).

Installation. We installed the software package ATLAS on every architectures. The ATLAS routine *cblas_dgemm* is the MM we used as reference: we time this routine for each architecture so to determine our baseline and the coefficient π (i.e.,, for square matrices of size 1000), and this routine is also the leaf computation of HASA.

We timed the execution of MA (i.e.,, for square matrices of size 1000) [9, 10] and we determined α. Once we have π and α, we determined the recursion point $n_1{=}22\frac{\pi}{\alpha}$, which we have used to install our codes. We then determined experimentally the recursion point $\dot{n}_1$ (i.e., $n_1{=}22(\frac{\pi}{\alpha}+\epsilon)$) based on a simple linear search.

Performance presentation. We present two measures of performance: the relative execution time of HASA over *cblas_dgemm* and *cblas_dgemm* relative MFLOPS over ideal machine peak performance (i.e., maximum number of multiply-add operations per second). In fact, the execution time is what any final user cares about when comparing two different algorithms. However, a measure of performance for *cblas_dgemm*, such as MFLOPS, shows whether or not HASA improves the performance of a MM kernel that is either efficiently or poorly designed. This basic measure is important in as much as it shows the space for improvement for both *cblas_dgemm* and HASA.

We use the following terminology: HASA is the recursion algorithm for which the recursion point is based on the experimental $\dot{n}_1$ (i.e., $=22(\frac{\pi}{\alpha}+\epsilon)$), which is different for each architecture, and it is statically computed once. The **S-k** is the Strassen's algorithm

Table 3: Systems and performance: $\frac{1}{\pi}10^6$ is the performance of *cblas_dgemm* or DGEMM (GotoBLAS) in MFLOPS for n=1000; $\frac{1}{\alpha}10^6$ is the performance of MA in MFLOPS for n=1000; n_1 is the theoretical recursion point as estimated in $22\frac{\alpha}{\pi}$; instead, $\dot{n}_1$ is the measured recursion point.

System	Processors	$\frac{1}{\pi}10^6$	$\frac{1}{\alpha}10^6$	n_1=$22\frac{\alpha}{\pi}$	$\dot{n}_1$	Figure
Fujitsu HAL 300	SPARC64 100MHz	177	10	390	400	–
RX1600	Itanium 2@1.0GHz	3023	105	487	725	–
ES40	Alpha ev67 4@667MHz	1240	41	665	700	Fig. 7
Altura 939	Athlon 64 2.45MHz	4320	110	860	900	Fig. 5
Optiplex GX280	Pentium 4 3.2GHz	4810	120	900	1000	Fig. 4
RP5470	8600 PA-RISC 550MHz	763	21	772	1175	Fig. 8
Ultra 5	UltraSparc2 300MHz	407	9	984	1225	–
ProLiant DL140	Xeon 2@3.2GHz	2395	53	995	1175	–
ProLiant DL145	Opteron 2@2.2GHz	3888	93	918	1175	Fig. 9
Ultra-250	UltraSparc2 2@300MHz	492	10	1061	1300	–
HP ZV 6005	Athlon 64 2GHz	3520	71	1106	1500	Fig. 3
Sun-Fire-V210	UltraSparc3 1GHz	1140	22	1140	1150	–
Sun Blade	UltraSparc2 500MHz	460	8	1191	1884	–
ASUS	AthlonXP 2800+ 2GHz	2160	39	1218	1300	–
Unknown server	Itanium 2@700MHz	2132	27	1737	2150	–
Fosa	Pentium III 800MHz	420	4	2009	N/A	–
SGI O2	MIPS 12K 300MHz	320	2	2816	N/A	–

for which k is the recursion depth before yielding to *cblas_dgemm*, independently of the problem size. Note that we did not report negative relative performance for S-k because they would have mostly negative bars cluttering the charts and the results. So for clarity, we omitted the correspondent negative bars in the charts. The performance obtained by the systems in Table 3, are obtained by the collection of the best performance among several trials and thus with *hot* caches.

