Adaptive Winograd's Matrix Multiplications

PAOLO D'ALBERTO
Yahoo! Inc.
and
ALEXANDRU NICOLAU
Department of Computer Science, University of California Irvine

Modern architectures have complex memory hierarchies and increasing parallelism (e.g., multi-cores). These features make achieving and maintaining good performance across rapidly changing architectures increasingly difficult. Performance has become a complex trade-off, not just a simple matter of counting cost of simple CPU operations.

We present a novel, hybrid, and adaptive recursive Strassen-Winograd's matrix multiplication (MM) that uses automatically tuned linear algebra software (ATLAS) or GotoBLAS. Our algorithm applies to any size and shape matrices stored in either row or column major layout (in double-precision in this work) and thus is efficiently applicable to both C and FORTRAN implementations. In addition, our algorithm divides the computation into equivalent in-complexity sub-MMs and does not require any extra computation to combine the intermediary sub-MM results.

We achieve up to 22% execution-time reduction versus GotoBLAS/ATLAS alone for a single core system and up to 19% for a 2 dual-core processor system. Most importantly, even for small matrices such as 1500×1500, our approach attains already 10% execution-time reduction and, for MM of matrices larger than 3000×3000, it delivers performance that would correspond, for a classic $O(n^3)$ algorithm, to faster-than-processor peak performance (i.e., our algorithm delivers the equivalent of 5 GFLOPS performance on a system with 4.4 GFLOPS peak performance and where GotoBLAS achieves only 4 GFLOPS). This is a result of the savings in operations (and thus FLOPS). Therefore, our algorithm is faster than any *classic* MM algorithms could ever be for matrices of this size. Furthermore, we present experimental evidence based on established methodologies found in the literature that our algorithm is, for a family of matrices, as accurate as the classic algorithms.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Algorithm design and analysis; D.2.2 [**Design Tools and Techniques**]: Library; D.2.8 [**Metrics**]: Performance Measure

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Winograd's matrix multiplications,fast algorithms

1. INTRODUCTION

Our main interest is the design and implementation of highly portable codes; that is, codes that automatically adapt to the architecture evolution. We want to write efficient and easy to maintain codes, which can be used for several generations of architectures. **Adaptive**

Author's address: Paolo D'Alberto *pdalbert@yahoo-inc.com*; Alexandru Nicolau *nicolau@ics.uci.edu*.

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., [Frigo and Johnson 2005; Demmel et al. 2005; Püschel et al. 2005; Gunnels et al. 2001]). In this paper, we discuss a single but fundamental algorithm in dense linear algebra: **matrix multiply** (MM). We propose an algorithm that automatically adapts to any architecture and applies to any size and shape matrices stored in double precision and in either row or column-major layout (i.e., our algorithm is suitable for both C and FORTRAN, algorithms using row-major order [Frens and Wise 1997; Eiron et al. 1998; Whaley and Dongarra 1998; Bilardi et al. 2001], and using column-major order [Higham 1990; Whaley and Dongarra 1998; Goto and van de Geijn 2008]).

In practice, software packages such as LAPACK [Anderson et al. 1995] are based on a basic routine set such as the basic linear algebra subprograms **BLAS 3** [Lawson et al. 1979; Dongarra et al. 1990b; 1990a; Blackford et al. 2002], which, in turn, can be based on efficient implementations of the MM kernel [Kagstrom et al. 1998a; 1998b]. ATLAS [Whaley and Dongarra 1998; Whaley and Petitet 2005; Demmel et al. 2005] (successor of PHiPAC [Bilmes et al. 1997]) has been a leading example of an adaptive software package implementing BLAS, by automatically adapting codes for many architectures around a highly tuned MM kernel. Recently however, GotoBLAS [Goto and van de Geijn 2008] is offering consistently better performance than ATLAS, because of a careful code organization that utilizes optimally the TLB coupled with hand tuned kernels written directly in assembly.

In this paper, we show how, when, and where a hybrid adaptive implementation of Strassen-Winograd's algorithm [Strassen 1969; Douglas et al. 1994] improves the performance of the best available adaptive matrix multiply (e.g., ATLAS or GotoBLAS). We do this by using a novel algorithm and a simple installation process so as to adjust the algorithm to modern architectures and systems automatically. In this paper, we extend some of the concepts introduced in our previous work [D'Alberto and Nicolau 2005a; 2005b] related to the original Strassen's algorithm. In particular, in this paper we generalize Strassen-Winograd's original MM algorithm (Winograd) to apply to any problem sizes and shapes similarly to the approach by [Douglas et al. 1994] but without their **dynamic overlapping** (i.e., conceptually overlapping one row or column, computing the results for the overlapped row or column in both subproblems, and ignoring one of the copies) and thus fewer operations and cleaner formulation. We also propose a *balanced* division process that assures a constant but lower operation count than previously proposed versions, exploits better data locality, and ultimately outperforms any implementation based on the classic algorithm of complexity $O(n^3)$ (we expand this comparison in Section 2); these modifications are critical (especially the balancing) to the extension of the algorithm to multicore systems, which we also present in this paper.

The results we present include performance on single-core and multicore processors that are becoming common in state-of-the-art machines and everyday laptops. We present experimental results for 3 systems (two uni-processor systems and one multi–core-multiprocessor system) where we tested our codes.

2. RELATED WORK

Strassen's algorithm [Strassen 1969] is the first and the most widely used among the fast algorithms for MM. In this paper we use the term *fast algorithms* to refer to the algorithms that have asymptotic complexity less than $O(N^3)$, and we use the term *classic* or *conventional* algorithms for those that have complexity $O(N^3)$. Strassen discovered that the classic recursive MM 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 and $O(N^2)$). These MAs make the algorithm faster, however they make it weakly numerically stable and not unstable [Higham 2002]. As the starting point for our hybrid adaptive algorithm we use Winograd's algorithm (e.g., [Douglas et al. 1994]), which requires only 15 MAs. Thus, Winograd's algorithm has, like the original by Strassen, asymptotic operation count $O(n^{2.81})$, but it has a smaller constant factor and thus fewer operations than Strassen's algorithm.

The asymptotically fastest algorithm to date is by Coppersmith and Winograd $O(n^{2.376})$ [Coppersmith and Winograd 1987]. This has a theoretical contribution, but it is not practical for common problem sizes. Pan showed a bi-linear algorithm that is asymptotically faster than Strassen-Winograd [Pan 1978] $O(n^{2.79})$ and he presented a survey of the topic [Pan 1984] with best asymptotic complexity of $O(n^{2.49})$. The practical implementation of Pan's algorithm $O(n^{2.79})$ is presented by Kaporin [Kaporin 1999; 2004]. For the range of problem sizes presented in this work, the asymptotic complexity of Winograd's and Pan's is similar, however Kaporin implementation requires padding of matrices such that the algorithm exploits specific matrix sizes (for the best implementation matrices should be aligned to $n = 48$).

Recently, new, group-theoretic algorithms that have complexity $O(n^{2.41})$ [Cohn et al. 2005] have been proposed. These algorithms are numerically stable [Demmel et al. 2006] because they are based on the Discrete Fourier Transform (DFT) kernel computation. However, there have not been any experimental quantification of the benefits of such approaches.

In practice, for small matrices, Winograd's MM has a significant overhead and classic MMs are more appealing. To overcome this, several authors have proposed *hybrid* algorithms; that is, deploying Strassen/Winograd's MM in conjunction with classic MM [Brent 1970b; 1970a; Higham 1990], where for a specific problem size n_1, or **recursion point** [Huss-Lederman et al. 1996], Strassen/Winograd's algorithm yields the computation to the classic MM implementations. [1] Our approach has three advantages versus previous approaches:

(1) Our algorithm works for any matrix size and shape and it is a single algorithm, independently of the matrix sizes, and that contains *no conditional branches*. In practice, our algorithm requires only 43 lines of C code (i.e., *including* declarations, initialization and de-allocation of local variables, and thus it is simple to understand and to maintain). This implementation, because it has no conditional branches, offers an easier means to investigate different scheduling optimizations/organizations without control-flow dependency (see the appendix on page 20).

(2) Our algorithm divides the MM problems into a set of *balanced* subproblems; that is, with minimum difference of operation count (i.e., complexity) between subproblems. This balanced division leads to: a cleaner algorithm formulation (and a simpler/shorter code), easier parallelization and more efficient parallel execution (i.e., because the

[1] Thus, for a problem of size $n \leq n_1$, this hybrid algorithm uses the classic 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.

parallel subproblems are balanced, the workload between processors is balanced) and little or no work in combining the solutions of the subproblems, and thus fewer operations (w.r.t. algorithms applying *peeling*, more obliviously, *padding* [Panda et al. 1999], where the problem size is artificially increased, or data re-organization by using recursive layout at run time).

