-

Exploiting Parallelism in Matrix-Computation Kernels for Symmetric Multiprocessor Systems

Matrix-Multiplication and Matrix-Addition Algorithm Optimizations by Software Pipelining and Threads Allocation

Paolo D'Alberto, Yahoo! Sunnivale, CA, USA
Marco Bodrato, University of Rome II, Tor Vergata, Italy
Alexandru Nicolau, University of California at Irvine, USA

We present a simple and efficient methodology for the development, tuning, and installation of matrix algorithms such as the hybrid Strassen's and Winograd's fast matrix multiply or their combination with the 3M algorithm for complex matrices (i.e., hybrid: a recursive algorithm as Strassen's until a highly tuned BLAS matrix multiplication allows performance advantages). We investigate how modern symmetric multiprocessor (SMP) architectures present old and new challenges that can be addressed by the combination of an algorithm design with careful and natural parallelism exploitation at the function level (optimizations) such as function-call parallelism, function percolation, and function software pipelining.

We have three contributions: first, we present a performance overview for double and double complex precision matrices for state-of-the-art SMP systems; second, we introduce new algorithm implementations: a variant of the 3M algorithm and two new different schedules of Winograd's matrix multiplication (achieving up to 20% speed up w.r.t. regular matrix multiplication). About the latter Winograd's algorithms: one is designed to minimize the number of matrix additions and the other to minimize the computation latency of matrix additions; third, we apply software pipelining and threads allocation to all the algorithms and we show how that yields up to 10% further performance improvements.

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*

Additional Key Words and Phrases: Matrix Multiplications, Fast Algorithms, Software Pipeline, Parallelism

ACM Reference Format:
D'Alberto, P., Bodrato, M., and Nicolau, A. 2011. Exploiting Parallelism in Matrix-Computation Kernels for Symmetric Multiprocessor Systems. ACM TOMS -, -, Article - (2011), 30 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Multicore multiprocessor blades are the most common computational and storage nodes in data centers, multi-nodes are becoming the trend (a blade supporting one or two boards), and mixed systems based on the Cell processor are topping the fastest-supercomputers list.

In this work, we experiment with *general purpose* processors; this type of node is often used in search-engine data centers, it can reach up to 200 Giga FLOPS (i.e., depending on the number of processors and CPU's frequency; for example, we

Email the authors: Paolo D'Alberto *pdalbert@yahoo-inc.com*, Marco Bodrato *bodrato@mail.dm.unipi.it*, Alexandru Nicolau *nicolau@ics.uci.edu*, for request a copy of the code *fastmm@ics.uci.edu*.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

will present a system providing 150 GFLOPS in practice in single precision and 75 GFLOPS in double precision), and such nodes can (and do) serve as the basis for quite complex parallel architectures. These machines implement a shared memory system so that developers may use the PRAM model as an abstraction (e.g., [JáJá 1992]). That is, developers think and design algorithms as a collaboration among tasks without explicit communications by using the memory for data sharing and data communications.

In this work, we investigate the implementation features and performance caveats of fast matrix-multiplication algorithms: the hybrid Strassen's, Winograd's, and 3M matrix multiplications algorithms (the last is for complex matrices). For example, the hybrid Winograd's matrix multiplication is a recursive algorithm deploying the Winograd's algorithm using 15 matrix addition in place of a (recursive) matrix multiplication until —the size of the operand matrices are small enough so that— we can deploy a highly tuned BLAS matrix multiplication implementation so that to achieve the best performance. These algorithms are a means for a fast implementation of matrix multiplication (MM). Thus, we present optimizations at algorithmic level, thread allocation level, function scheduling level, register allocation level, and instruction scheduling level so as to improve these MM algorithms, which are basic kernels in matrix computations. However, we look at them as an example of matrix-computations on top of the matrix algebra (*, +). As such, we think the concepts introduced and applied in this work are of more general interests. Our novel contributions are:

(1) We present new algorithms that are necessary for the next contributions and optimizations (the following optimization in the paper). Here, we present optimizations that when applied to the-state-of-the-art algorithms $\mathcal{A}$ provide a speed up of, let us say, 2–5%. To have full effect of our optimizations we need to formulate a new family of algorithms $\mathcal{B}$, though these algorithms are slower than the ones in $\mathcal{A}$. When we apply our optimizations in a parallel architecture to the algorithms in $\mathcal{B}$, we may achieve even better performance, for example, 10–15% w.r.t. $\mathcal{A}$. During the discovery of the algorithms in $\mathcal{B}$, we have found and present another class of algorithms $\mathcal{C}$ that are faster than $\mathcal{A}$ (up to 2% faster) but when we apply our optimizations these algorithms achieve a speed up of *only* 5–7% w.r.t. $\mathcal{A}$.
(2) We show that these fast algorithms are compelling for a large set of SMP systems: they are fast, simple to code and maintain.
(3) We show that fast algorithms offer a much richer scenario and more opportunities for optimizations than the classical MM implementation. Actually, we show that we have further space for improvements (10–15%) making fast algorithms even faster than we have already shown in previous implementations. In practice, at the fast-algorithms core is the trade off between MMs and matrix additions MAs (trading one MM for a few MAs). In this paper, we propose a new approach to scheduling, and thread allocation for MA and MM in the context of fast MM, and we explore and measure the advantages of these new techniques. As a side effect, we show that thanks to parallelism we can speed up the original Strassen's algorithm such that it can be as fast as the Winograd's formulation and we can implement the resulting algorithm in such a way as to achieve the ideal speed up w.r.t. highly tune implementations of MM. That is, We have found an operation schedule that hides the MAs latency completely; thus, we reduce or nullify the performance influence of the number of MAs and, thus, achieving the ideal speedups that Winograd/Strassen algorithms could achieve.
In Figure 1, we show graphically a summary of the best possible speed up we can achieve using Winograd's algorithms and our optimizations (GotoBLAS DGEMM is the reference).

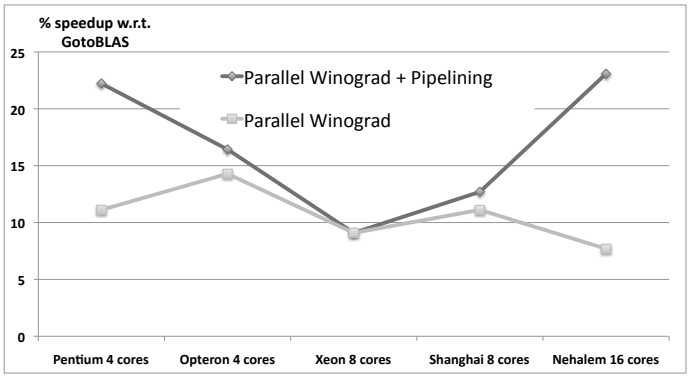

Fig. 1. Summary: speed up w.r.t. GotoBLAS double precision for a few systems. GotoBLAS has speed up 1

(4) We believe this work could be of interest for compiler designers and developers from other fields. At first, our contribution seems limited to fast algorithms, but in practice, what we propose is applied to matrix computations build on the matrix algebra (*,+) composed of parallel basic MM (*) and MA (+) subroutines (i.e., matrix computations are simply a sequence of * and + operations and the optimizations we propose could be applied to any such schedules).

In practice, we present a family of fast algorithms that can achieve an ideal speed up w.r.t. the state-of-the-art algorithms such as the ones available in GotoBLAS and ATLAS library (i.e., one recursion level of Strasswe/Winograd algorithm will be 8/7 faster than GotoBLAS GEMM). The introduction of our optimizations and algorithms provide speed up that other implementations of Winograd's algorithm —based only on algorithmic improvements without taking in account of the architecture— cannot match nor outperform. For any machine where a fast implementation (not using our techniques) is available, our approach can derive a significantly faster algorithm using this BLAS implementation as building block (leaf).

The paper is organized as follows. In Section 2, we introduce the related work. In Section 3, we introduce the notations and matrix computation algorithms such as MM and MA. In Section 4, we introduce the fast algorithms and thus our main contributions. In Section 6, we introduce our experiments and divide them into single and double precision sections (thus single and double complex).

2. RELATED WORK

Our main interest is the design and implementation of highly portable codes that automatically adapt to the architecture evolution, that is, **adaptive codes** (e.g., [Frigo and Johnson 2005; Demmel et al. 2005; Püschel et al. 2005; Gunnels et al. 2001]). In this paper, we discuss a family of algorithms from dense linear algebra: **Matrix Multiply** (MM). All algorithms apply to any size and shape matrices stored 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], and using column-major order [Higham 1990; Whaley and Dongarra 1998; Goto and van de Geijn 2008]).

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]. In turn, BLAS 3 can be built on top of an efficient implementations of the MM kernel [Kagstrom et al. 1998a; 1998b]. ATLAS [Whaley and Dongarra 1998; Whaley and Petitet 2005; Demmel et al. 2005]

Table I. Recursion point (problem size when we yield to GEMM) for a few architectures and for different precisions

machine	single	double	single complex	double complex
Opteron	3100	3100	1500	1500
Pentium	3100	2500	1500	1500
Nehalem	3800	3800	2000	2000
Xeon	7000	7000	3500	3500
Shanghai	3500	3500	3500	3500

(pseudo successor of PHiPAC [Bilmes et al. 1997]) is a very good example of an adaptive software package implementing BLAS, by tuning codes for many architectures around a highly tuned MM kernel optimized automatically for the L1 and L2 data caches. Recently however, GotoBLAS [Goto and van de Geijn 2008] (or Intel's MKL or a few vendor implementations) is offering consistently better performance than ATLAS.

In this paper, we take a step forward towards the parallel implementation of fast MM algorithms: we show how, when, and where our (novel) hybrid implementations of Strassen's, Winograd's, and the 3M algorithms [Strassen 1969; Douglas et al. 1994; D'Alberto and Nicolau 2009; 2007] improve the performance over the best available adaptive matrix multiply (e.g., ATLAS or GotoBLAS). 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 $O(N^3)$. We take the freedom to stretch the term *fast algorithm* in such a way to comprise the 3M algorithm for complex matrices. Strassen's algorithm [Strassen 1969] is the first practically used among the fast algorithms for MM.

The asymptotically fastest algorithm to date is by Coppersmith and Winograd $O(n^{2.376})$ [Coppersmith and Winograd 1987]. Pan showed a bi-linear algorithm that is asymptotically faster than Strassen-Winograd [Pan 1978] $O(n^{2.79})$ (i.e., see Pan's survey [Pan 1984] with best asymptotic complexity of $O(n^{2.49})$). Kaporin [Kaporin 1999; 2004] presented the implementation of Pan's algorithm $O(n^{2.79})$. For the range of problem sizes presented in this work, the asymptotic complexity of Winograd's and Pan's is similar. 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 relatively small matrices, Winograd's MM has a significant overhead and classic MMs are more appealing. To overcome this, Strassen/Winograd's MM is used in conjunction with classic MM [Brent 1970b; 1970a; Higham 1990]: 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. The recursion point depends on various properties of the machine hardware, and thus will vary from one machine to the next.

In table I, we show an excerpt of the possible recursion points for the architecture presented in this paper. However, the recursion point is immaterial *per se* because it can be always estimated and tuned for any architecture and family of problems (e.g., real or complex).

A practical problem with Strassen and Winograd's algorithms is how to divide the matrix when not a power of two. All our algorithms divide the MM problems into a set of *balanced* subproblems; that is, with minimum difference of operation count (i.e., complexity). This balanced division leads to simple code, and natural recursive parallelism. 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 is applied once or recursively, and a few irregular computations based on matrix-vector operations.