Performance interpretation. In principle, the S-1 algorithm should have about $sp_1 = (1 - \dot{n}_1/n)/8$ relative speedup where $\dot{n}_1$ is the recursion point as found as in Table 3 and n is the problem size (e.g., it is about 7–10% for our set of systems and n=5000). The speedup, for the algorithm with recursion depth ℓ, is additive, $sp = \sum_{i=1}^{\ell} sp_i$; however, each recursion contribution is always less than the first one ($sp_i < sp_1$), because the number of operations saved is decreasing when going down the division process. In the best case, for a three level recursion depth, we should achieve about $3sp_1 \sim 24-30\%$. We achieve such a speedup for at least one architecture, the system ES40, Figure 7 (and similar trend for the Altura system Figure 5). However, for the other architectures, a recursion depth of three is often harmful.

5. 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 in that we use an-easy-to-adapt recursive algorithm using a balanced division process. This division process simplifies the algorithm and it enables us to combine an easy performance model and highly tuned MM kernels so that to determine off-line and at installation what is the best strategy. We have tested extensively the performance of our approach on 17 systems and we have shown that Strassen is not always applicable and, for modern systems, the recursion point is quite large; that is, the problem size where Strassen's start having a performance edge is quite large (i.e., matrix sizes larger than 1000×1000). However, speedups up to 30% are observed over already tuned MM using this hybrid approach.

We conclude by observing that a sound experimentation environment in combination with a simple complexity model —which quantifies the interactions among the kernels of an application and the underlying architecture— can go a long way in helping the design of complex-but-portable codes. Such metrics can improve the design of the algorithms and may serve as a foundation for a fully automated approach.

6. ACKNOWLEDGMENTS

This work was supported in part by DARPA through the Department of Interior grant NBCH1050009.

7. REFERENCES

[1] D. Bailey, K. Lee, and H. Simon. Using strassen's algorithm to accelerate the solution of linear systems. *J. Supercomput.*, 4(4):357–371, 1990.

[2] 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*, pages 419–424. IEEE Computer Society Press, 1988.

[3] 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.

[4] R. P. Brent. Algorithms for matrix multiplication. Technical Report TR-CS-70-157, Stanford University, Mar 1970.

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

[6] S. Chatterjee, A.R. Lebeck, P.K. Patnala, and M. Thottethodi. Recursive array layout and fast parallel matrix multiplication. In *Proc. 11-th ACM SIGPLAN*, June 1999.

[7] H. Cohn, R. Kleinberg, B. Szegedy, and C. Umans. Group-theoretic algorithms for matrix multiplication, Nov 2005.

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

[9] P. D'Alberto and A. Nicolau. Adaptive Strassen and ATLAS's DGEMM: A fast square-matrix multiply for

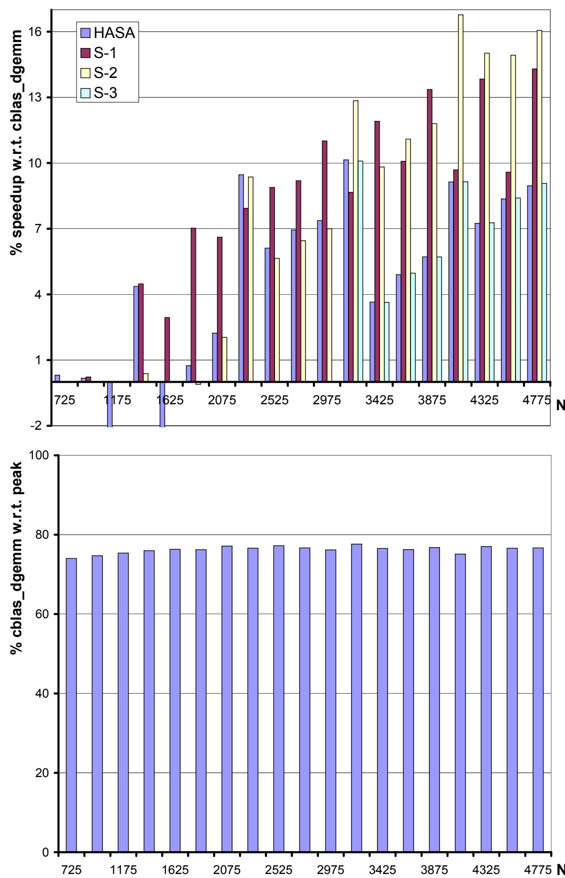

Figure 6: RX1600 Itanium 2@1.0GHz.