This balanced division strategy differs from the division process proposed by Huss-Lederman et al. [Huss-Lederman et al. 1996; Huss-Lederman et al. 1996; Higham 1990], where the division is a function of the problem size. In fact, for odd-matrix sizes, they divide the problem into a large even-size problem (*peeling*), on which Strassen's algorithm can be applied, and a small, and extremely irregular, computation. This computation tail exploits little data locality and, even if for a constant factor, in practice this affects negatively the operation count and the overall performance.

(3) At every recursive step, we use only 3 temporary matrices, which is the minimum number possible [Douglas et al. 1994]. Furthermore, we differ from Douglas et al. work in that we do not perform redundant computations for odd-size matrices.

We store matrices in standard row/column-major format and, at any time, we can yield control to a highly tuned MM such as ATLAS/GotoBLAS $DGEMM$ without any overhead. Such an overhead would be incurred while changing to/from different data layout and it has been often neglected in previous performance evaluations. [Chatterjee et al. 2002; Thottethodi et al. 1998] estimated such overheads as 5–10% of the total execution time. Furthermore, because we use the standard layout for our matrices throughout the process, if faster implementations of BLAS emerge (or other alternatives appear), we can always integrate these in our hybrid algorithm with no (further) modifications, a major practical advantage.

While for large multi-processors our algorithm can be further optimized to yield even better results, such work is beyond the scope of the current paper, that aims to present our fundamental algorithm and demonstrate how it can yield significant improvements over the current state-of-the-art for some of the most widely used modern high-performance processors. In this work, we present a parallel implementation that uses fast algorithms only at processor level and for few cores/processors. This is in contrast with previous algorithms by Grayson et al. [Grayson et al. 1995] and more recently for machine clusters [Ohtaki et al. 2004; Nguyen et al. 2005].

In fact, in this paper we do not claim a general parallel algorithm. We present an algorithm designed for standalone desktop parallel systems with one or a few powerful processors deploying multicore technology (i.e., the vast majority of state-of-the-art desktops available today). We show how in these systems the algorithm proposed adapts and scales maintaining superior performance because of a scalable approach where the major speed up is the result of faster computation at the core level.

3. FAST MULTIPLICATION ALGORITHMS

For the description of our algorithms, we postopone the description of our parallel algorithm to Section 5.3, where we divide the problem among processors and cores, yielding our parallel algorithm; in Section 3.2, we describe how to reduce the operation count so as to have fast algorithms for a single core. However, here, we start with some basic notations and definitions.

3.1 Matrix Multiplication: Definition and Notations

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 $\mathbf{C}=\mathbf{AB}$ (i.e., we omit the symbol $*$), where the component $c_{i,j}$ at row i and column j of the result matrix $\mathbf{C}$ is defined as $c_{i,j} = \sum_{k=0}^{n-1} a_{i,k}b_{k,j}$.

We use a simplified notation to identify submatrices. We choose to divide logically a matrix $\mathbf{M}$ into 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 the $\mathbf{M}_0$, and $\mathbf{M}_3$ is beneath $\mathbf{M}_1$ and to the right of $\mathbf{M}_2$.

The computation is divided into four parts, one for each submatrix composing $\mathbf{C}$. Thus, for every matrix $\mathbf{C}_i$ $(0 \le i \le 3)$, the classic approach computes two products, using a total of 8 MMs and 4 MAs —notice that the 4 MAs are computed in combination with 4 MMs and require no further passes through the data. Notice that every product computes a result that has the same size and shape as the destination submatrix $\mathbf{C}_i$. If we decide to compute the products recursively, each product $\mathbf{A}_i\mathbf{B}_j$ is divided further into four subproblems, and the computation in Equation 1 applies unchanged to these subproblems.

$$\begin{bmatrix} \mathbf{C}_0 & \mathbf{C}_1 \\ \mathbf{C}_2 & \mathbf{C}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_1 \\ \mathbf{A}_2 & \mathbf{A}_3 \end{bmatrix} \begin{bmatrix} \mathbf{B}_0 & \mathbf{B}_1 \\ \mathbf{B}_2 & \mathbf{B}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{A}_0\mathbf{B}_0+\mathbf{A}_1\mathbf{B}_2 & \mathbf{A}_0\mathbf{B}_1+\mathbf{A}_1\mathbf{B}_3 \\ \mathbf{A}_2\mathbf{B}_0+\mathbf{A}_3\mathbf{B}_2 & \mathbf{A}_2\mathbf{B}_1+\mathbf{A}_3\mathbf{B}_3 \end{bmatrix}, \qquad (1)$$

3.2 Adaptive Winograd's Matrix Multiply

The combination of our MA and our adaptation of the original Winograd's algorithm permits a cleaner implementation. As result, our algorithm derives always a balanced subproblem division independently of the problem size and thus a consistent performance across problem sizes, see the pseudo code in Algorithm 1 and the C-code implementation in Figure 9 Appendix 6.

To extend Winograd's algorithm to non-square matrices, we have to face the possibility of adding un-even size matrices. A trivial extension of the definition of matrix addition is the following: we simply add, element-wise, corresponding elements up to the size of the smaller matrix, and fill the rest of the result matrix with the remaining elements of the larger matrix [D'Alberto and Nicolau 2005a] (see for a simple implementation Figure 10 in the appendix).

The schedule of the operations is derived from the schedule proposed by Thottethodi et al. [Thottethodi et al. 1998]; this requires one MA and one temporary more than the schedule proposed by Douglas et al. [Douglas et al. 1994] (in the best case), because we do not use the result matrix $\mathbf{C}$ as temporary matrix for the first MM (we use the temporary matrix $\mathbf{U}_2$. However, this schedule is applied for the multiply-add matrix operations (i.e., $\mathbf{C}{+}{=}\mathbf{AB}$), for which we cannot use the result matrix as temporary space, and, in this case, we perform the minimum number of MAs and we use the minimum number of temporary matrices. Furthermore, the ability to combine the MA with the MM speeds up the overall computation.

Notice that the matrix $\mathbf{U}_2$ is used to exploit common expressions (as Winograd's proposed so as to reduce the number of MAs) and the matrix is used not as temporary for matrix additions (as matrices $\mathbf{S}$ and $\mathbf{T}$) but for the accumulation of matrix products. In fact at the end of the computation, $\mathbf{U}_2$ summarizes the result of three MMs: $M_1 = \mathbf{A}_0\mathbf{B}_0$,

Algorithm 1 : Adaptive Winograd's MM: $\mathbf{C}{=}\mathbf{A}*_w\mathbf{B}$ with $\sigma(\mathbf{A}){=}m{\times}n$ and $\sigma(\mathbf{B}){=}n{\times}p$

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

$M_1 + M_2$ with $M_2 = (\mathbf{A}_2 + \mathbf{A}_3 + \mathbf{A}_0)(\mathbf{B}_3 - \mathbf{B}_1 - \mathbf{B}_0)$, and $M_1 + M_2 + M_3$ with $M_3 = (\mathbf{A}_0 + \mathbf{A}_2)(\mathbf{B}_3 + \mathbf{B}_1)$.

In practice, a matrix copy is memory bound and thus it takes approximately as much time as a matrix addition, and we count matrix copies as MAs. Thus, this algorithm performs 7 MM, 18 MA, and it requires three temporary matrices (i.e., $\mathbf{S}$, $\mathbf{T}$, and $\mathbf{U}_2$) at every recursion step. [2]

Our contributions are: first, we present an extensive/detailed experimental data and comparisons of performance (Section 5), in particular when it comes to choose the leaf computation kernel of the Winograd's algorithm, the kernel performance for relatively small matrices is the most important factor (e.g., $N = 1000$) and not the best asymptotic performance (e.g., $N = 3000$). Second, we provide a quantitative evaluation of the numerical stability of our algorithm and a comparison with other implementations such as in GotoBLAS, ATLAS or Strassen's algorithm (Section 5.4). Third, we extend our algorithm to

[2]Notice that the temporary matrix $\mathbf{S}$ is used to store MAs involving only submatrices of $\mathbf{A}$, $\mathbf{T}$ is used to store MAs involving only submatrices of $\mathbf{T}$, and $\mathbf{U}_2$ for $\mathbf{C}$.

deal with the important multicore systems now emerging (Section 5.3).