Recently, there is a new interest in discovering new algorithms (or schedules) as well as new implementations taken from the Winograd's MM. Our work is similar to the ones presented in [Dumas et al. 2008; Boyer et al. 2009] and takes from a concurrent work in [Bodrato 2010]. The first presents a library (how to integrate finite precision MM computation with high performance double precision MM) and the second presents new implementations in such a way to minimize the memory space (i.e., foot print) of Strassen–Winograd's MM. Memory efficient fast MMs are interesting because the natural reduction of memory space also improves the data locality of the computation (extremely beneficial when data spill to disk). The third proposes an optimized algorithm to reduce even further the computation for matrices of odd sizes. In contrast, in this work, we are going to show that to achieve performance, we need parallelism; to exploit parallelism, we need more temporary space (worse memory foot print). We show that the fastest algorithm (among the ones presented in this work) requires the largest space to allow more parallelism; thus, if we can work in memory without disk spills, we present the best performance strategy, otherwise a hybrid approach is advised.

3. ALGORITHMS DESIGN AND TASKS SCHEDULING

In this section, we present MM (i.e., the classic one) and MA algorithms and implementations for multicore multiprocessor SMP systems. We present both the main algorithms and the optimizations in an intuitive but rigorous way.

Before we present any algorithm, we introduce our notations. Consider any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, this can be always divided into four quadrants:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_1 \\ \mathbf{A}_2 & \mathbf{A}_3 \end{bmatrix} \tag{1}$$

where $\mathbf{A}_0 \in \mathbb{R}^{\lceil \frac{m}{2} \rceil \times \lceil \frac{n}{2} \rceil}$ and $\mathbf{A}_3 \in \mathbb{R}^{\lfloor \frac{m}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor}$. In the same way, we can divide a matrix into two vertical matrices (similarly horizontal ones):

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_1 \end{bmatrix} \tag{2}$$

where $\mathbf{A}_0 \in \mathbb{R}^{m \times \lceil \frac{n}{2} \rceil}$ and $\mathbf{A}_1 \in \mathbb{R}^{m \times \lfloor \frac{n}{2} \rfloor}$.

3.1. Classic Matrix Multiplication $O(N^3)$

The classic matrix multiplication (MM) of $\mathbf{C}{=}\mathbf{A}\mathbf{B}$ with $\mathbf{C}, \mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$ is defined by the following recursive algorithm:

$$\begin{array}{ll} \mathbf{C}_0{=}\mathbf{A}_0\mathbf{B}_0 + \mathbf{A}_1\mathbf{B}_2 & \mathbf{C}_1{=}\mathbf{A}_0\mathbf{B}_1 + \mathbf{A}_1\mathbf{B}_3, \\ \mathbf{C}_2{=}\mathbf{A}_2\mathbf{B}_0 + \mathbf{A}_3\mathbf{B}_2 & \mathbf{C}_3{=}\mathbf{A}_2\mathbf{B}_1 + \mathbf{A}_3\mathbf{B}_3 \end{array} \tag{3}$$

The computation is divided into four parts, one for each sub-matrix composing $\mathbf{C}$. Thus, for every matrix $\mathbf{C}_i$ ($0 \leq i \leq 3$), the classic approach computes two products, using a total of 8 MMs and 4 MAs (which can be computed concurrently with MM and thus not contributing to the overall execution time). Notice that every product computes a result that has the same size and shape as the destination sub-matrix $\mathbf{C}_i$. 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 3 applies unchanged to these subproblems.

The time complexity of MM is $2n^3$ (n^3 multiplications and n^3 additions); the space complexity is $3n^2$ requiring at least the storage of the addenda and the result; the number of memory accesses (if the problem fits in memory) also known as I/O complexity or access complexity is $O(\frac{n^3}{\sqrt{S}})$ where S is the size of the largest cache as a function of the number of matrix elements; the constant term is a function of how the problem is divided. [1]

The MM procedure has data locality and we can easily parallelize MM using threads on a SMP architecture; for example, Equation 3 presents one policy of how to divide the computation. In this work, we deploy either ATLAS or GotoBLAS multi-threaded MM where the number of threads (the parallelism) and the core allocation can be driven by either a few global environment variables or, as in our case, using the Linux utility, *set_affinity*.

3.2. Matrix Addition

Matrix addition (MA) $\mathbf{C} = \mathbf{A} + \mathbf{B}$ with $\mathbf{C}, \mathbf{A}, \mathbf{B} \in \mathbb{R}^{n\times n}$, has time complexity $O(n^2)$ because it requires n^2 additions, $2n^2$ reads and n^2 writes; thus $3n^2$ memory access operations. The MA has space complexity $3n^2$ because its input and output are three matrices.

The MA does not have temporal locality because any matrix element is used just once. It has spatial locality; that is, we tend to read and write contiguous elements from/to the matrices. The nonexistent temporal locality makes a hierarchical memory (level of caches) completely useless and actually harmful (we could achieve better performance if we just circumvent the caches and go directly to the register files). The MA is embarrassingly parallel. For example, if we divide a matrix into two vertical sub-matrices we can easily describe the addition as two parallel MAs (Equation 4):

$$\left| \begin{array}{cc} \overset{p_0}{\mathbf{C}_0} & \overset{p_1}{\mathbf{C}_1} \end{array} \right| = \left| \begin{array}{cc} \overset{p_0}{\mathbf{A}_0 + \mathbf{B}_0} & \overset{p_1}{\mathbf{A}_1 + \mathbf{B}_1} \end{array} \right| \tag{4}$$

We can perform this division recursively until we achieve the number of tasks necessary and, also, we can determine a logical allocation of each MA to a specific core/processor. We can time the MA performance for the purpose of investigating the data throughput to all cores. For example, if the MA performance improves as the number of threads increases till matching the number of cores, we can state that the memory system has the bandwidth to provide data to all cores (the computation cost is negligible compared to the communication cost). In this work, all architectures can actually feed all cores for computing MA (note this is not necessarily true for other applications such as the classic MM because of the different access complexity). However, notice that even when the memory system has the bandwidth to sustain computation to all cores, it does not means that we can achieve the maximum throughput of any core.

Remark. to simplify the complexity model and operation counts, in the following, we consider a matrix copy (i.e., $A = B$) as time consuming as a matrix addition.

3.3. The classic MM and MA algorithm as building blocks for fast algorithms

In the following, we consider the classic MM and the MA as parallel routines optimized for the specific architectures. Our fast matrix multiplication algorithms are written as a sequence of these operations (recursive divide-and-conquer algorithms) and we model the system simply as the composition of a core set and a single memory storage (i.e., parallel random access memory machine PRAM). We further make sure that: first,

[1] The recursive division in Equation 3 is not the best but asymptotically optimal nonetheless.

each basic operation is optimized for the architecture achieving optimal/best throughput; second, the problem size is large enough that a parallel system is necessary to obtain results in a timely manner. In the following, we investigate how to integrate and combine MM and MA to optimize the code and improve the performance.

4. FAST MATRIX MULTIPLICATION ALGORITHMS

We present all the following fast MMs as a sequence of MMs and MAs. As such, any MM can be either a recursive call to a fast MM or a direct call to an optimized and classic BLAS MM (GEMM). A recursive MM algorithm will switch to the classic MM for any problem sizes smaller than a specified threshold or recursion point (we specify in the set up how this point is computed, Section 6.1). The recursion point is a function of the architecture, the implementation of the classic MM algorithm and the implementation of MA. In this work, the recursion point is found empirically for each architecture once the library is installed. We present four basic hybrid algorithms based on a balanced decomposition of the matrix operands: 3M Section 4.1, Strassen's 4.3 , Winograd 4.2, Winograd optimized to reduce MAs (WOPT) 4.4, and Winograd optimized to hide MA latency (WIDEAL) 4.5.

4.1. 3M Matrix Multiply

Table II. 3M algorithm.

Sequential		Parallel/Pipelining			
$\mathbf{C}_R =$	$\mathbf{A}_R\mathbf{B}_R$	1: $\mathbf{C}_R = \mathbf{A}_R\mathbf{B}_R$	$\mathbf{T} = \mathbf{A}_R + \mathbf{A}_I$	$\mathbf{S} = \mathbf{B}_R + \mathbf{B}_I$	
$\mathbf{C}_I =$	$\mathbf{A}_I\mathbf{B}_I$	2: $\mathbf{C}_I = \mathbf{A}_I\mathbf{B}_I$			
$\mathbf{C}_I =$	$\mathbf{C}_I + \mathbf{C}_R$	3: $\mathbf{C}_I = \mathbf{C}_I + \mathbf{C}_R$			
$\mathbf{C}_R =$	$2\mathbf{C}_R - \mathbf{C}_I$	4: $\mathbf{C}_R = 2\mathbf{C}_R - \mathbf{C}_I$			
$\mathbf{T} =$	$\mathbf{A}_R + \mathbf{A}_I$	5: $\mathbf{C}_I - = \mathbf{ST}$			
$\mathbf{S} =$	$\mathbf{B}_R + \mathbf{B}_I$				
$\mathbf{C}_I - =$	$\mathbf{ST}$				

We can consider a complex matrix $\bar{\mathbf{A}}$ as either a matrix where each element is a complex number $\bar{a}_{k,\ell} = (b_{k,\ell} + ic_{k,\ell})$, or the composition of two real matrices $\bar{\mathbf{A}} = \mathbf{A}_R + i\mathbf{A}_I$. If we adopt the former, we can compute the MM by any implementation discussed so far. If we adopt the latter, we can use an even faster algorithm: the 3M multiplication as in Table II. Our implementation hides the latency of the last MA because MA and MM are computed at the same time, and it introduces a small extra computation because one addition is of the form: $\mathbf{C}_R = 2\mathbf{C}_R - \mathbf{C}_I$. In practice, this algorithm has 3 MM, thus the name of the algorithm, and only four MAs (instead of five).

The MM dominates the overall time complexity. Independently of our real-MM-algorithm choice, the 3M algorithm requires extra space for the temporary matrices $\mathbf{T}$ and $\mathbf{S}$ (i.e., an extra complex matrix). So at a minimum, the 3M algorithm has a space complexity of $4n^2$. If we apply the classic algorithm, the 3M algorithm has the time complexity of $O(3 * 2n^3)$. Due to the data locality, we know that given the largest cache sizes S (e.g., L2 or L3) we have $O(3\frac{2n^3}{\sqrt{S}})$ memory accesses. The 3M algorithm —in comparison with the classic MM applied to complex matrices— offers a $\sim \frac{4}{3}$ speed up. This fast algorithm, actually every 3M algorithm, *compromises* the precision of the imaginary part of the complex result matrix because of the compounding error effects of both MA and MM similar to the Strassen's algorithm.

Algorithm and scheduling. We notice that the MAs $\mathbf{T} = \mathbf{A}_R + \mathbf{A}_I$ and $\mathbf{S} = \mathbf{B}_R + \mathbf{B}_I$ can be executed in parallel to the previous addition and actually as early as the first matrix multiplications as we suggested in Table II; that is, we perform a function percolation (upward movement), in principle as in [Nicolau et al. 2009]. Notice that we did schedule the two independent MMs $\mathbf{C}_R = \mathbf{A}_R\mathbf{B}_R$ and $\mathbf{C}_I = \mathbf{A}_I\mathbf{B}_I$ into two different steps (line 1 and 2) (instead of executing them in parallel together). We do this because each MM will be parallelized and we will achieve the best asymptotic performance for each multiplication separately, thus there would be no loss of performance.