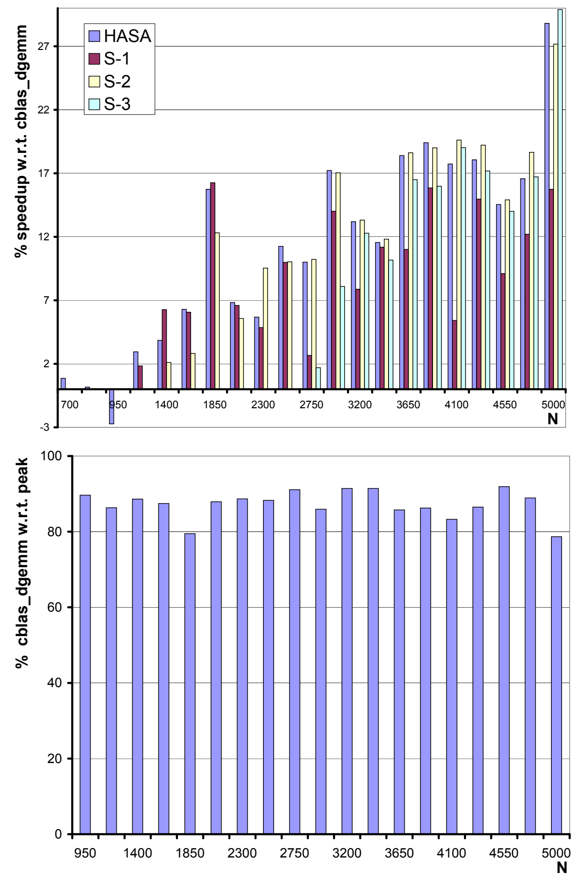

Figure 7: ES40 Alpha ev67 4@667MHz.

modern high-performance systems. In *The 8th International Conference on High Performance Computing in Asia Pacific Region (HPC asia)*, pages 45–52, Beijing, Dec 2005.

[10] P. D'Alberto and A. Nicolau. Using recursion to boost ATLAS's performance. In *The Sixth International Symposium on High Performance Computing (ISHPC-VI)*, 2005.

[11] J. Demmel, J. Dongarra, E. Eijkhout, E. Fuentes, E. Petitet, V. Vuduc, R.C. Whaley, and K. Yelick. Self-Adapting linear algebra algorithms and software. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2), 2005.

[12] J. Demmel, J. Dumitriu, O. Holtz, and R. Kleinberg. Fast matrix multiplication is stable, Mar 2006.

[13] J. Demmel and N. Higham. Stability of block algorithms with fast level-3 BLAS. *ACM Transactions on Mathematical Software*, 18(3):274–291, 1992.

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

[15] J.D. Frens and D.S. Wise. Auto-Blocking matrix-multiplication or tracking BLAS3 performance from source code. *Proc. 1997 ACM Symp. on Principles and Practice of Parallel Programming*, 32(7):206–216, July 1997.

[16] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):216–231, 2005.

[17] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache oblivious algorithms. In *Proceedings 40th Annual Symposium on Foundations of Computer Science*, 1999.

[18] K. Goto and R.A. van de Geijn. Anatomy of high-performance matrix multiplication. ACM Transactions on Mathematical Software.

[19] J.A. Gunnels, F.G. Gustavson, G.M. Henry, and R.A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[20] N.J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Trans. Math. Softw.*, 16(4):352–368, 1990.

[21] N.J. Higham. *Accuracy and Stability of Numerical Algorithms, Second Edition*. SIAM, 2002.