4. ALGORITHM INSTALLATION AND EXPERIMENTAL SETUP

To make our algorithm self-installing we proceed as follows. For every machine, we installed both GotoBLAS (Ver. 1.6.0) and ATLAS (Ver. 3.7.1). The installation time is minimal because these libraries have been configured already. We installed our codes in conjunction with these libraries. Each hybrid version uses either ATLAS or GotoBLAS for the leaf computations in our version of Winograd's algorithm, so we have two implementations that we identify as follows: **W-Goto** for the hybrid adaptive Winograd using GotoBLAS; and **W-ATLAS** for the hybrid adaptive Winograd using **ATLAS**. For conciseness, we identify the pure GotoBLAS MM as simply **Goto** and pure ATLAS MM as **ATLAS**.

Our setting-up process follows these steps:

(1) **Recursion point estimation.** First, we determine the execution time T_{mm} of GotoBLAS and ATLAS MM for matrices of size 1000×1000, which is in practice a problem size where MM does not fit in the caches of the surveyed machines, the MM performance reaches the architecture limits and where Winograd's algorithm could start being beneficial. We compute $\pi = 2 * 1000^3 / T_{mm}$ (the actual MM floating point per second FLOPS, which is usually varies only slightly for problem larger than 1000×1000). Then, we measure the execution time T_{ma}, which is the estimate overhead due to MA for Winograd's algorithm, of MA for matrices 1000×1000 and we compute $\alpha = 1000^2 / T_{ma}$ (which will have negligible variations, and thus can be approximated as an experimentally derived-constant for problems larger than 1000×1000). As an approximation, we use the formula $n_1 \geq 22\frac{\pi}{\alpha}$ [D'Alberto and Nicolau 2005a] to estimate the recursion point. That is, the point (matrix size) when the execution time of 22 MAs of matrices of size $n_1 \times n_1$ (i.e., $\frac{22}{\alpha}$ seconds) is equal to the execution time of one MM (i.e., $\frac{1}{\pi}$ seconds) the time we save if we use Strassen/Winograd's algorithm. This is the matrix size when Strassen/Winograd yields control to **GotoBLAS/ATLAS**.

(2) **Search and evaluation.** Empirically, we perform a linear search starting from $n = 22\frac{\pi}{\alpha}$ (thus reducing the search space): we increment n (size of a square matrix $n \times n$) until the execution time of **Goto** or **ATLAS** is slower than **W-Goto/W-ATLAS** with one level of recursion always applied. We find the practical recursion point $\dot{n}_1$. In practice, $n_1 > 22\frac{\pi}{\alpha}$ (even for Winograd's algorithm requiring only 18 MAs) because the term $22\frac{\pi}{\alpha}$ accounts for the MAs and 7 MMs performance contributions in the Winograd's algorithm in isolation, instead, in the implementation, the MAs disrupt locality of the 7 MMs and the time saved is practically less than the time of a single MM in isolation; thus, we achieve the performance balance only for larger problem sizes (n_1). The recursion point is determined at this stage and used at run time. [3]

(3) **Code installation.** We compile and install the hybrid adaptive **W-Goto** and **W-ATLAS** codes, where they yield control to **Goto** and **ATLAS** respectively, for problems such that one matrix operand size is smaller than the practical recursion point

[3]The authors in [Huss-Lederman et al. 1996] and ourselves [D'Alberto and Nicolau 2007] investigated the relation between problem sizes and recursion point in general and dynamically at run time. This requires a run time adaptation and this is beyond the scope of this work.

$\dot{n}_1$. The compiler used in this work is *gcc* with optimization flags *-O2 -Wall -march=* -mtune=* -msse2*. The code is available on-line *http://www.ics.uci.edu/~fastmm/* or email *fastmm@ics.uci.edu*.

4.1 Measurement methodology

We select a set of problem sizes representing square and rectangular MMs. For example, 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 $\mathbf{s} = [m, n, p]$. We investigate the input space $\mathbf{s} \in \mathbb{T} \times \mathbb{T} \times \mathbb{T}$ with $\mathbb{T} = \{500\ 1000\ 2000\ 3000\ 4000\ 5000\ 6000\}$ (i.e., $\mathbf{A} * \mathbf{B}$ of size $\sigma(\mathbf{A}) = m \times n$ and $\sigma(\mathbf{B}) = n \times p$ with $m, n, p \in \mathbb{T}$). Given the input set, we measure the execution times. Naturally, this would be a 4-dimensional plot, because the problem is specified by $\mathbf{s}$ and its MM($\mathbf{s}$) performance. We present all 2-dimensional plots where the problem is specified by the number of operations $2mnp$. Thus, differently shaped matrices will have the same number of operations and thus the same value in the abscissa. However, they can have different performance. [4]

We chose to present two performance measures: normalized GFLOPS and relative time.

Normalized GFLOPS. The complexity of Winograd's algorithm has asymptotic complexity $O(n^{2.81})$ operations and the classic algorithm has $2n^3$. In practice for our hybrid algorithm, the number of operations depends on how many times the algorithm recursively divides the problem, which is a function of the problem size, architecture, and performance of the leaf MM. For both the classic algorithm (i.e., **Goto/ATLAS**) and our algorithm (i.e, **W-Goto/W-ATLAS**), we set the normalized GFLOPS (Giga Floating Point Operations per second) performance as $(2mnp/Time)/10^9$ where $Time$ is the execution time of the MM under examination (e.g., $Time_{Goto}$ execution time of **Goto** or $Time_{W-Goto}$ execution time of **W-Goto**). The advantage of using such a normalized performance is threefold: first, we can plot clearly the performance of very small and very large problems in the same chart; second, this measure maintains the execution-time order among the algorithms (e.g., higher normalized GFLOPS means faster time and *vice versa*); the normalized performance for **Goto/ATLAS** specifies the distance to reach the architecture throughput or peak performance (i.e., operation per second usually available in the processor/machine manual). However, the normalized GFLOPS performance overestimates the GFLOPS of our algorithms, because the actual number of floating-point operations is less than $2mnp$.

Relative Time. Given a reference algorithm, for example **Goto**, we determine the relative time reduction by our algorithm, for example **W-Goto**, as $100 * (Time_{Goto} - Time_{W-Goto})/Time_{Goto}$. The best relative improvement is 100 and the minimum is $-\infty$. This measure makes crystal clear the performance advantage of our algorithm; however, such a measure must be used in combination with the normalized GFLOPS performance in order to emphasize that we can improve an algorithm that already achieves peak performance and thus its performance limits.

5. PERFORMANCE EVALUATION

In this section, we present experimental results for our hybrid algorithms and we present three important aspects of our code performance. First, our hybrid adaptive Winograd algorithms are faster than both the best GotoBLAS and ATLAS MM. For problems larger

[4]GotoBLAS is relatively un-affected by the matrix shape

than 3000×3000, our algorithm is faster than every DGEMM implementations. Our algorithm's maximal performance cannot be matched by any classical GEMM implementation, because such a GEMM would exceed the theoretical peak FLOP rate of the machine. Second, even though **ATLAS** on its own is slower than **Goto** on its own, nevertheless, for one Opteron system our algorithm deploys ATLAS MM to achieve the best performance (because for matrices of sizes 1000×1000, which is what our leaf computation uses, ATLAS provides better performance than the GotoBLAS). Third, we extend and apply a scalable hybrid algorithm for a common multicore multi-processor desktop and show our performance advantage.

In Table I, we present the three machines we used and the minimum problems size when Winograd's algorithm is profitable. The HP xw9300 is a multicore system and each processor can be used separately (Section 5.2) or together (Section 5.3) and thus having different recursion points. In Table II, we summarize the performance and relative improvements for each processor and, thus two configurations are related to the multicore system HP xw9300. In the following subsections, we discuss the results in detail.

Table I. Systems and recursion points: π is the performance of *DGEMM* on matrices of size 1000×1000 in MFLOPS; α is the performance of MA in MFLOPS; n_1 is the theoretical recursion point as estimated in $22\frac{\pi}{\alpha}$; instead, $\dot{n}_1$ is the measured recursion point.