If the MM cannot exploit full parallelism from an architecture, some cores are better running idle than executing small sub problems of MM (because we may achieve a lower throughput). A MA has the same space complexity of a MM, it has a (much) smaller time complexity and, more importantly, fewer memory accesses: $3n^2$ vs. $2n^3/\sqrt{S}$, where S is the size of the largest cache. Here, we have the opportunity to use those idle cores to run MAs thus hiding the MAs latency and taking full advantage of an otherwise under utilized system. In practice, we will show that the thread parallelism between MAs concurrently with a parallel MM is beneficial even when there are no idle/under-utilized cores.

Our idea is to schedule a sequential implementation of MA such as $\mathbf{T} = \mathbf{A}_R + \mathbf{A}_I$ and $\mathbf{S} = \mathbf{B}_R + \mathbf{B}_I$ (line 1) to different cores (e.g., core 0 and core 1 respectively of a 4 core system), thus to exploit functional/thread parallelism in assigning a specific set of threads to specific cores. The MAs in line 3 and 4 are parallelized independently and fully (e.g., both using all cores, 4) as described in Section 3.2. We will show that this multi-threading can be advantageous even when the MM will use all cores.

Finally, we have a 3M algorithm with only two MAs in the critical path. In hindsight, this is a simple observation that is the backbone of our approach and we apply it to other algorithms (i.e., Winograd's and Strassen's) as well.

4.2. Winograd

Winograd reduced Strassen's number of MAs by the reuse of the partial results. In Table III, we present our algorithm.

We use three temporary matrices $\mathbf{S}, \mathbf{T}$, and $\mathbf{U}$, where we use the first two to combine submatrices of $\mathbf{A}$ and $\mathbf{B}$, respectively, and we use the last one to combine results of the MMs (thus of sub-matrices of $\mathbf{C}$). This ability to remember part of the computation using the temporary matrices saves MAs on one side (w.r.t. the Strassen algorithm see next section), but it forces a *stronger* data dependency in the computation on the other side; that is, for the Winograd's algorithm, we can overlap 4 MMs with MAs as we are going to show in Table III —for Strassen's algorithm, we can overlap 6 MMs as discussed in Section 4.3.

Algorithm and scheduling. On the left of Table III, we present the sequential recursive algorithm where MM and MA operations between matrices in the schedule are assumed to be parallel in the following discussion (e.g., MM is a call to the parallel GotoBLAS or a recursive call to the Winograd's algorithm and MA is parallelized as explained in Section 3.2). Once again, the schedule on the right in Table III will allocate up to two MAs in parallel with MMs. We decided to use sequential code for these MAs and to set statically the core executing them as a function of the system; however, we could have used parallel codes as well.

This has the potential to utilize better the system while hiding the latency of 6 MAs (we exploit thread parallelism in such a way to hide the MA latency). In practice, we may say that the effective length of the schedule (critical path) is only 8MAs (i.e., we consider a matrix copy as expensive as a MA).

Table III. Winograd's MM (our implementation, WINOGRAD).

Sequential		Parallel/Pipelining			
$\mathbf{S}$	$= \mathbf{A}_2 + \mathbf{A}_3$	1:	$\mathbf{S} = \mathbf{A}_2 + \mathbf{A}_3$		
$\mathbf{T}$	$= \mathbf{B}_1 - \mathbf{B}_0$	2:	$\mathbf{T} = \mathbf{B}_1 - \mathbf{B}_0$		
$\mathbf{U}$	$= \mathbf{ST}$	3:	$\mathbf{U} = \mathbf{ST}$		
$\mathbf{C}_1$	$= \mathbf{U}$				
$\mathbf{C}_3$	$= \mathbf{U}$				
$\mathbf{C}_0$	$= \mathbf{A}_0\mathbf{B}_0$	4:	$\mathbf{C}_0 = \mathbf{A}_0\mathbf{B}_0$	$\mathbf{C}_1 = \mathbf{U}$	$\mathbf{C}_3 = \mathbf{U}$
$\mathbf{U}$	$= \mathbf{C}_0$	5:	$\mathbf{U} = \mathbf{C}_0$		
$\mathbf{C}_0$	$+= \mathbf{A}_1\mathbf{B}_2$	6:	$\mathbf{C}_0 += \mathbf{A}_1\mathbf{B}_2$	$\mathbf{S} = \mathbf{S} - \mathbf{A}_0$	$\mathbf{T} = \mathbf{B}_3 - \mathbf{T}$
$\mathbf{S}$	$= \mathbf{S} - \mathbf{A}_0$				
$\mathbf{T}$	$= \mathbf{B}_3 - \mathbf{T}$				
$\mathbf{U}$	$+= \mathbf{ST}$	7:	$\mathbf{U} += \mathbf{ST}$		
$\mathbf{C}_1$	$+= \mathbf{U},$	8:	$\mathbf{C}_1 += \mathbf{U},$		
$\mathbf{S}$	$= \mathbf{A}_1 - \mathbf{S}$	9:	$\mathbf{S} = \mathbf{A}_1 - \mathbf{S}$		
$\mathbf{C}_1$	$+= \mathbf{SB}_3$	10:	$\mathbf{C}_1 += \mathbf{SB}_3$	$\mathbf{T} = \mathbf{B}_2 - \mathbf{T}$	
$\mathbf{T}$	$= \mathbf{B}_2 - \mathbf{T}$				
$\mathbf{C}_2$	$= \mathbf{A}_3\mathbf{T}$	11:	$\mathbf{C}_2 = \mathbf{A}_3\mathbf{T}$	$\mathbf{S} = \mathbf{A}_0 - \mathbf{A}_2$	
$\mathbf{S}$	$= \mathbf{A}_0 - \mathbf{A}_2$	12:	$\mathbf{T} = \mathbf{B}_3 - \mathbf{B}_1$		
$\mathbf{T}$	$= \mathbf{B}_3 - \mathbf{B}_1$	13:	$\mathbf{U} += \mathbf{ST}$		
$\mathbf{U}$	$+= \mathbf{ST}$	14:	$\mathbf{C}_3 += \mathbf{U}$		
$\mathbf{C}_3$	$+= \mathbf{U}$	15:	$\mathbf{C}_2 += \mathbf{U}$		
$\mathbf{C}_2$	$+= \mathbf{U}$				

On the recursion and parallelism. The algorithm description presents a *flat* representation of the computation, that is, it is static. The main difference between the 3M algorithm scheduling and the Winograd's algorithm regarding the parallel computation of MAs and MMs is about the recursive nature of the algorithm. Assume we are computing MM for matrices of size $k * n_1 \times k * n_1$ where n_1 is the recursion point where we use GEMM directly and $k > 1$ is a natural number.

If we follow the recursion, after k recursive calls, it is in step 3 that we have the first MM described in the static algorithm. The MM is on matrices of size $n_1 \times n_1$ and we actually call the GEMM. In parallel, we are going to execute up to two MAs, for which the maximum size will be $(k-1)n_1 \times (k-1)n_1$ for the Winograd algorithm and $k * n_1 \times k * n_1$ for the 3M algorithm (if we are using this Winograd algorithm). The complexity is a function of which recursion level demands the MAs. To keep constant the number of MAs to be executed in parallel with GEMM, we keep the first MM recursive call free of any parallel MA (as in step 3 in the Winograd algorithm). Otherwise, we may have to compute as many as $2 * k$ MAs "as we would for the Strassen's algorithm, discussed next.

4.3. Strassen

This idea of exploiting parallelism between MMs and MAs exposes another interesting scenario especially for the Strassen's algorithm. In Table IV, we present our implementation of Strassen's algorithm. Each matrix multiplication is independent of each other; that is, with minor modifications in how to store the data into the matrix $\mathbf{C}$, we could change the order of the computation as we like. If we sacrifice a little more space using four matrices (instead of just two $\mathbf{S}$ and $\mathbf{T}$), we could exploit more parallelism between MA and MM as we show on the right in Table IV.

Table IV. Strassen's MM (our implementation STRASSEN).

Sequential	Parallel/Pipelining			
$\mathbf{T} = \mathbf{B}_1 - \mathbf{B}_3$	1:	$\mathbf{T} = \mathbf{B}_1 - \mathbf{B}_3$		
$\mathbf{U} = \mathbf{A}_0 * \mathbf{T}$	2:	$\mathbf{U} = \mathbf{A}_0 * \mathbf{T}$	$\mathbf{V} = \mathbf{A}_2 - \mathbf{A}_0$	$\mathbf{Z} = \mathbf{B}_0 + \mathbf{B}_1$
$\mathbf{C}_1 = \mathbf{U}$	3:	$\mathbf{C}_1 = \mathbf{U}$		
$\mathbf{C}_3 = \mathbf{U}$				
$\mathbf{S} = \mathbf{A}_2 - \mathbf{A}_0$				
$\mathbf{T} = \mathbf{B}_0 + \mathbf{B}_1$				
$\mathbf{U} = \mathbf{S} * \mathbf{T}$	4:	$\mathbf{U} = \mathbf{V} * \mathbf{Z}$	$\mathbf{C}_3 = \mathbf{C}_1$	$\mathbf{S} = \mathbf{A}_2 + \mathbf{A}_3$
$\mathbf{C}_3 += \mathbf{U}$	5:	$\mathbf{C}_3 += \mathbf{U}$		
$\mathbf{S} = \mathbf{A}_2 + \mathbf{A}_3$				
$\mathbf{U} = \mathbf{S} * \mathbf{B}_0$	6:	$\mathbf{U} = \mathbf{S} * \mathbf{B}_0$	$\mathbf{V} = \mathbf{A}_0 + \mathbf{A}_3$	$\mathbf{Z} = \mathbf{B}_0 + \mathbf{B}_3$
$\mathbf{C}_3 -= \mathbf{U}$				
$\mathbf{C}_2 = \mathbf{U}$	7:	$\mathbf{C}_2 = \mathbf{U}$		
$\mathbf{S} = \mathbf{A}_0 + \mathbf{A}_3$				
$\mathbf{T} = \mathbf{B}_0 + \mathbf{B}_3$				
$\mathbf{U} = \mathbf{S} * \mathbf{T}$	8:	$\mathbf{C}_0 = \mathbf{V} * \mathbf{Z}$	$\mathbf{C}_3 -= \mathbf{C}_2$	$\mathbf{S} = \mathbf{A}_0 + \mathbf{A}_1$
$\mathbf{C}_3 += \mathbf{U}$	9:	$\mathbf{C}_3 += \mathbf{C}_0$		
$\mathbf{C}_0 = \mathbf{U}$				
$\mathbf{S} = \mathbf{A}_0 + \mathbf{A}_1$				
$\mathbf{U} = \mathbf{S} * \mathbf{B}_3$	10:	$\mathbf{U} = \mathbf{S} * \mathbf{B}_3$	$\mathbf{V} = \mathbf{A}_1 - \mathbf{A}_3$	$\mathbf{Z} = \mathbf{B}_2 + \mathbf{B}_3$
$\mathbf{C}_0 -= \mathbf{U}$	11:	$\mathbf{C}_0 -= \mathbf{U}$		
$\mathbf{C}_1 += \mathbf{U}$	12:	$\mathbf{C}_1 += \mathbf{U}$		
$\mathbf{S} = \mathbf{A}_1 - \mathbf{A}_3$				
$\mathbf{T} = \mathbf{B}_2 + \mathbf{B}_3$				
$\mathbf{U} = \mathbf{S} * \mathbf{T}$				
$\mathbf{C}_0 += \mathbf{U}$	13:	$\mathbf{C}_0 += \mathbf{V} * \mathbf{Z}$	$\mathbf{S} = \mathbf{B}_2 - \mathbf{B}_0$	
$\mathbf{S} = \mathbf{B}_2 - \mathbf{B}_0$	14:	$\mathbf{U} = \mathbf{A}_3 * \mathbf{S}$		
$\mathbf{U} = \mathbf{A}_3 * \mathbf{S}$	15:	$\mathbf{C}_0 += \mathbf{U}$		
$\mathbf{C}_0 += \mathbf{U}$	16:	$\mathbf{C}_2 += \mathbf{U}$		
$\mathbf{C}_2 += \mathbf{U}$				

We can hide the latency of more MAs behind the computation of MMs than what we did for the Winograd's algorithm. Nonetheless, this Strassen's implementation has 9 parallel MAs in the critical path and it has only one MA more than the Winograd's implementation.