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

[23] I. Kaporin. A practical algorithm for faster matrix multiplication. *Numerical Linear Algebra with Applications*, 6(8):687–700, 1999. Centre for Supercomputer and Massively Parallel Applications, Computing Centre of the Russian Academy of Sciences, Vavilova 40, Moscow

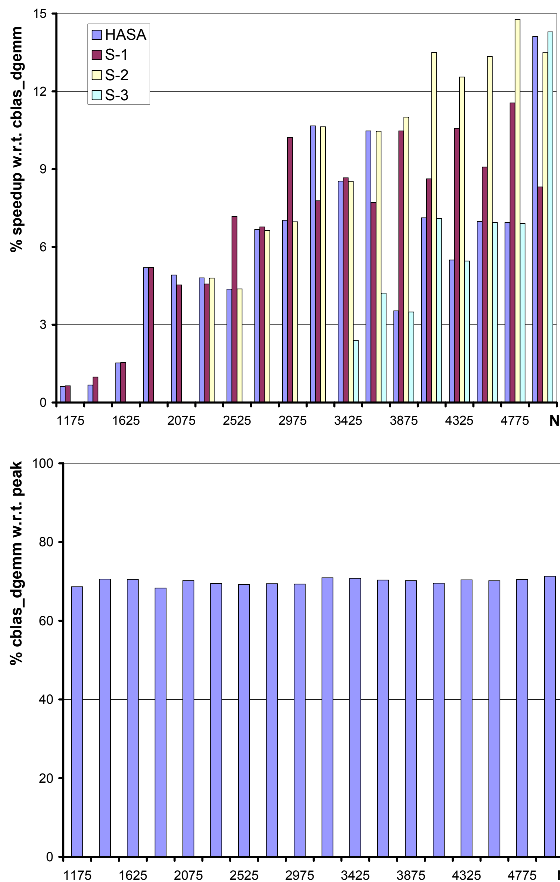

Figure 8: RP5470 8600 PA-RISC 550MHz.

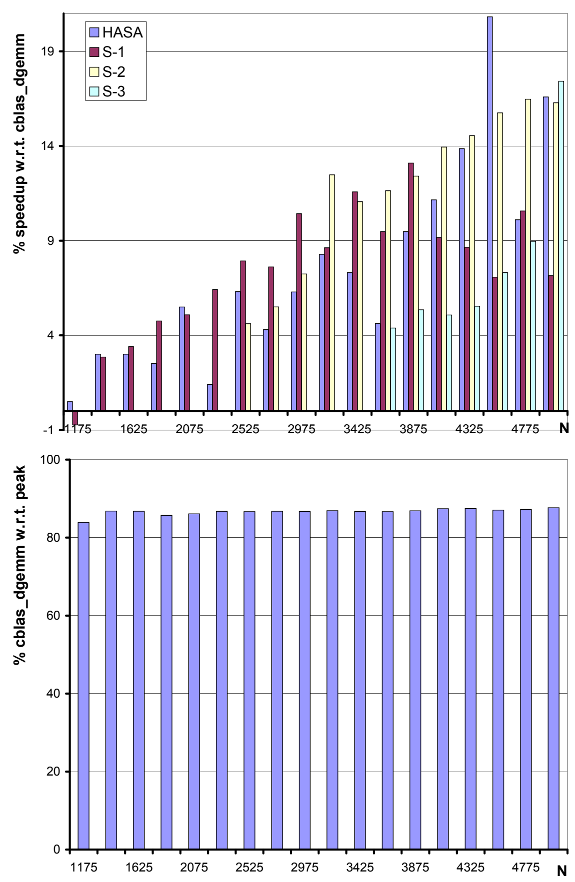

Figure 9: ProLiant DL145 Opteron 2@2.2GHz.

117967, Russia.

[24] Igor Kaporin. The aggregation and cancellation techniques as a practical tool for faster matrix multiplication. *Theor. Comput. Sci.*, 315(2-3):469–510, 2004.

[25] P. Knight. Fast rectangular matrix multiplication and QR-decomposition. *Linear algebra and its applications*, 221:69–81, 1995.

[26] X. Li, M. Garzaran, and D. Padua. Optimizing sorting with genetic algorithms. In *In In Proc. of the Int. Symp. on Code Generation and Optimization*, pages 99–110, March 2005.

[27] V. Pan. Strassen's algorithm is not optimal: Trililnear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *FOCS*, pages 166–176, 1978.

[28] V. Pan. How can we speed up matrix multiplication? *SIAM Review*, 26(3):393–415, 1984.

[29] D. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic (Arith-10)*, pages 132–144, Grenoble, France, 1991. IEEE Computer Society Press, Los Alamitos, CA.

[30] 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"*, 93(2), 2005.

[31] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

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

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