System	**Processors**	π	α	$n_1 = 22\frac{\pi}{\alpha}$	$\dot{n}_1$	**Figure**
HP xw9300	Opteron 2.2GHz	3680	104	810	950	Fig. 3
-	Opteron 4 cores 2.2GHz	-	-	-	1300	Fig. 5
Altura 939	Athlon64 2.45GHz	4320	110	860	900	Fig. 2
Optiplex GX280	Pentium 4 3.2GHz	4810	120	900	1000	Fig. 1

Table II. Processors and performance

Processors	Peak GFLOPS	Best DGEMM GFLOPS	Best Winograd Norm. GFLOPS	Average Relative	Best Relative
Opteron 1@2.2GHz	4.4	4	5	7.39%	22%
Athlon64 2.45GHz	4.9	4.4	5.7	8.33%	23%
Pentium 4 3.2GHz	6.4	5.5	7.1	7.55%	21%
Opteron 4@2.2GHz	17.6	15.6	19.5	11.8%	19%

5.1 W-Goto

In this section, we present evidence that **W-Goto** is faster than the current best implementation (i.e., using only GotoBLAS/ATLAS) and better than any implementation based on the classic matrix multiply (i.e., any future MM implementation of complexity $O(N^3)$). We present experimental results for two architectures commonly used in desktops —i.e., Pentium 4 3.2 GHz and Athlon64 2.45 GHz— and we compare the actual performance of fast algorithms w.r.t. the GotoBLAS DGEMM, which is optimized for these machines and faster than ATLAS alone.

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 $\mathbf{s} = [m, n, p]$. We investigated the input space $\mathbf{s}{\in}\mathbb{T}{\times}\mathbb{T}{\times}\mathbb{T}$ with $\mathbb{T}{=}\{$500 1000 2000 3000 4000 5000 6000$\}$.

Optiplex GX280: Pentium 4 3.2 GHz. This is a single core Pentium 4 3.2 GHz system with 1GHz bus and a stand alone desktop running Kubuntu Linux. For matrices of size $1000{\times}1000$, GotoBLAS MM achieves 4.8 GFLOPS (π) and achieves 5.5 GFLOPS as peak/best performance. For matrices $1000{\times}1000$, matrix addition achieves 120 MFLOPS (α). This suggests that Winograd's algorithm should have a recursion point at about 900 ($22\frac{\pi}{\alpha}$). In practice, the recursion point is at 1000.

W-Goto has on average 7.55% relative time improvement, and achieves 7.10 Normalized GFLOPS best performance (i.e., Normalized GFLOPS is computed as $2mnp/Time$, instead of the effective number of operation of Winograd's algorithm, Section 4.1). This yields an improvement up to 21% for large problems.

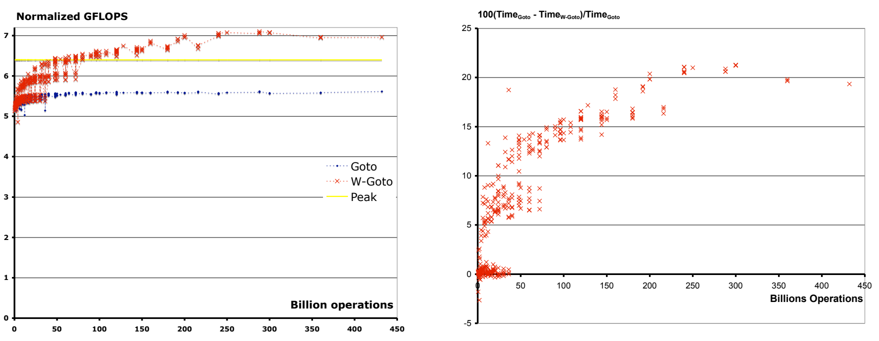

Fig. 1. Pentium 4 3.2GHz: GFLOPS and Relative Performance

In Figure 1, we present the normalized performance of the two algorithms. We present also the relative time saving using **W-Goto**. Notice that the peak performance of this machine is 6.4 GFLOPS and our algorithm can achieve 7.10 Normalized GFLOPS.

Altura 939: Athlon64 2.45 GHz. This is a single core Athlon64 2.45 GHz with a 1 GHz front bus and a stand alone desktop running Kubuntu. For matrix sizes of $1000{\times}1000$, GotoBLAS MM achieves 4.3 GFLOPS and a best performance of 4.46 GFLOPS. For matrices $1000{\times}1000$, matrix addition achieves 110 MFLOPS. This suggests that Winograd's algorithm should have a recursion point at about 860. In practice, the recursion point is at 900.

W-Goto has on average 8.23% relative time improvement, and achieves the best performance of 5.7 Normalized GFLOPS. This yields an improvement up to 23%.

In Figure 2, we present the normalized performance of the two algorithms (**Goto** and **W-Goto**) and the relative time saving using the **W-Goto**. Notice how the **W-Goto** performance is such that no classic matrix multiplication can match our performance, because the peak performance of the system is 5 GFLOPS.

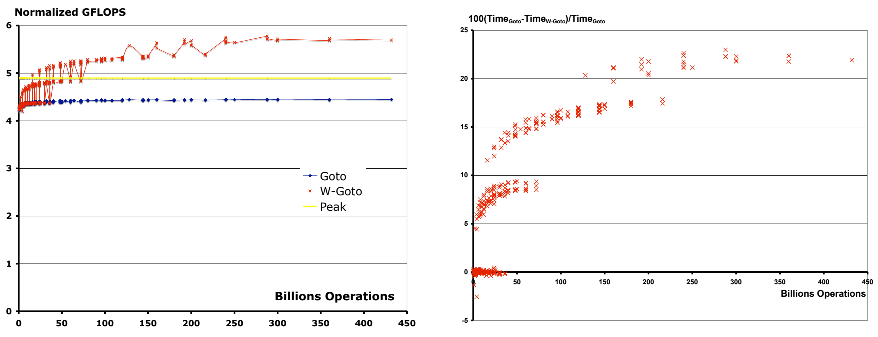

Fig. 2. Athlon64 2.45GHz.

5.2 W-Goto vs. W-ATLAS

In the following, even though GotoBLAS is the fastest conventional algorithm for this system (as it is for the Pentium 4 and slightly faster for the Athlon64), we demonstrate that in our hybrid versions we should use ATLAS MM instead. We investigated the input space $s \in \mathbb{T}\times\mathbb{T}\times\mathbb{T}$ with $\mathbb{T}=\{500\ 1000\ 2000\ 3000\ 4000\ 5000\ 6000\}$.

HP xw9300: 1-Core Opteron 2.2 GHz. This is a 2 dual-core Opteron processor 2.2 GHz system with a 1 GHz front bus and a stand alone desktop running Kubuntu. For matrix sizes of 1000×1000, GotoBLAS MM achieves 3.6 GFLOPS and the best performance obtained by GOTOBLAS on this machine is 4.03 GFLOPS. For matrix sizes of 1000×1000, ATLAS's matrix multiply achieves 3.9 GFLOPS which is also its best performance for this machine. Thus, **ATLAS** achieves better performance for small matrices, however falls behind for larger ones when compared to GotoBLAS.

For matrices 1000×1000, matrix addition MA achieves 104 MFLOPS. This suggests that Strassen's algorithm should have a recursion point at about 810. In practice, the recursion point is at 950.

If we deploy GotoBLAS MM as leaf computation for the fast algorithms, **W-Goto** has on average 4.78% relative time improvement and it achieves 4.83 Normalized GFLOPS. This algorithm has an improvement of up to 16% relative execution time.

However, if we deploy **ATLAS**, **W-ATLAS** has on average 7.39% relative time improvement (w.r.t. Goto) and achieves 5.07 Normalized GFLOPS. This in turn yields an improvement of up to 22% relative execution time.

In Figure 3, we present the normalized performance of the four algorithms: **Goto**, **ATLAS**, **W-ATLAS** and **W-Goto**. **W-ATLAS** performance is such that no classic matrix multiplication can match our performance, because the architecture peak is 4.4 GFLOPS.