What we have is a Strassen's algorithm that, in this specific situation and using an unusual computational model, can be as fast as the Winograd's implementation. However, the round-off error and its characterization is exactly as for the sequential case (we just exploit parallelism among operations that have no data dependency and thus no error propagation). So when this scheduling is applicable, we have a new algorithm with the speed of the Winograd's algorithm and the asymptotic error analysis of the Strassen's algorithm (i.e., faster and more accurate).

This schedule is new and a contribution of this paper and we may conjecture that there may be a formulation of Winograd's, a formulation of Strassen's algorithm, and a parallel architectures for which the algorithms have the same execution time (time complexity). In this work, we show that there are a couple of systems where this type of software pipelining is beneficial and we show that Strassen's algorithm (Table IV) has performance very close to the Winograd's algorithm performance (Table III).

On the recursion and parallelism. Notice that the two MAs executed in parallel with MM (e.g., in step 2 Table IV) are set to specific cores and are sequential codes.

The MAs such as the ones in the critical path are parallel as described in Section 3.2. Because of our algorithm design, during the unfolding of the recursion, we may accumulate up to two MAs per recursion level to be executed in parallel with the GEMM MM. In practice and for the problem sizes we present in this paper, we can apply between 1–3 levels of recursion. If we start with a problem size $3n_1 \times 3n_1$ we will have 2 MAs of complexity $O(9n_1^2)$, 2 MAs of complexity $O(4n_1^2)$, 2 of complexity $O(n_1^2)$, and in parallel with one GEMM of complexity $O(2n_1^3)$.

4.4. Improved Winograd: reduced number of MAs (WOPT)

Table V. Winograd's MM (improved implementation C=AB WOPT).

Sequential		Parallel/Pipelining			
$\mathbf{S}$	$= \mathbf{A}_3 - \mathbf{A}_2$	1:	$\mathbf{S} = \mathbf{A}_3 - \mathbf{A}_2$		
$\mathbf{T}$	$= \mathbf{B}_3 - \mathbf{B}_2$	2:	$\mathbf{T} = \mathbf{B}_3 - \mathbf{B}_2$		
$\mathbf{C}_3$	$= \mathbf{ST}$	3:	$\mathbf{C}_3 = \mathbf{ST}$		
$\mathbf{U}$	$+= \mathbf{A}_1\mathbf{B}_2$	4:	$\mathbf{U} += \mathbf{A}_1\mathbf{B}_2$		
$\mathbf{C}_0$	$= \mathbf{A}_0\mathbf{B}_0$	5:	$\mathbf{C}_0 = \mathbf{A}_0\mathbf{B}_0$	$\mathbf{S} = \mathbf{S} + \mathbf{A}_1$	$\mathbf{T} = \mathbf{T} + \mathbf{B}_1$
$\mathbf{C}_0$	$+= \mathbf{U}$	6:	$\mathbf{C}_0 += \mathbf{U}$		
$\mathbf{S}$	$= \mathbf{S} + \mathbf{A}_1$				
$\mathbf{T}$	$= \mathbf{T} + \mathbf{B}_1$				
$\mathbf{U}$	$+= \mathbf{ST}$	7:	$\mathbf{U} += \mathbf{ST}$		
$\mathbf{C}_1$	$= \mathbf{U} - \mathbf{C}_3$	8:	$\mathbf{C}_1 = \mathbf{U} - \mathbf{C}_3$		
$\mathbf{S}$	$= \mathbf{A}_0 - \mathbf{S}$	9:	$\mathbf{S} = \mathbf{A}_0 - \mathbf{S}$		
$\mathbf{C}_1$	$+= \mathbf{SB}_1$	10:	$\mathbf{C}_1 += \mathbf{SB}_1$	$\mathbf{T} = \mathbf{B}_0 - \mathbf{T}$	
$\mathbf{T}$	$= \mathbf{B}_0 - \mathbf{T}$				
$\mathbf{C}_2$	$+= \mathbf{A}_2\mathbf{T}$	11:	$\mathbf{C}_2 += \mathbf{A}_2\mathbf{T}$	$\mathbf{S} = \mathbf{A}_3 - \mathbf{A}_1$	$\mathbf{T} = \mathbf{B}_3 - \mathbf{B}_1$
$\mathbf{S}$	$= \mathbf{A}_3 - \mathbf{A}_1$				
$\mathbf{T}$	$= \mathbf{B}_3 - \mathbf{B}_1$				
$\mathbf{U}$	$-= \mathbf{ST}$	12:	$\mathbf{U} -= \mathbf{ST}$		
$\mathbf{C}_3$	$-= \mathbf{U}$	13:	$\mathbf{C}_3 -= \mathbf{U}$		
$\mathbf{C}_2$	$-= \mathbf{U}$	14:	$\mathbf{C}_2 -= \mathbf{U}$		

The previous Strassen–Winograd algorithms (i.e., in Section 4.2 and 4.3) have two basic weaknesses. First, the algorithms present the same scheduling whether or not we perform the regular MM $\mathbf{C} = \mathbf{AB}$ or the accumulate matrix multiply $\mathbf{C+=AB}$, requiring more space and performing 4 more MAs. Second, these same algorithms use more the larger sub-matrices $\mathbf{A}_0$ and $\mathbf{B}_0$ (e.g., $\lceil \frac{N}{2} \times \frac{N}{2} \rceil$), instead of the smaller sub-matrices $\mathbf{A}_3$ and $\mathbf{B}_3$ (e.g., $\lfloor \frac{N}{2} \times \frac{N}{2} \rfloor$), and thus a potential saving of about $O(N^2)$ for odd matrices. In this section, we address and provide a solution for both:

$\mathbf{C} = \mathbf{AB}$ vs. $\mathbf{C+=AB}$.

In the literature, we can find that the former algorithm (without post matrix addition $\mathbf{C} = \mathbf{AB}$) requires four fewer matrix additions and it requires one fewer temporary matrix than the latter, by using the destination matrix $\mathbf{C}$ as temporary matrix. Though, we do not pursue the minimization of the space (by temporary matrices), we find it useful to reduce the number of MAs when possible. So we propose an algorithm where the two computations (with and without accumulation) are considered separately (Table V and VI) and also the software pipelining is considered separately.

Fewer uses of $\mathbf{A}_0$ and $\mathbf{B}_0$.

We proposed the division of a matrix $\mathbf{A}$ into four balanced sub matrices $\mathbf{A}_0$, $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$ as in Equation 1, because we wanted to reduce Strassen–Winograd's algorithm into seven balanced sub-problem [D'Alberto and Nicolau 2007]. Such a division is optimal (i.e., asymptotically and w.r.t. to any unbalanced division used in peeling and padding) and provides a natural extension for the Strassen's algorithm to any problem sizes (and thus to the Winograd's variant). Such a generalization uses more the larger sub-matrix $\mathbf{A}_0$ than the smaller matrix A_3 (i.e., more MMs with operands of the sizes of A_0 and B_0 instead of the size of A_3 and B_3).

In the literature we can find different algorithms especially in light of the following simple equation taken from [Loos and Wise]:

$$\mathbf{C} = (\mathbf{AP})(\mathbf{P}^t\mathbf{B}) \tag{5}$$

where the unitary permutation matrix $\mathbf{P}$ rotates the sub-matrices of $\mathbf{A}$ and $\mathbf{B}$ accordingly (for example, anti-clockwise half rotation $\mathbf{A}_0 \rightarrow \mathbf{A}_3$ and $\mathbf{A}_1 \rightarrow \mathbf{A}_2$). This means, we can obtain further savings by a reorganization of the computation and a re-definition of matrix addition (the savings are of the order of $O(N^2)$, a few MAs).

In fact, we have found a schedule (algorithm) that allows us to use our matrix additions —thus without changing our algebra— and it has equivalent complexity. In Table V and VI we present the final algorithms.

Notice that for even-sized matrices the size of $\mathbf{A}_0$ is equal to the size of $\mathbf{A}_3$, and thus there is no savings. For odd-size these savings are of the order of $O(N^2)$ per each level of recursion, and in relative terms, if we have a recursion point N_0, we achieve a relative saving of $O(\frac{1}{N_0})$.

If we consider only the number of operations as savings, for the systems presented in this work we may save 1% operations. For most researchers, this represents little or no improvement; however, with the application of the techniques here presented, we can achieve up to 10% (on top of 20% improvement from the classic MM), making this more appealing.

4.5. Ideal Winograd: No MAs in the critical path

In the previous sections, we presented algorithms that may hide a few MAs in parallel with MMs. In this section, we present a variant of the Winograd algorithm that requires two more temporary matrices (for a total of 5 temporary matrices) but for which we are able to hide the latency of all MAs. Thus we have an algorithm that, in principle, can achieve the theoretical speed up of (approximated well by) $\sim (\frac{8}{7})k$ where k is the number of recursive levels (in practice, $1 \leq k \leq 3$ and thus about a potential speed up of 30%). In other words, in the presence of sufficient cores being available, we can hide the MAs by executing them in parallel with MMs and since the MAs take less time than the MMs, the MAs therefore have no impact/cost in terms of the critical path (i.e., total execution time) of the overall algorithm.

In Table VII, we present the algorithm for the product $\mathbf{C} = \mathbf{AB}$. As we can see, there are 7 parallel steps. The first recursive step is always executed without MAs in parallel. This assures that the number of MAs executed in parallel with the basic kernel MM is independent of the recursive level and, for this particular case, no more than 3 MAs (i.e., each MA is executed by a single thread on a different core in parallel with a multi-threaded MM; for example, ATLAS DGEMM).

Notice that the no-pipelined algorithm with fewer MAs (Table V) has the potential to be faster than the no-pipelind algorithm presented in this section (Table VII). We will show that this is the case in the experimental results. However, we have the opposite situation for the pipelined algorithms: the algorithm presented in this section is faster (than any of the previous ones).

Table VI. Winograd's MM (improved implementation C+=AB WOPT).