5.3 Extension to Multicore processors: 2 Dual-Core Processor

Multicore multiprocessor systems are becoming ubiquitous. They represent small-scale parallel architectures in standalone state-of-the-art desktops. For example, we consider an AMD 2-dualcore processor Opteron 275 system. Each processor has two cores on the same die. A core has a separate memory hierarchy composed of two levels: the first level is composed of a data and an instruction cache (64KB each) and a unified second level (1MB). Inter-processor communication is performed through a dedicated interconnection

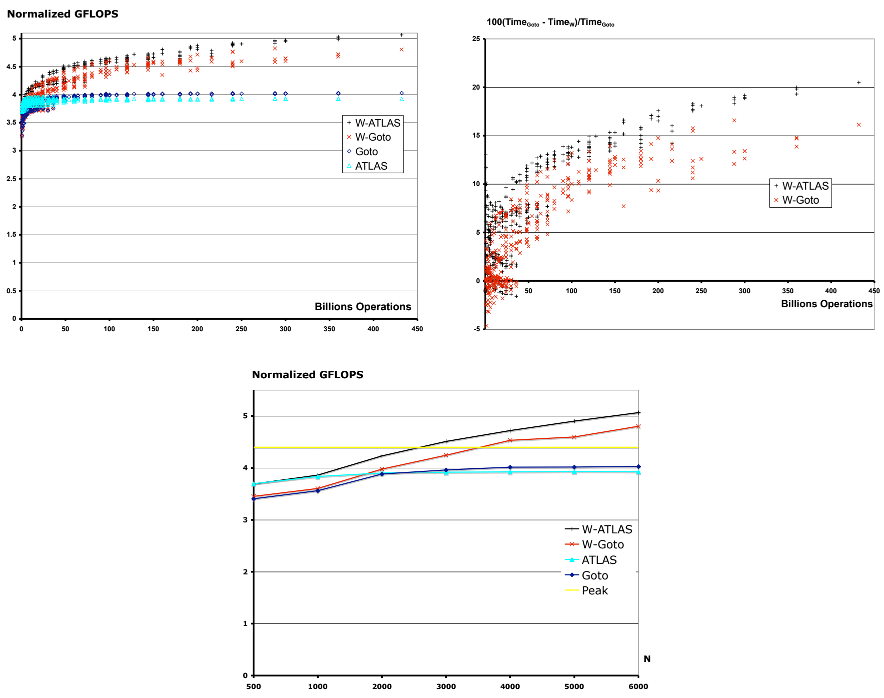

Fig. 3. Opteron 2.2GHz: (top left) Absolute performance for rectangular matrices, (top right) relative performance for rectangular matrices, and (bottom) absolute performance for square matrices.

directly from the cores. Memory-core connection is separate and the memory is up to 2 GByte (for this system).

We present a parallel algorithm (Figure 4) that scales up relative well for multicore architectures. The parallel algorithm employs the hierarchical division process expressed in Equation 1. The algorithm divides the problem in four subproblems, thus it allocates a balanced work to each core and distributes data so as to optimize both the data communication among processors (i.e., minimize communication) and the common data among cores (i.e., exploit local memory and caches). If more processors and cores are available, we can recursively divide each subproblem and perform a similar allocation. In practice, it is not arbitrarily scalable as for large numbers of processors the inter-processor/core bandwidth, and data distribution will be a significant bottleneck and thus decrease performance. Rather, this is a natural extension of our basic algorithm, that performs very well for current limited parallelism state-of-the-art multicores.

The parallel-algorithm description. We start with one basic task (or process): the *mother* of all tasks. The mother starts two tasks: T_0 and T_1. Mother moves T_0 to processor P_0 and T_1 to P_1. Each processor has two cores. The operands are distributed as follows: $\mathbf{C}_0$, $\mathbf{C}_1$, $\mathbf{A}_0$, $\mathbf{A}_1$, $\mathbf{B}$ are allocated within T_0 thus processed using processor P_0; $\mathbf{C}_2$, $\mathbf{C}_3$, $\mathbf{A}_2$, $\mathbf{A}_3$, and $\mathbf{B}$ are allocated within T_1, thus processed by P_1. Notice that $\mathbf{B}$ is duplicated in both processors.

The data is allocated at this time. That is, T_0 and T_1 make an explicit call to *malloc()* (or

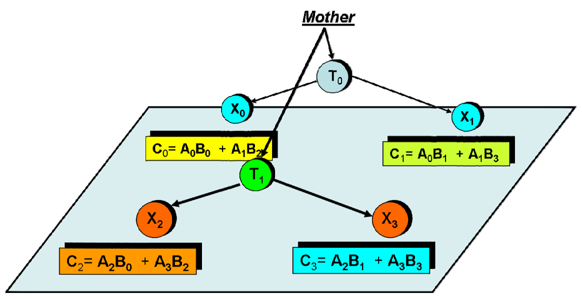

Fig. 4. Hierarchical Parallel Algorithm

cmalloc()). While *malloc()* does not initialize the data it assures the data association to the processor. Thus, data will be stored into the memory closest to the processor.

T_0 spawns two tasks X_0 and X_1 that share the same virtual space. X_0 is associated with CPU_0, and X_1 with CPU_1. T_1 spawns two tasks as well X_2 and X_3 (CPU_2, CPU_3). X_0 is responsible to compute $\mathbf{C}_0 = \mathbf{A}_0\mathbf{B}_0 + \mathbf{A}_1\mathbf{B}_2$, and X_1 is responsible to compute $\mathbf{C}_1 = \mathbf{A}_0\mathbf{B}_1 + \mathbf{A}_1\mathbf{B}_3$ (similar for X_2 and X_3).

In such a scenario, tasks in different processors do not communicate. Both CPUs in each processor have tasks that share the same memory space and the same data of $\mathbf{A}$ and $\mathbf{B}$ and compute a basic computation such as $\mathbf{C}_0 = \mathbf{A}_0\mathbf{B}_0 + \mathbf{A}_1\mathbf{B}_2$ once. We shall present the execution time such that starting from the spawn of tasks T_0 and T_1 to a barrier that specifies the end of the main computation of X_0, X_1, X_2, and X_3. In this scenario, the computation of tasks X_i dominates the overall execution time.

Notice that the division process and the data allocation is performed such that we can benefit from the shared memory space without explicit data movement to/from different processors keeping the programming simple and very close to what a sequential algorithm would be.

HP xw9300: 4-Cores Opteron 2.2 GHz. We investigated the input space $\mathbf{s}\in\mathbb{T}\times\mathbb{T}\times\mathbb{T}$ with $\mathbb{T}=\{3000\ 5000\ 6000\ 7000\ 8000\ 10000\ 11000\ 12000\ 13000\ 14000\}$ and we apply this parallel algorithm presented here. That is, the cores will compute MM on matrices of size between 1500×1500 and 7000×7000. At core level (CPU) we adopt **W-ATLAS**. The parallel solution that deploys **ATLAS** only achieves 15.6 GFLOPS performance.

Empirically, the recursion point for **W-ATLAS** to yield to **ATLAS** is at about 1300, which is larger than the single core system (i.e., where it is about 950). To feed two cores during MAs, which are memory bound, implies that memory and interconnection speeds are not fast enough, and we have to adjust our strategy. For our system, the recursion point is taken care of during installation and no further modifications are necessary w.r.t. the single core case.

W-ATLAS has on average 11.8% time reduction and it achieves 19.5 Normalized GFLOPS (i.e., Normalized GFLOPS is computed as $2mnp/Time$, instead of the effective number of operation of Winograd's algorithm, Section 4.1). Thus, we achieve improvements of up to 19% relative to execution time. The parallel solution achieves faster than peak performance, however the asymptotic improvement is smaller than the single core system (i.e., 22% relative execution time improvement). This is due to the larger recursion point (1300 instead of 950) and its effects on the performance of **W-ATLAS** on each core. In principle, if the recursion point for the parallel version would increase even further we should deploy GotoBLAS instead of ATLAS. Due to space limitations we choose to show the graphs for only the better performing (**W-ATLAS** instead of **W-Goto**) codes.

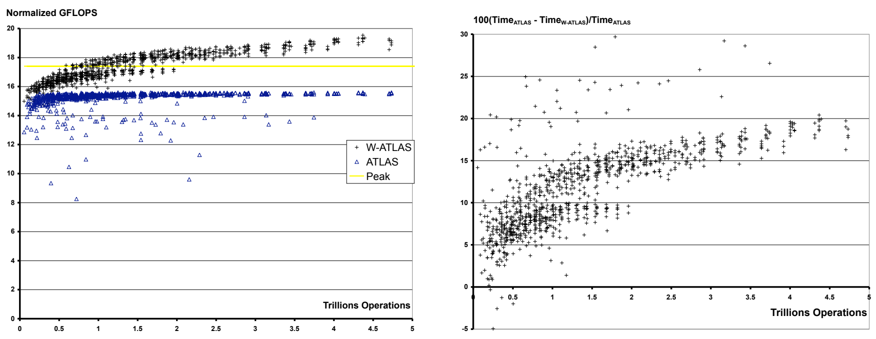

Fig. 5. 2 dual-core processor system: Opteron 2.2GHz

In Figure 5, we present the normalized performance and relative performance of the two parallel algorithms (**ATLAS** and **W-ATLAS**). Notice how the **W-ATLAS** performance is such that no classic matrix multiplication can match our performance, because the peak performance of the system is 17.6 GFLOPS.

5.4 Error evaluation

As an example, fast MM algorithms find application in iterative methods for the solution of equation systems (e.g., in the matrix factorization and determination of the starting solution) where the iterative-algorithm convergence is independent of the starting solution and the natural feedback of the method keeps the error under control. In the literature, there is clear evidence of the *practical* stability of fast algorithms such as Winograd's algorithm [Demmel and Higham 1992; Higham 2002], which are known to be **weakly stable** as we reiterate the definition in the following (Equation 2).