Sequential		Parallel/Pipelining			
$\mathbf{S}$	$= \mathbf{A}_3 - \mathbf{A}_2$	1:	$\mathbf{S} = \mathbf{A}_3 - \mathbf{A}_2$		
$\mathbf{T}$	$= \mathbf{B}_3 - \mathbf{B}_2$	2:	$\mathbf{T} = \mathbf{B}_3 - \mathbf{B}_2$		
$\mathbf{U}$	$= \mathbf{ST}$	3:	$\mathbf{U} = \mathbf{ST}$		
$\mathbf{C}_1$	$+= \mathbf{U}$	4:	$\mathbf{C}_1 += \mathbf{U}$		
$\mathbf{C}_3$	$-= \mathbf{U}$	5:	$\mathbf{C}_3 -= \mathbf{U}$		
$\mathbf{U}$	$+= \mathbf{A}_1\mathbf{B}_2$	6:	$\mathbf{U} += \mathbf{A}_1\mathbf{B}_2$		
$\mathbf{C}_0$	$+= \mathbf{U}$	7:	$\mathbf{C}_0 += \mathbf{U}$		
$\mathbf{C}_0$	$+= \mathbf{A}_0\mathbf{B}_0$	8:	$\mathbf{C}_0 += \mathbf{A}_0\mathbf{B}_0$	$\mathbf{S} = \mathbf{S} + \mathbf{A}_1$	$\mathbf{T} = \mathbf{T} + \mathbf{B}_1$
$\mathbf{S}$	$= \mathbf{S} + \mathbf{A}_1$				
$\mathbf{T}$	$= \mathbf{T} + \mathbf{B}_1$				
$\mathbf{U}$	$+= \mathbf{ST}$	9:	$\mathbf{U} += \mathbf{ST}$		
$\mathbf{C}_1$	$+= \mathbf{U}$	10:	$\mathbf{C}_1 += \mathbf{U}$		
$\mathbf{S}$	$= \mathbf{A}_0 - \mathbf{S}$	11:	$\mathbf{S} = \mathbf{A}_0 - \mathbf{S}$		
$\mathbf{C}_1$	$+= \mathbf{SB}_1$	12:	$\mathbf{C}_1 += \mathbf{SB}_1$	$\mathbf{T} = \mathbf{B}_0 - \mathbf{T}$	
$\mathbf{T}$	$= \mathbf{B}_0 - \mathbf{T}$				
$\mathbf{C}_2$	$+= \mathbf{A}_2\mathbf{T}$	13:	$\mathbf{C}_2 += \mathbf{A}_2\mathbf{T}$	$\mathbf{S} = \mathbf{A}_3 - \mathbf{A}_1$	$\mathbf{T} = \mathbf{B}_3 - \mathbf{B}_1$
$\mathbf{S}$	$= \mathbf{A}_3 - \mathbf{A}_1$				
$\mathbf{T}$	$= \mathbf{B}_3 - \mathbf{B}_1$				
$\mathbf{U}$	$-= \mathbf{ST}$	14:	$\mathbf{U} -= \mathbf{ST}$		
$\mathbf{C}_3$	$-= \mathbf{U}$	15:	$\mathbf{C}_3 -= \mathbf{U}$		
$\mathbf{C}_2$	$-= \mathbf{U}$	16:	$\mathbf{C}_2 -= \mathbf{U}$		

Expected improvements. What is going to be the maximum speed up by hiding the latency of MAs? In our previous work [D'Alberto and Nicolau 2009; 2007], we used a simplified complexity model to determine the recursion point (when the Winograd/Strassen algorithm yields to the BLAS GEMM) and for example with a single recursion level and for a classic Winograd algorithm (with 3 or more temporary matrices):

$$MM(N) = 7\alpha 2(\frac{N}{2})^3 + 15\beta(\frac{N}{2})^2 \tag{6}$$

The number of operations is obtained by 7 MMs, each performing $2(\frac{N^3}{2}^3)$ operations, and 15 MAs, each performing $\frac{N^2}{4}$ operations. Where α is the throughput of the MM and β is the throughput of MA. In practice, for most of the architectures presented in this paper it is fair to estimate the ratio $\frac{\beta}{\alpha} \sim 100$. In the original work the algorithms were sequential and what we wanted to compute was an estimate of the execution time.

What will be the speed up if we remove all MAs? First let us explicitly introduce the effect of the recursion in the execution time:

$$\begin{aligned} T(N) &= 7^i T(\frac{N}{2^i}) + \frac{15\beta N^2}{4}\sum_{j=0}^{i} -1(\frac{7}{4})^j \\ &= 7^i T(\frac{N}{2^i}) + 5\beta N^2[(\frac{7}{4})^i - 1] \end{aligned} \tag{7}$$

Table VII. Winograd's MM (no MAs in the critical path of C=AB WIDEAL).

Sequential		Parallel/Pipelining				
$\mathbf{U}$	$= \mathbf{A}_1\mathbf{B}_2$	1:	$\mathbf{U} = \mathbf{A}_1\mathbf{B}_2$			
$\mathbf{C}_0$	$=\mathbf{A}_0\mathbf{B}_0$	2:	$\mathbf{C}_0 = \mathbf{A}_0\mathbf{B}_0$	$\mathbf{S} = \mathbf{A}_3 - \mathbf{A}_2$	$\mathbf{T} = \mathbf{B}_3 - \mathbf{B}_2$	
$\mathbf{S}$	$= \mathbf{A}_3 - \mathbf{A}_2$					
$\mathbf{T}$	$= \mathbf{B}_3 - \mathbf{B}_2$					
$\mathbf{C}_3$	$= \mathbf{ST}$	3:	$\mathbf{C}_3 = \mathbf{ST}$	$\mathbf{C}_0$+=U	$\mathbf{V} = \mathbf{S} + \mathbf{A}_1$	$\mathbf{Z} = \mathbf{T} + \mathbf{B}_1$
$\mathbf{C}_0$	+= U					
$\mathbf{V}$	$= \mathbf{S} + \mathbf{A}_1$					
$\mathbf{Z}$	$= \mathbf{T} + \mathbf{B}_1$					
$\mathbf{U}$	$+= \mathbf{VZ}$	4:	$\mathbf{U}$+=VZ	$\mathbf{S} = \mathbf{A}_3 + \mathbf{A}_1$	$\mathbf{T} = \mathbf{B}_0 - \mathbf{Z}$	
$\mathbf{S}$	$= \mathbf{A}_3 + \mathbf{A}_1$					
$\mathbf{T}$	$= \mathbf{B}_0 - \mathbf{Z}$					
$\mathbf{C}_2$	$= \mathbf{A}_2\mathbf{T}$	5:	$\mathbf{C}_2 = \mathbf{A}_2\mathbf{T}$	$\mathbf{Z} = \mathbf{B}_3 + \mathbf{B}_1$	$\mathbf{C}_1 = \mathbf{U} - \mathbf{C}_3$	
$\mathbf{Z}$	$= \mathbf{B}_3 + \mathbf{B}_1$					
$\mathbf{C}_1$	$= \mathbf{U} - \mathbf{C}_3$					
$\mathbf{U}$	-= SZ	6:	U-=SZ	$\mathbf{V} = \mathbf{A}_0 - \mathbf{V}$		
$\mathbf{V}$	$= \mathbf{A}_0 - \mathbf{V}$					
$\mathbf{C}_1$	$+= \mathbf{VB}_1$	7:	$\mathbf{C}_1$+=$\mathbf{VB}_1$	$\mathbf{C}_3$-=U	$\mathbf{C}_2$-=U	
$\mathbf{C}_3$	-= U					
$\mathbf{C}_2$	-= U					

The ratio between the computation with additions and without is:

$$\begin{aligned} R(N) &= 1 + \frac{5\beta N^2[(\frac{7}{4})^i - 1]}{7^i T(\frac{N}{2^i})} \\ S(N) &= \frac{5\beta N^2[(\frac{7}{4})^i - 1]}{7^i T(\frac{N}{2^i})} \\ &\leq \frac{5\beta N^2}{4^i T(\frac{N}{2^i})} = \frac{5\beta 2^{i-1}}{\alpha N} \end{aligned} \tag{8}$$

For any level $i > 0$ of the recursion $T(\frac{N}{2^i}) = 2\alpha(\frac{N}{2})^3$, when we perform the computation of the leaf using the classic GEMM computation.

The speed up achievable with a single recursion is $S(N) \sim \frac{2\beta}{\alpha N}$. First, increasing N, the dominant term is αN^3 and the effect of hiding the MAs is decreasing as the problem size increases (we should find a decreasing speed up and we do show in the experimental results section). Of course, as we increases the number of recursions, we have an accumulative effect and we should experience a seesaw effect.

In fact, for a few recursion levels ($1 \leq k \leq 4$)

$$S_i(N) \geq 2^{i-1} S_1(N)$$

For example, for $N = 5000$, $k = 1$, and $\frac{\beta}{\alpha} = 100$ we should expect a speed up of about $S_1(5000) \sim 4\%$ (for $N = 10000$ we can expect to have at least two recursion levels and thus we could expect $S_2(10000) \sim 8\%$).

In all algorithms presented in this paper, we try to minimize the number of temporary matrices. In the literature, we can find that the minimum number of temporary matrices is three (without using the result matrix C as temporary space) and we must

perform more copy or matrix additions. In other words no implementations trying to reduce the temporary space will perform just 15.

When we take our original implementation of Winograd [D'Alberto and Nicolau 2009] the number of additions (and copies) is 14 and thus the speedup we could expect is $\frac{14\beta}{\alpha N}$ which is about 4% (per recursion level and N = 5000), in Figure 8 we show results exceeding this expectations to reach about 5%.

As last remark, if we consider double precision complex numbers and thus double complex operations, the throughput of the MM and MAs must be adjusted accordingly. For the architectures presented in this paper, for double precision complex data, we can say that the ratio $\frac{\alpha}{\beta}$ easily double ($\frac{\alpha}{\beta} \sim 200$ because MMs performs more operations per matrix element, thus we should expect an even better speed up, and in Figure 11 we meet such an expectation).

5. ON THE ERROR ANALYSIS

In this work, we will not present a theoretical evaluation of the numerical error we would incur when we use our fast algorithms. Such a topic is well covered in the literature [Higham 1990; Demmel and Higham 1992; Higham 2002; Dumas et al. 2008; D'Alberto and Nicolau 2009]. Instead, we will present a practical evaluation of the error analysis. That is, we present a quantitative evaluation of the error we would have in case we run these algorithms instead of standard GEMM algorithms or the MM algorithm based on the **doubly compensated summation algorithm** DCS [Priest 1991], which is tailored to minimize the error.

On one side, it is always possible to construct cases for which the worst case scenario is applicable, making these fast algorithms worse than standard algorithms. [2] On the other side, we show an example of the error on average: that is, what the error could be if the matrices are built using a random number generator and thus without the structure to create the worst case scenarios.

5.1. Parallel DCS based MM

In this paper, we emphasize the recursive nature of the MM algorithms. However, it is more intuitive to describe the MM algorithm based on the DCS by using a vector notation.

A single entry of a matrix c_{ij} is the result of the row-by-column computation $\sum_k a_{ik}b_{k,j}$. This is also the basic computation of the BLAS GEMM computation. Each element of the result matrix C is based on the independent computation of a summation. Of course, this algorithm is highly parallel, as soon as we split the computation of the matrix C to different cores.

The DCS algorithm reorganizes the summation in such a way that the products are ordered (in absolute module decreasing) and the error committed in every addition is compensated (actually three times compensated). The computation is naturally divided into parallel computations by splitting the matrix result C. This approach assures that a MM produces results at the precision of the architecture, however, we have found that this algorithm is often three order of magnitude slower than any less accurate algorithms.

In Section 6.4, we provide a quantitative estimate of the maximum absolute error (i.e., $\|C_{alg} - C_{DCS}\|_\infty$).

[2] For any problem size and for a finite number of recursive calls like we present in this paper, the bound is a known–constant

6. EXPERIMENTAL RESULTS

We divide this section into five parts. First, in Section 6.1, we introduce the set up starting with the five architectures we used to carry the experiments. Second, we provide our abbreviations and conventions in Section 6.2. Third, in Section 6.3, we provide the experimental results for matrices in complex double precision (double complex) and real double precision (i.e., 64 bits) in Figure 2–5. In Section 6.3.1, we give an in depth performance evaluation of the optimized Winograd's algorithm as presented in Table V and VI (e.g., Figure 6). Fifth, and last, we present a representative error analysis for one representative architecture and a selected set of fast algorithms, in Section 6.4.

6.1. Set up

We experiment with five multi-core multiprocessors systems: 2-core 2 Xeon (Pentium-based), 2-core 2 Opteron 270 (i386 and x86_64), 4-core 2 Xeon; 4-core 2 Opteron (Shanghai), and 8-core 2 Xeon (Nehalem).

Our approach views these architectures as one memory and one set of cores. In other words, we do not optimize the data layout for any specific architecture (i.e., the Nehalem, the shanghai, and the Opteron 270 have a separate and dedicated memory for each processor, while the others use a single memory bank and single bus). We optimize the performance of MM and MA for each architecture independently, by tuning the code, and then optimize the fast algorithms. We explain the procedure in the following.

MM installation, optimization, and tuning. For all architectures, we have installed GotoBLAS and ATLAS. Once the installation is finished, we tune the number of threads so as to achieve the best performance. We then have chosen the implementation that offers the best performance. If the optimal number of threads is smaller than the number of cores, the architecture has the potential for effective scheduling optimizations. However, notice that even when MM performance scales up nicely with the number of cores and will use all cores, we can still improve performance by the application of MA and MM pipelining (we present two systems for which this is possible and, for completeness, we show one system for which this is not).

MA installation, optimization, and tuning. For matrix addition we follow these steps: For double and double complex matrices (as well as for single and single complex, not presented here), we probe the performance with different loop-unrolling; that is, we exploit and test different register allocation policies. In fact, MA is a routine with 2-level nested loops and we unroll the inner loop. For each loop unrolling, we tested the performance for a different number of threads as explained in Section 3.2; that is, we split the computation as a sequence of MAs function calls, one for each thread. In this fashion, we optimize the parallel MA (which is in the critical path of the computation).

Strassen and Winograd algorithm installation. For each architecture and problem type (e.g., double or double complex), we determine the recursion point for the Winograd's algorithm; that is, we determine the problem size when the Winograd's algorithm must yield to the classic implementation of MM (GotoBLAS or ATLAS). We use this recursion point for all fast algorithms, even for the Strassen's algorithm. Of course, this is not optimal for Strassen's algorithm, which should yield control to the classic MM for larger recursion points because the algorithm requires more additions. Furthermore, if the pipeline of MA and MM is beneficial, we could exploit a smaller recursion point (and thus better performance).

Performance Measure. In this section, we measure performance by **normalized giga floating point per second (Normalized GFLOPS)**. This measure is the ratio between the number of operations, which we fixed to $2n^3$ (where n is the matrix size), and the execution time (wall clock). We choose the standard number of operation $2n^3$

for three reasons (even though fast algorithms perform fewer operations): First, this sets a common bound that is well known in the field; second, we can compare easily the performance for small and large problems; third, we can use it safely for comparing performance across algorithms —i.e., with different operation numbers and thus we compare execution time— and architectures —i.e., specific architecture throughput.

We measure wall-clock time when the application is computed with cold caches (just once) or with hot caches (the average for a set of runs, at least two for very large sizes) as we describe in the following code and explanation:

Average execution time.

```
// see code  mat-mulkernels.h
#define TIMING(X,time, interval) { int i,j; \
/* 1*/    j=1; \
/* 2*/    X; \
/* 3*/    do { \
/* 4*/      j*=2; \
/* 5*/      START_CLOCK; \
/* 6*/      for (i=0;i<j;i++) { X;  } \
/* 7*/      END_CLOCK; \
/* 8*/      time = duration/j; \
/* 9*/      printf(" average %f\n",time); \
/*10*/    } while (duration<interval); \
/*11*/  }

// see code example.3.c
#define MULINTERVAL 10

#ifdef MARCO_TEST
  TIMING(CMC(c, =, a,BMOWR , b),time_mul,MULINTERVAL);
#endif
```

The TIMING macro takes three operands: the matrix multiplication algorithm (i.e., the application we want to measure the execution time), the time variable where we will store the measure, and the minimum interval of time we require to run the application.

First, notice that we run the application once without timing, to warm up the system (line 2), then we run the application twice (line 6). If the duration is less than the minimum interval (line 10), we double the number of times we run the application and we repeat the process. We will use the average execution time.

We used an interval between 10–45 seconds (as a function of the architecture, 10 seconds for fast architecture such as the Nehalem, 45 seconds for slow ones such as the Pentium) as minimum interval. Of course, this can be tailored further to any specific need. This will assure that for small size problems we have a representative measure, for large problems, this is still a reasonable estimate of the execution time (the applications will run at least 10 seconds on machines where they will perform at least 20 Giga operations, if not 150, and thus a 10% improvement translates into 1 second or 2 Giga operations). For this paper and for the large problem sizes, we measured execution times of the order of minutes ($\sim$ at least 10 seconds improvement, easy to measure without considerable error).

We repeat here that we measured cold execution time (just one run) and the average execution time (average over at least two runs), and we presented the best of the two. Furthermore, this process has been repeated in case we found outliers and exceptions (for all codes and especially for older architectures).

Remark. It must be clear that we used a reasonable approach to measure execution time, we also provided a review and retrial of the experiments to provide reasonable and representative measure of execution time, and, in summary, we followed the sensible steps that other research groups already are taking to collect performance in the field.

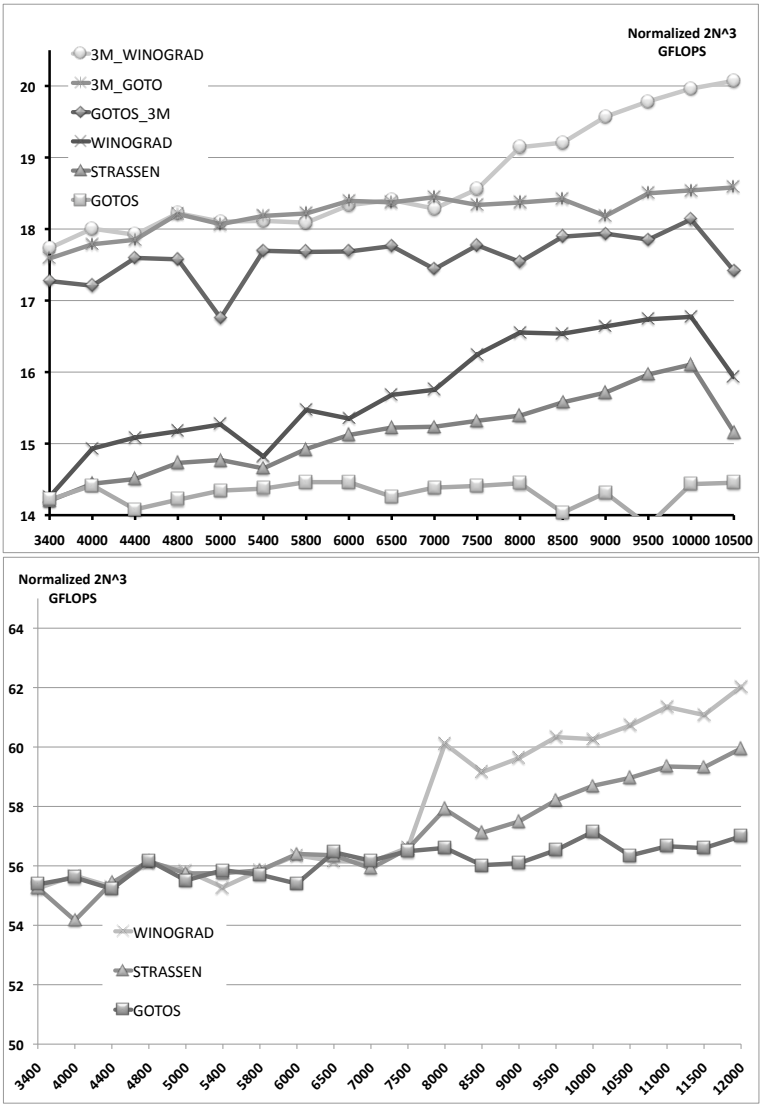

Fig. 2. 4-core 2 Xeon processor: Complex (top) GotoBLAS ZGEMM 14 GFLOPS, ZGEMM_3M 18 GFLOPS and our 3M_WINOGRAD 20 GFLOPS; and double precision (bottom) GotoBLAS DGEMM 56 GFLOPS and WINOGRAD 62 GFLOPS. Function pipelining does not provide significant improvements

6.2. Abbreviations

In the following sections and figures, we use a consistent but not truly standardized conventions in calling algorithms, we hope this will not be a major problem and this section should be used whenever consulting a performance plot.

— **STRASSEN**: Strassen's algorithms as in Table IV.
— **WINOGRAD**: Winograd's algorithm as in Table III.
— **WOPT**: Winograd's algorithm as in Table V with fewer MAs.
— **WIDEAL**: Winograd's algorithm as in Table VII optimized for a pipeline execution (but not pipelined).
— **GOTOS**: MM implementation as available in GotoBLAS.
— **BLAS MM or MM** only: MM implementation row-by-column (this is used in the error analysis only).
— **(STRASSEN|WINOGRAD|WOPT|WIDEAL)_PIPE**: software pipeline implementation of Strassen–Winograd algorithms as in Table IV, III, and VII where some MMs and MAs are interleaved.
— **GOTOS_3M**: 3M algorithm as available in GotoBLAS where matrices are stored as complex matrices.
— **3M_(GOTOS|WINOGRAD|STRASSEN|ATLAS)**: our implementation of the 3M algorithm as presented in Table II, where MM is implemented as STRASSEN, WINOGRAD, GOTOS, or ATLAS and thus complex matrices are stored as two distinct real matrices.
— **3M_(WINOGRAD|STRASSEN)_PIPE**: our implementation of the 3M algorithm as presented in Table II, where MM is implemented as STRASSEN_PIPE, WINOGRAD_PIPE and thus there is software pipelining between MMs and MAs, and complex matrices are stored as two real matrices.

6.3. Double-Precision and Complex Matrices

We divide this section into two parts: where software pipelining does not provide any improvement (actually is detrimental and not shown) and where software pipelining provides performance improvement. Notice that we postpone the experimental results for the optimized Winograd's algorithm as presented in Section 4.4 in the following experiment section (Section 6.3.1), where we present an in depth analysis.

No Software Pipelining. In Figure 2, we show the performance for at least one architecture where software pipelining does not work. Notice that the recursion point is quite large: N=7500 in Figure 2, with a small speed up (up to 2–10%) for double precision matrices. The performance improvements are more effective for complex matrices: smaller recursion point (i.e., we can apply fast algorithm for smaller problems) and best speed up (i.e., faster).

Software Pipelining. In Figures 3–5, we present the performance plots for three architectures where software pipelining offers performance improvements, we will give more details in Section 6.3.1.

For only one architecture, 2-core 2 Xeon (Pentium based) Figure 3, the 3M fast algorithms have a speedup of 4/3 (+25%) achieving a clean and consistent performance improvement. For this architecture, Goto's MM has best performance when using only two threads (instead of four), and thus this system is underutilized. Using fast algorithms and our scheduling optimizations we improve performance consistently.