Nevertheless, the stability of fast algorithms is an issue that always raises questions. As a final contribution, we now turn to the study of the stability of our algorithm and, by experimentation, we offer a graphical, quantitative, and practical representation of the numerical stability of our algorithm. We start with the known upper-bound of the numerical error. Then, for classes of matrices, we show how far our algorithm may go from these upper bounds .

An upper bound to the error of Winograd's algorithm is (Theorem 23.3 [Higham 2002]):

$$\|\mathbf{C} - \dot{\mathbf{C}}\| \leq \left[\left(\frac{n}{n_1} \right)^{\log_2 18} (n_1^2 + 6n_1) - 6n \right] u \|\mathbf{A}\| \|\mathbf{B}\| + O(u^2) \tag{2}$$

where $\sigma(\mathbf{A}) = \sigma(\mathbf{B}) = \sigma(\mathbf{C}) = n \times n$, $\|A\| = \max_{ij} |a_{ij}|$, n_1 is the size where Winograd's algorithm yields to the usual MM, $\mathbf{C}$ is the exact output ($\mathbf{C} = \mathbf{A} *_{\mathbf{exact}} \mathbf{B}$) and $\dot{\mathbf{C}}$ is the computed output (using Winograd's algorithm $\dot{\mathbf{C}} = \mathbf{A} *_{\mathbf{w}} \mathbf{B}$), and u is the inherent floating point precision. If we define the recursion depth as ℓ (i.e., the number of times we divide the problem using Winograd's division), this upper bound can be approximated as

$$\|\mathbf{C} - \dot{\mathbf{C}}\| \leq 4.5^{\ell} n^2 u \|\mathbf{A}\|\|\mathbf{B}\| + O(u^2) \tag{3}$$

Similarly, Strassen's algorithm has an upper bound of $3^{\ell} n^2 u \|\mathbf{A}\|\|\mathbf{B}\| + O(u^2)$. In comparison, the forward error of the conventional computation of matrix multiplication is

$$|\mathbf{C} - \dot{\mathbf{C}}| \leq nu|\mathbf{A}||\mathbf{B}| \textit{ and } \|\mathbf{C} - \dot{\mathbf{C}}\| \leq n^2 u \|\mathbf{A}\|\|\mathbf{B}\| \tag{4}$$

That is, the norm-wise error of the Winograd's MM increases by a factor of 4.5^{ℓ} w.r.t. the conventional algorithm (Equation 4 norm-wise bound) as we divide the problem further. For all the architectures and problem sizes we have investigated, ℓ is less than 3; thus for practical purposes, both terms 4.5^{ℓ} and 3^{ℓ} are bound by a constant.

These bounds are tight; that is, there are matrices for which the actual error is close to the bound. [5]

Here, we use the experiments and the approach used by Higham to quantify empirically and illustrate graphically that the error experienced in practice could be far less than the upper bound analysis suggests in Equation 2, which can be extremely pessimistic.

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. This type of input reduces the range of the MM so that $\|\mathbf{AB}\| \leq n\|\mathbf{A}\|\|\mathbf{B}\| \leq n$, and basically, the error bound is a function of only the problem size and it is independent of the matrix values. The same operand values $[-1, 1]$ and $[0, 1]$ are used by Higham and presented in Ch. 23 [Higham 2002]. Notice that probability matrices have range [0,1] and thus they represent a practical case where the upper-bound and quantitative evaluation is not just a speculation. In practice, we could choose matrix operands to make the products $|\mathbf{A}||\mathbf{B}|$ and $\|\mathbf{A}\|\|\mathbf{B}\|$ arbitrarily large, and thus the error arbitrarily large; however, in the same fashion, matrix scaling can be applied to normalize matrices to the range investigated.

Reference DCS. Consider the output $\mathbf{C}$=$\mathbf{AB}$. We compute every element c_{ij} by performing first a *dot* product of the row vector a_{i*} by the column vector b_{*j} and we store it into a vector $\mathbf{z}$. Then, we sort the vector in decreasing order such that $|z_i| \geq |z_j|$ with $i < j$ [Li et al. 2005]. Finally, we compute the reference output using Priest's **doubly compensated summation** (DCS) procedure [Priest 1991] as described in Algorithm 4.3 of [Higham 2002] in double precision. This is our baseline or ultimate reference $\mathbf{C}$ in Equation 2.

Architecture. We consider our adaptive hybrid algorithm for the Opteron based architecture and we use the same architecture to evaluate the error analysis.

Comparison. We compare the output-value difference (w.r.t the DCS based MM) of GotoBLAS algorithm, **W-Goto**, **S-Goto** (Strassen's algorithm using Goto's MM), **ATLAS**, **W-ATLAS**, **S-ATLAS** (Strassen's algorithm using ATLAS's MM), and the classic **row-by-column** algorithm (RBC) (for which the summation is not compensated and the values are not ordered in any way and it is the BLAS FORTRAN reference `www.netlib.org/blas/`).

In Figure 6, 7 and 8, we show the error evaluation w.r.t. the DCS MM for square matrices only, and the results confirm previous published results [Higham 2002].

[5]As there are matrices for which LU factorization with partial pivoting has the elements of factor $U \in \mathbb{R}^{n \times n}$ increasing as 2^n.

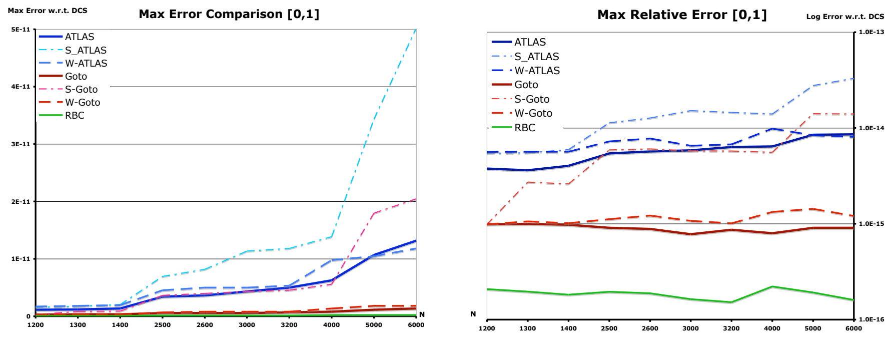

Fig. 6. Opteron 270: Error evaluation matrices in the range $[0, 1]$

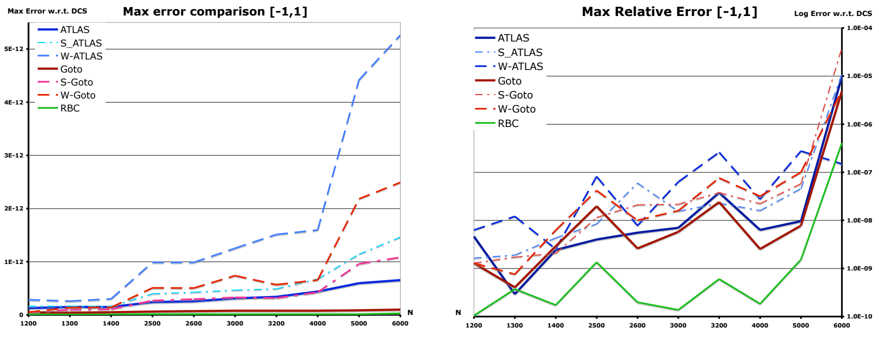

Fig. 7. Opteron 270: Error evaluation matrices in the range $[-1, 1]$

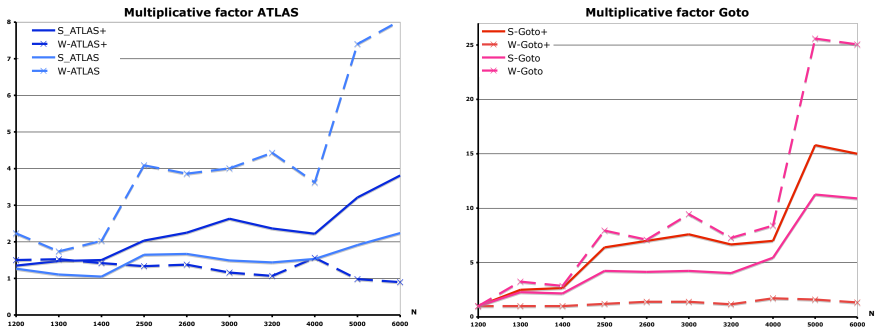

Fig. 8. Opteron 270: Error multiplicative factor. For example, **W-ATLAS+** and **W-Goto+** represent the multiplicative error factors for positive matrices only

As we expected, as the number of recursive calls increases, so does the error. However, the magnitude of the error is small. For Strassen's algorithm the error ratio of **S-Goto** over **Goto** is no larger than 15 (instead of the upper bound $3^3 = 27$) for both ranges

$[0, 1]$ and $[-1, 1]$; that is, we lose only one significant decimal digit of the 16 available. For the range $[-1, 1]$, the error ratio of **W-Goto** over **Goto** is no larger than 25 (instead of $4.5^3 = 91$, loss of $1\frac{1}{4}$ decimal digit instead of almost 2), and for the range $[0, 1]$ the error ratio is no larger than 1.5; that is, we have no practical loss (see Figure 8). Also, the multiplicative error factors for the implementations using ATLAS are even more moderate (less than 8); however, ATLAS based codes have larger maximum and maximum relative error. [6] Nonetheless, these error ratios are less dramatic than what an upper bound analysis suggest.

In our experiments, we have found that for matrices with values in the range $[-1, 1]$, Strassen's algorithm has better accuracy than Winograd's, and for the range $[0, 1]$ the situation is reversed. Previously, [Higham 2002] has shown similar accuracy relationship among Strassen's, Winograd's and the conventional algorithm for power-of-two matrices.

In summary, Winograd's algorithm has empirically comparable stability as that of **Goto** or **ATLAS**, and Strassen's algorithm loses one digit (out of 16) of precision, making both our hybrid algorithms usable in many applications and arbitrary sizes.

6. CONCLUSIONS

In this paper, we present a novel Winograd's hybrid variant of Strassen's fast matrix multiply for arbitrarily shaped matrices. We demonstrate the performance of this algorithm for single and multi core processors and we show the minimum problem size for which our algorithm is beneficial. We present evidence that for matrices larger than 3000×3000 our hybrid Winograd algorithm achieves performance that no *classic* algorithm will match.

Our hybrid version of Winograd's algorithm is weakly stable and it is not (in practice) unstable. It is also faster than previous hybrids, it is applicable to irregular shapes and sizes in either row or column major order, and it is ultimately simpler to implement/understand. In the literature, several authors have *justified* the use of fast algorithms in combination with the classic algorithm (e.g., [Higham 2002]). We show that when the problem is not *ill conditioned* the error introduced by our algorithm is under control and the weak stability of the algorithm should not be used for *a priori* deterrent against its use. In line with [Demmel and Higham 1992], we conclude that the algorithm we propose is viable in most applications as the error introduced will be too small to matter.

In the appendix, we present an excerpt of our codes of Winograd's algorithm and matrix addition, but the code is available on line, please send email to *fastmm@ics.uci.edu* or to look up experimental results for complex matrices visit us at *http://www.ics.uci.edu/~fastmm*.

Acknowledgments. The first author worked on this project during his post-doctorate fellowship in the SPIRAL Project at the Department of Electric and Computer Engineering in the Carnegie Mellon University and his work was supported in part by DARPA through the Department of Interior grant NBCH1050009.

We would like to acknowledge the reviewers and the editors for their helpful, clear, detailed, and very well formulated suggestions that ultimately improved this manuscript.

[6]Probably, because Goto's DGEMMs use a larger tiling —i.e., tailored for the L2 cache— than ATLAS —i.e., tailored for L1 cache— thus exploiting more reuse at register level and exploiting the 90-bit extended precision of the MSSE register file further.

REFERENCES

ANDERSON, E., BAI, Z., BISCHOF, C., DONGARRA, J. D. J., DUCROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., OSTROUCHOV, S., AND SORENSEN, D. 1995. *LAPACK User' Guide, Release 2.0*, 2 ed. SIAM.

BILARDI, G., D'ALBERTO, P., AND NICOLAU, A. 2001. Fractal matrix multiplication: a case study on portability of cache performance. In *Workshop on Algorithm Engineering 2001*. Aarhus, Denmark.

BILMES, J., ASANOVIC, K., CHIN, C., AND DEMMEL, J. 1997. Optimizing matrix multiply using PHiPAC: a portable, high-performance, Ansi C coding methodology. In *Proceedings of the annual International Conference on Supercomputing*.

BLACKFORD, L. S., DEMMEL, J., DONGARRA, J., DUFF, I., HAMMARLING, S., HENRY, G., HEROUX, M., KAUFMAN, L., LUMSDAINE, A., PETITET, A., POZO, R., REMINGTON, K., AND WHALEY, R. C. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathemathical Software 28,* 2, 135–151.

BRENT, R. P. 1970a. Algorithms for matrix multiplication. Tech. Rep. TR-CS-70-157, Stanford University. Mar.

BRENT, R. P. 1970b. Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity. *Numerische Mathematik 16*, 145–156.

CHATTERJEE, S., R., A., PATNALA, P., AND THOTTETHODI, M. 2002. Recursive array layouts and fast matrix multiplication. *IEEE Transactions on Parallel Distributed Systems 13,* 11, 1105–1123.

COHN, H., KLEINBERG, R., SZEGEDY, B., AND UMANS, C. 2005. Group-theoretic algorithms for matrix multiplication.

COPPERSMITH, D. AND WINOGRAD, S. 1987. Matrix multiplication via arithmetic progressions. In *Proceedings of the 19th annual ACM conference on Theory of computing*. 1–6.

D'ALBERTO, P. AND NICOLAU, A. 2005a. Adaptive Strassen and ATLAS's DGEMM: A fast square-matrix multiply for modern high-performance systems. In *The 8th International Conference on High Performance Computing in Asia Pacific Region (HPC asia)*. Beijing, 45–52.

D'ALBERTO, P. AND NICOLAU, A. 2005b. Using recursion to boost ATLAS's performance. In *The Sixth International Symposium on High Performance Computing (ISHPC-VI)*.

D'ALBERTO, P. AND NICOLAU, A. 2007. Adaptive Strassen's matrix multiplication. In *Proceedings of the 21st annual international conference on Supercomputing*. ACM, New York, NY, USA, 284–292.

DEMMEL, J., DONGARRA, J., EIJKHOUT, E., FUENTES, E., PETITET, E., VUDUC, V., WHALEY, R., AND YELICK, K. 2005. Self-Adapting linear algebra algorithms and software. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93,* 2.

DEMMEL, J., DUMITRIU, J., HOLTZ, O., AND KLEINBERG, R. 2006. Fast matrix multiplication is stable.

DEMMEL, J. AND HIGHAM, N. 1992. Stability of block algorithms with fast level-3 BLAS. *ACM Transactions on Mathematical Software 18,* 3, 274–291.

DONGARRA, J. J., CROZ, J. D., DUFF, I. S., , AND HAMMARLING, S. 1990a. Algorithm 679: A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathemathical Software 16*, 18–28.

DONGARRA, J. J., CROZ, J. D., DUFF, I. S., , AND HAMMARLING, S. 1990b. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathemathical Software 16*, 1–17.

DOUGLAS, C., HEROUX, M., SLISHMAN, G., AND SMITH, R. 1994. GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix–matrix multiply algorithm. *J. Comp. Phys. 110*, 1–10.

EIRON, N., RODEH, M., AND STEINWARTS, I. 1998. Matrix multiplication: a case study of algorithm engineering. In *Proceedings WAE'98*. Saarbrucken, Germany.

FRENS, J. AND WISE, D. 1997. Auto-Blocking matrix-multiplication or tracking BLAS3 performance from source code. *Proc. 1997 ACM Symp. on Principles and Practice of Parallel Programming 32,* 7 (Jul.), 206–216.

FRIGO, M. AND JOHNSON, S. 2005. The design and implementation of FFTW3. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93,* 2, 216–231.

GOTO, K. AND VAN DE GEIJN, R. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software 34,* 3, 1–25.

GRAYSON, B., SHAH, A. P., AND VAN DE GEIJN, R. 1995. A high performance parallel Strassen implementation. Tech. Rep. CS-TR-95-24. 1,.

GUNNELS, J., GUSTAVSON, F., HENRY, G., AND VAN DE GEIJN, R. 2001. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software 27,* 4 (Dec.), 422–455.

HIGHAM, N. 1990. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Transactions on Mathematical Software 16,* 4, 352–368.

HIGHAM, N. 2002. *Accuracy and Stability of Numerical Algorithms, Second Edition*. SIAM.

HUSS-LEDERMAN, S., JACOBSON, E., JOHNSON, J., TSAO, A., AND TURNBULL, T. 1996. Strassen's algorithm for matrix multiplication: Modeling analysis, and implementation. Tech. Rep. CCS-TR-96-14, Center for Computing Sciences.