For the Shanghai system, Figure 4, Goto's MM achieves peak performance using all cores, so this architecture is fully utilized. Nonetheless, fast algorithms and optimizing schedules achieve a consistent performance improvement. Software pipelining

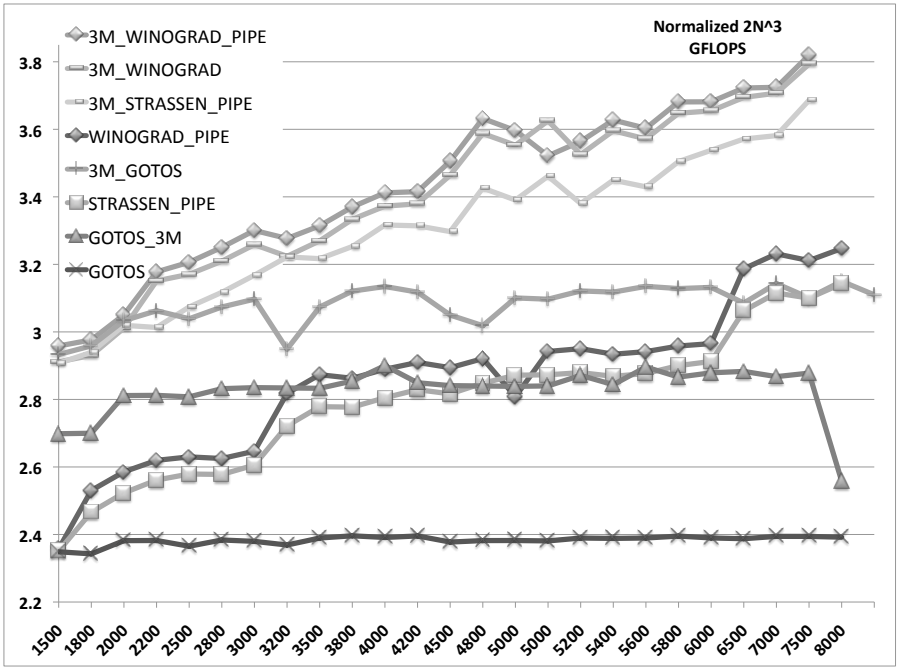

Fig. 3. 2-core 2 Xeon processor (Pentium based): Complex double precision with peak performance GotoBLAS GEMM 2.4 GFLOPS, GEMM_3M 2.8 GFLOPS, and our 3M_WINOGRAD_PIPE 3.8 GFLOPS

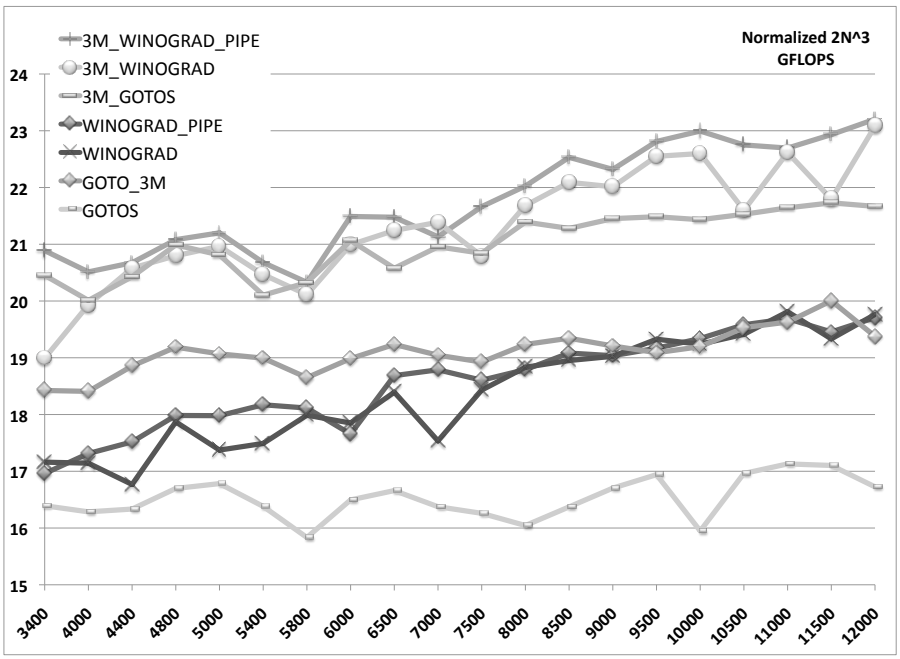

Fig. 4. 4-core 2 Opteron processor (Shanghai): Complex double precision with GotoBLAS peak performance ZGEMM 17 GFLOPS, ZGEMM_3M 20 GFLOPS, and our WINOGRAD 19 GFLOPS and 3M_WINOGRAD_PIPE 23 GFLOPS

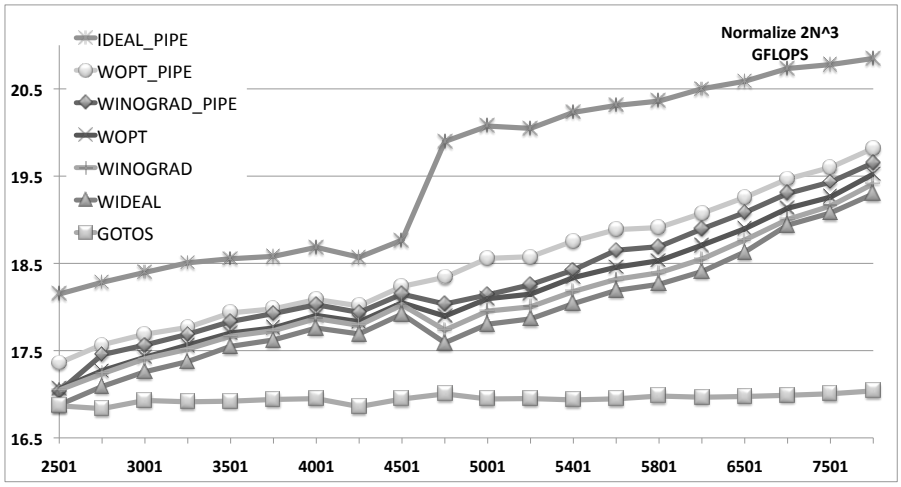

Fig. 5. 8-core 2 Xeon processor (Nehalem): Complex double precision with GotoBLAS peak performance ZGEMM 17 GFLOPS, our WINOGRAD 19 GFLOPS and WIDEAL_PIPE 20.5 GFLOPS

exploits the good bandwidth of the system and, even though MAs are competing for the resources against the MM, the overhead is very limited.

For the Nehalem system, Figure 5, we have very good speed ups (see the following section). This architecture has 4 physical cores for each processor and the architecture is designed to provide task parallelism; each physical core can execute in parallel 2 threads for a total of 8 virtual cores providing task as well as thread parallelism; this is the number of cores the operating system believes exist. Such architecture provides the best performance overall, just shy of 70 GFLOPS in Double precision, but also the best scenario for our optimizations.

6.3.1. Double precision: Software Pipelining and Optimized Algorithm. In the previous section, we presented the performance for the MM base line (e.g., GotoBLAS), Winograd, and Strassen with and without function software pipelining. In this section we focus on the optimized Winograd algorithms (Section 4.4) and the effect of (function) software pipelining (with respect to the Winograd's algorithm without software pipelining). This comparison will highlights the performance advantages of our pipelining optimizations (achieving ideal speedup w.r.t. the GEMM because the MA have not weight nor contribution).

We consider three architectures: 2-core 2 Xeon (Pentium Based), 8-core 2 Xeon (Nehalem), and 2-core 2 Opteron 270 (x86_64). The former two architectures are friendlier to software pipelining than the latter one. The first two architectures are underutilized because we can achieve the best performance with, respectively, two cores and 8 virtual cores idle. The latter architecture should provide a smaller space for improvement (if any) because there is no idle core. All architecture will provide information about the algorithms, the optimizations, and the overall performance.

We present odd problem sizes (e.g., $N = 2001, 3001$ etc.) because the optimized WOPT algorithm has potentially fewer MAs and smaller subproblems. We want to quantify such a performance advantage. In this section, we present experimental results for double (Figure 6–7), and double complex matrices (Figure 9–10). Especially, we present the performance advantage of the WIDEAL algorithm (the algorithm that has no MA in the critical path).

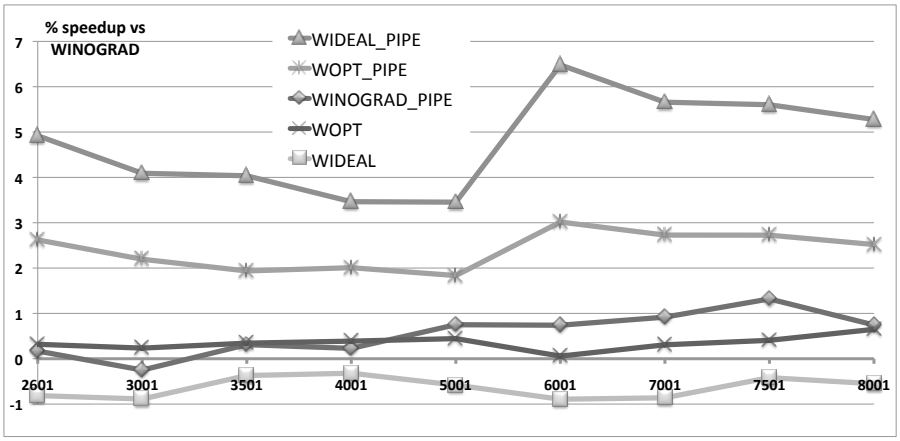

Fig. 6. 2-core 2 Xeon processor (Pentium Based) double precision: peak speedup WIDEAL_PIPE 7% w.r.t. WINOGRAD

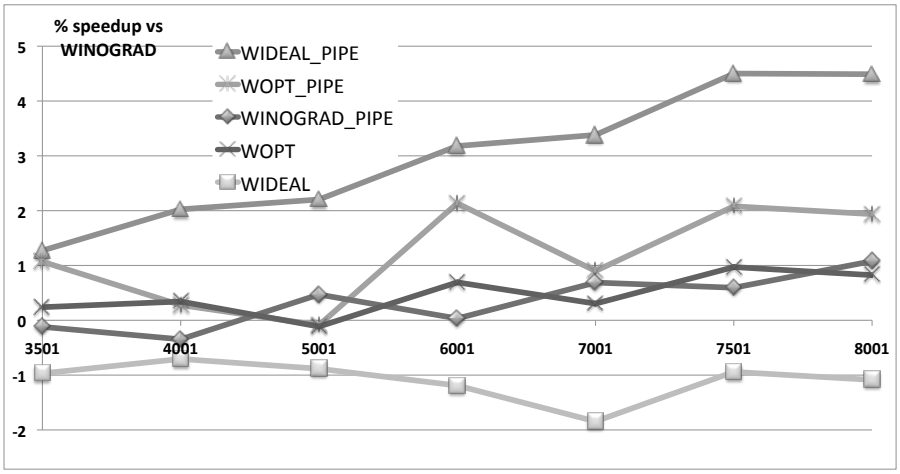

Fig. 7. 2-core 2 Opteron (x86_64) double precision relative: peak speedup WIDEAL_PIPE 4% w.r.t. WINOGRAD

For double precision and for the WIDEAL_PIPE algorithm (Figure 6, 7, and 8), we achieve 6% speed up for the Pentium based system, we achieve 4% for the Opteron system, and we achieve 10% for the Nehalem system.

For double complex matrices (Figure 9, 10, and 11), we have better relative improvements: we achieve 11% speed up for the Pentium based system, 7% for the Opteron, and 12% for the Nehalem (15% if we can perform 3 recursion levels). We show that WOPT (winograd's with fewer MAs) has a performance advantage (for these architectures) mostly because of the compounding effects of saving operations.

It is clear from the performance plots, that the WIDEAL algorithm has a performance advantage only when combined with our scheduling optimizations (WIDEAL_PIPE), otherwise WIDEAL is always slower than any other algorithms.