HUSS-LEDERMAN, S., JACOBSON, E., TSAO, A., TURNBULL, T., AND JOHNSON, J. 1996. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*. ACM Press, 32.

KAGSTROM, B., LING, P., AND VAN LOAN, C. 1998a. Algorithm 784: GEMM-based level 3 BLAS: portability and optimization issues. *ACM Transactions on Mathematical Software 24,* 3 (Sept), 303–316.

KAGSTROM, B., LING, P., AND VAN LOAN, C. 1998b. GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software 24,* 3 (Sept), 268–302.

KAPORIN, I. 1999. A practical algorithm for faster matrix multiplication. *Numerical Linear Algebra with Applications 6,* 8, 687–700. Centre for Supercomputer and Massively Parallel Applications, Computing Centre of the Russian Academy of Sciences, Vavilova 40, Moscow 117967, Russia.

KAPORIN, I. 2004. The aggregation and cancellation techniques as a practical tool for faster matrix multiplication. *Theoretical Computuer Science 315,* 2-3, 469–510.

LAWSON, C. L., HANSON, R. J., KINCAID, D., AND KROGH, F. T. 1979. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Transactions on Mathemathical Software 5*, 308–323.

LI, X., GARZARAN, M., AND PADUA, D. 2005. Optimizing sorting with genetic algorithms. In *In Proceedings of the International Symposium on Code Generation and Optimization*. 99–110.

NGUYEN, D., I.LAVALLEE, M.BUI, AND Q.HA. 2005. A general scalable implementation of fast matrix multiplication algorithms on distributed memory computers. In *Proceedings Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks*. 116–122. http://doi.ieeecomputersociety.org/10.1109/SNPD-SAWN.2005.2.

OHTAKI, Y., TAKAHASHI, D., BOKU, T., AND SATO, M. 2004. Parallel implementation of Strassen's matrix multiplication algorithm for heterogeneous clusters. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. 112. http://doi.ieeecomputersociety.org/10.1109/IPDPS.2004.1303066.

PAN, V. 1978. Strassen's algorithm is not optimal: Trililnear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *Foundation of Computer Science*. 166–176.

PAN, V. 1984. How can we speed up matrix multiplication? *SIAM Review 26,* 3, 393–415.

PANDA, P., NAKAMURA, H., DUTT, N., AND NICOLAU, A. 1999. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers 48,* 2, 142–149.

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

PÜSCHEL, M., MOURA, J., JOHNSON, J., PADUA, D., VELOSO, M., SINGER, B., XIONG, J., FRANCHETTI, F., GAČIĆ, A., VORONENKO, Y., CHEN, K., JOHNSON, R., AND RIZZOLO, N. 2005. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93,* 2.

STRASSEN, V. 1969. Gaussian elimination is not optimal. *Numerische Mathematik 14,* 3, 354–356.

THOTTETHODI, M., CHATTERJEE, S., AND LEBECK, A. 1998. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of the 1998 ACM/IEEE conference Supercomputing*. Orlando, FL.

WHALEY, R. AND DONGARRA, J. 1998. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, 1–27.

WHALEY, R. C. AND PETITET, A. 2005. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience 35,* 2 (Feb.), 101–121. http://www.cs.utsa.edu/~whaley/papers/spercw04.ps.

Appendix: Code

```
/***********************************
 C = a.C+(k.A)*(j.B) | a.C+(k.tran{A})*(j.tran{B}) | aC+(k.conj{A})*(j.conj{B})
 */
int wmadd(DEF(c), DEF(a), DEF(b)) {

  if (a.m<= LEAF || a.n<= LEAF || b.n<=LEAF) {
    fastCPU();
    CMC(USE(c), = , USE(a),  mm_leaf_computation_madd , USE(b));
    slowCPU();
  }
  else {
    Matrix tc0 = Q0(c),tc1 = Q1(c), tc2 =Q2(c),tc3=Q3(c);
    Matrix ta0 = Q0(a),ta1 = Q1(a), ta2 =Q2(a),ta3=Q3(a);
    Matrix tb0 = Q0(b),tb1 = Q1(b), tb2 =Q2(b),tb3=Q3(b);

    // temporary
    Matrix s =   {0, S0(a.m,a.n),S0(a.m,a.n),a.trans,a.beta};
    Matrix t =   {0, S0(b.m,b.n),S0(b.m,b.n),b.trans,b.beta};
    Matrix u2  = {0, S0(c.m,c.n),S0(c.m,c.n),c.trans,1};

    // temporary allocation
    s.data   = (Mat *) CALLOC(s);     t.data = (Mat *) CALLOC(t);
    u2.data  = (Mat *) CALLOC(u2);    assert(s.data && t.data  && u2.data);

    /* S  = A2 + A3 */   CMC(RQ2(s,a),  =, ta2,      s_add,   ta3);
    /* T  = B1 - B0 */   CMC(t ,        =, tb1,      s_sub,   tb0);
    /* U2 = S  * T  */   CMC(RQ2(u2,c) ,=, RQ2(s,a), wm ,     t);

    /* *C3 +=  U2 */     CMC(tc3 ,=, tc3, s_add_t ,   RQ3(u2,c)); tc3.beta = 1;
    /* *C1 +=  U2 */     CMC(tc1 ,=, tc1, s_add_t ,   RQ1(u2,c)); tc1.beta = 1;

    /* U2 = A0  * B0 */  CMC (u2, =, ta0, wm,      tb0);
    /* C0 += U2 + C0 */  CMC(tc0, =, tc0, s_add_t, u2); tc0.beta = 1;

    /* C0 += A1 * B2 */  CMC(tc0, =, ta1, wmadd,   tb2);

    /* S  = S  - A0 */   CMC(s,   =, RQ2(s,a), s_sub,  ta0);
    /* T  = B3 - T  */   CMC(t,   =, tb3,      s_sub,  t);

    /* U2 += S * T  */   CMC(u2,  =, s,   wmadd,  t);
    /* C1 += U2,    */   CMC(tc1, =, tc1, s_add,  RQ1(u2,c));

    /* S  =  A1 - S */   CMC(s,   =, ta1,       s_sub,    s);
    /* C1 += S * B3 */   CMC(tc1, =, RQ1(s,a),  wmadd,    tb3);

    /* T  =  B2 - T */   CMC(RQ2(t,b), =, tb2,    s_sub,  RQ2(t,b));
    /* C2 += A3 * T */   CMC(tc2,      =, ta3,    wmadd,  RQ2(t,b)); tc2.beta = 1;

    /* S  = A0 - A2 */   CMC(s,         =, ta0,       s_sub,   ta2);
    /* T  = B3 - B1 */   CMC(RQ1(t,b),  =, tb3,       s_sub,   tb1);
    /* U2 += S*T    */   CMC(RQ1(u2,c), =, s,         wmadd,   RQ1(t,b));
    /* C3 += U2     */   CMC(tc3,       =, tc3,       s_add,   RQ3(u2,c));
    /* C2 += U2     */   CMC(tc2,       =, RQ2(u2,c), s_add,   tc2);

    // free temporaries
    FREE(s.data);    FREE(t.data);    FREE(u2.data);
  }
  return 1;
}
```

Fig. 9. Winograd MM code

```
// C = A + B
void add_t(DEF(c), DEF(a), DEF(b)) {

  int i,j,x,y;

   /* minimum sizes */
   x = min(a.m,b.m); y = min(a.n,b.n);

   for (i=0; i<x; i++) {
     /* core of the computation */
     for (j=0;j<y;j++)
       E_(c.data,i,j,c.M,c.N) = a.beta*E_(a.data,i,j,a.M,a.N)
                              + b.beta*E_(b.data,i,j,b.M,b.N);

     if (y<a.n)  /* A is larger than B */
       E_(c.data,i,j,c.M,c.N) =  a.beta*E_(a.data,i,j,a.M,a.N) ;
     else
       if (y<b.n) /* B is larger than A */
         E_(c.data,i,j,c.M,c.N) = b.beta*E_(b.data,i,j,b.M,b.N);
   }
   /* last row */
   if (x<a.m)  {/* A is taller than B */
     for (j=0;j<a.n;j++)
       E_(c.data,i,j,c.M,c.N)  = a.beta*E_(a.data,i,j,a.M,a.N);
   }
   if (x<b.m)  {/* B is taller than A */
     for (j=0;j<b.n;j++)
       E_(c.data,i,j,c.M,c.N) = b.beta*E_(b.data,i,j,b.M,b.N);
   }
   //   c.beta = 1;
}
```

Fig. 10. Matrix addition code