Notice that the experimental results for the Nehalem architecture follows the expected seesaw performance as the problem size increases and the recursion number

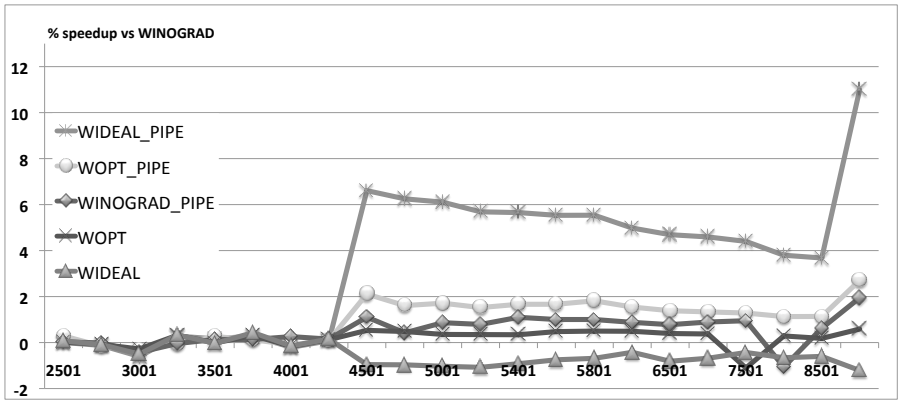

Fig. 8. 8-core 2 Xeon (Nehalem) double precision: peak speedup WIDEAL_PIPE 11% w.r.t. WINOGRAD

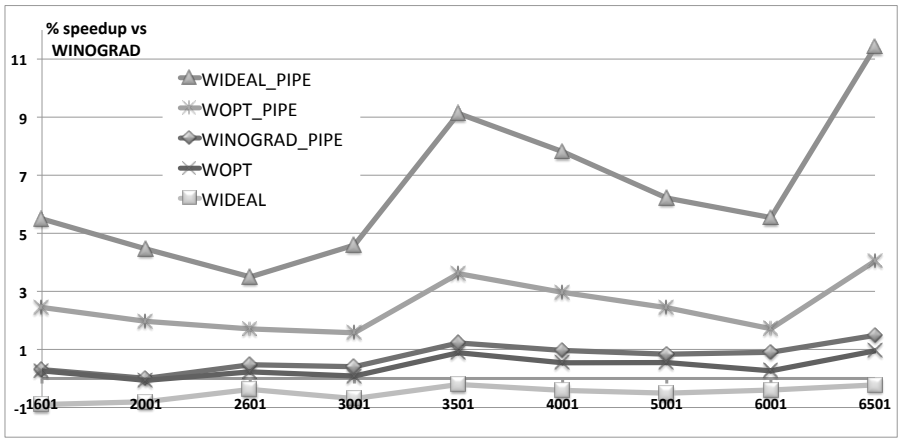

Fig. 9. 2-core 2 Xeon processor (Pentium Based) double complex precision: peak speedup WIDEAL_PIPE 11% w.r.t. WINOGRAD

increases, which follows roughly the formula $2^{r-1}\gamma/N$ (where N is the problem size and r is the number of recursions (i.e., see Section 4.5 Equation 8).

We know that for the Opteron-based system, there is not idle core and thus our approach will allocate two or more threads onto a single core. Using Goto's GEMM, we are achieving close to 95% utilization of the cores and thus there should be a very little space for improvements. In practice, hiding MAs provides little performance advantage for the WINOGRAD and WOP algorithm, however, there is quite a speed up in combination with WIDEAL (i.e., 5–7%).

6.4. Error analysis

In this section, we want to gather the error analysis for a few algorithms (i.e., fast algorithms), with different low-level optimizations and hardware operations, using either ATLAS or GotoBLAS, for single complex and double complex matrices, and for matrices in the range $|a| \in [0, 1]$ (probability matrices) and $|a| \in [-1, 1]$ (scaled matrices). What we present in the following figures can be summarized by a matrix norm:

$$\|C^{alg} - C^{DCS}\|_\infty = \max |c_{i,j}^{alg} - c_{i,j}^{DCS}|. \tag{9}$$

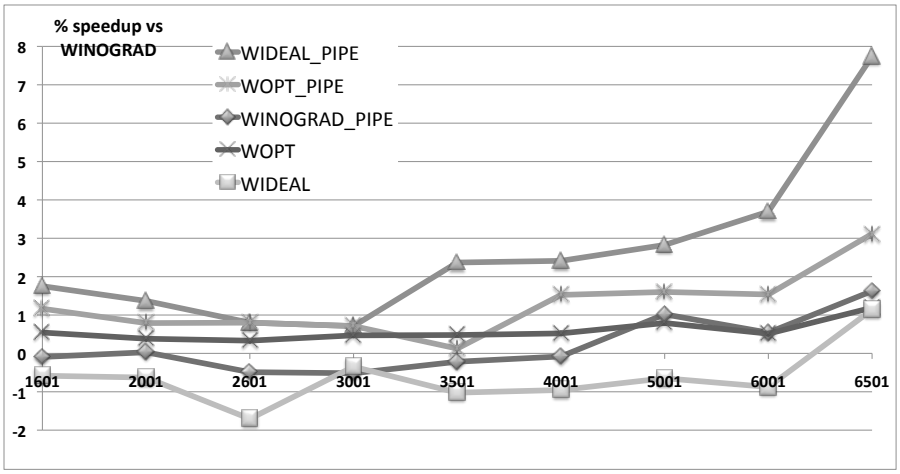

Fig. 10. 2-core 2 Opteron double complex precision relative: peak speedup WIDEAL_PIPE 8% w.r.t. WINOGRAD

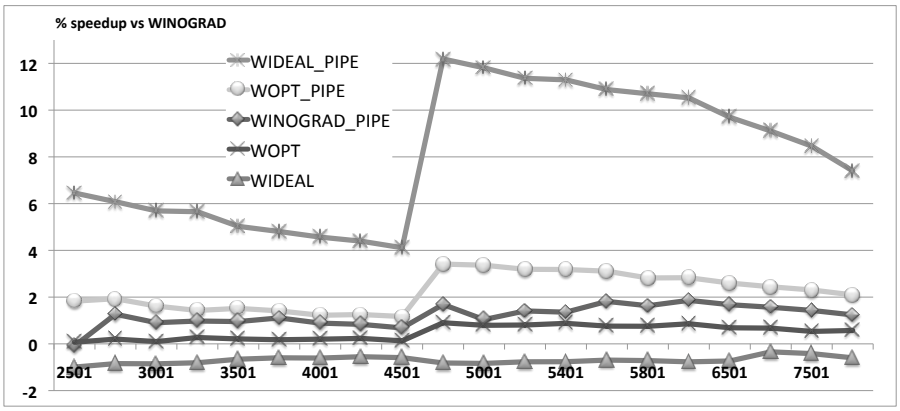

Fig. 11. 8-core 2 Xeon (Nehalem) double complex precision: peak speedup WIDEAL_PIPE 12% w.r.t. WINOGRAD

That is, we present the maximum absolute error. We investigate the error for complex matrices —thus we can compare all algorithms at once— but instead of showing the absolute maximum error for the complex matrix we present the maximum error for the *real* and *imaginary* parts of the matrices:

$$\left(\|Re(C^{alg}) - Re(C^{DCS})\|_\infty, \|Im(C^{alg}) - Im(C^{DCS})\|_\infty\right) \tag{10}$$

We will show that, in practice and for these architectures, the Winograd's based algorithms have similar error than the 3M algorithms and comparable to the BLAS classic algorithm. Furthermore we investigate the effect of the different schedules for the Winograd's-based algorithms. Notice, pipeline optimizations do not effect the error because we do not change the order of the computation.

We present a large volume of data, see Figure 12 and and 13.

Why two libraries have different error. GotoBLAS and ATLAS have different way to tailor the code to an architecture. This will provide different tiling techniques.

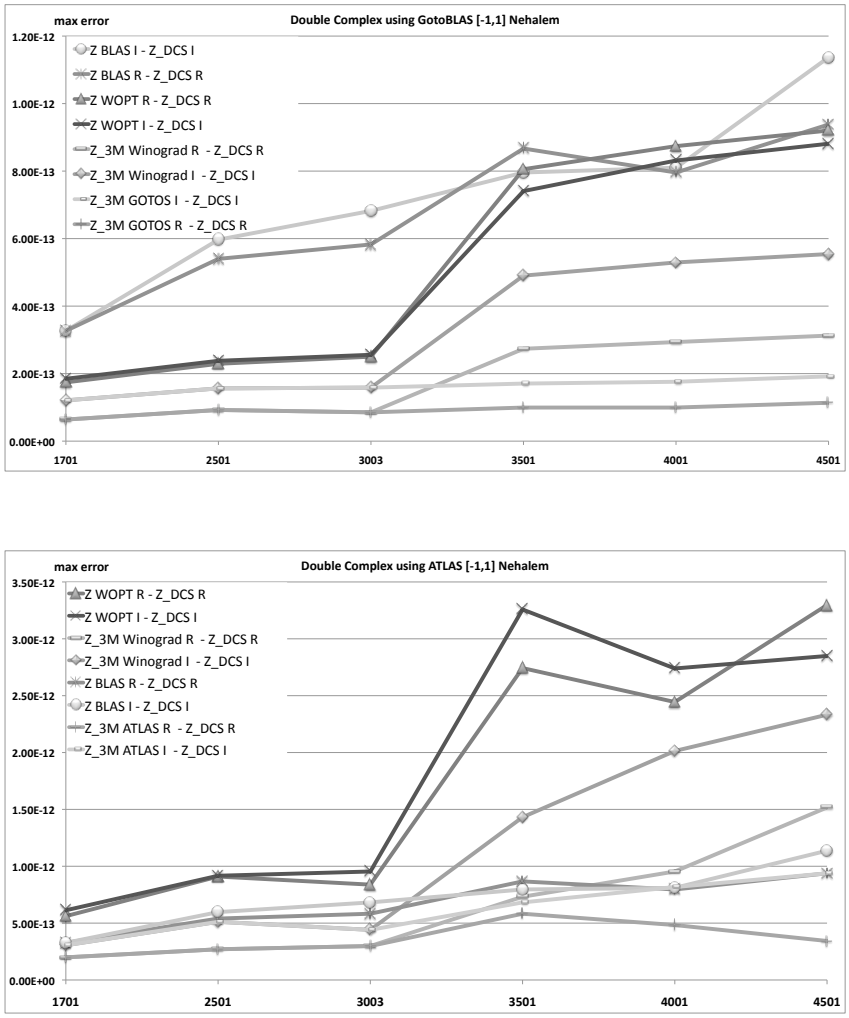

Fig. 12. 8-core 2 Xeon Nehalem Error Analysis [-1,1] with respect to the DCS algorithm: (top) based on GotoBLAS and (bottom) based on ATLAS. The GotoBLAS based implementation is 3 times more accurate than ATLAS based implementation. The GotoBLAS based WOPT (and thus WIDEAL) implementation has the same accuracy as the row-by-column definition BLAS implementation. Our 3M implementation has the same accuracy of GotoBLAS GEMM_3M.

For example, GotoBLAS exhibits a better accuracy (probably because the tiling size is larger). [3]

[3]We reached such a conclusion by our personal communications with the GotoBLAS' authors and, intuitively, because a larger tile may provide a better register reuse, thus the computation will exploit the internal extended precision of the register file, 90bits, instead of normal encoding in memory using 64bits; that is, inherently a better precision that will fit what we observed and it is against the intuitive idea that a larger tile will provide a longer string of additions thus a larger error.

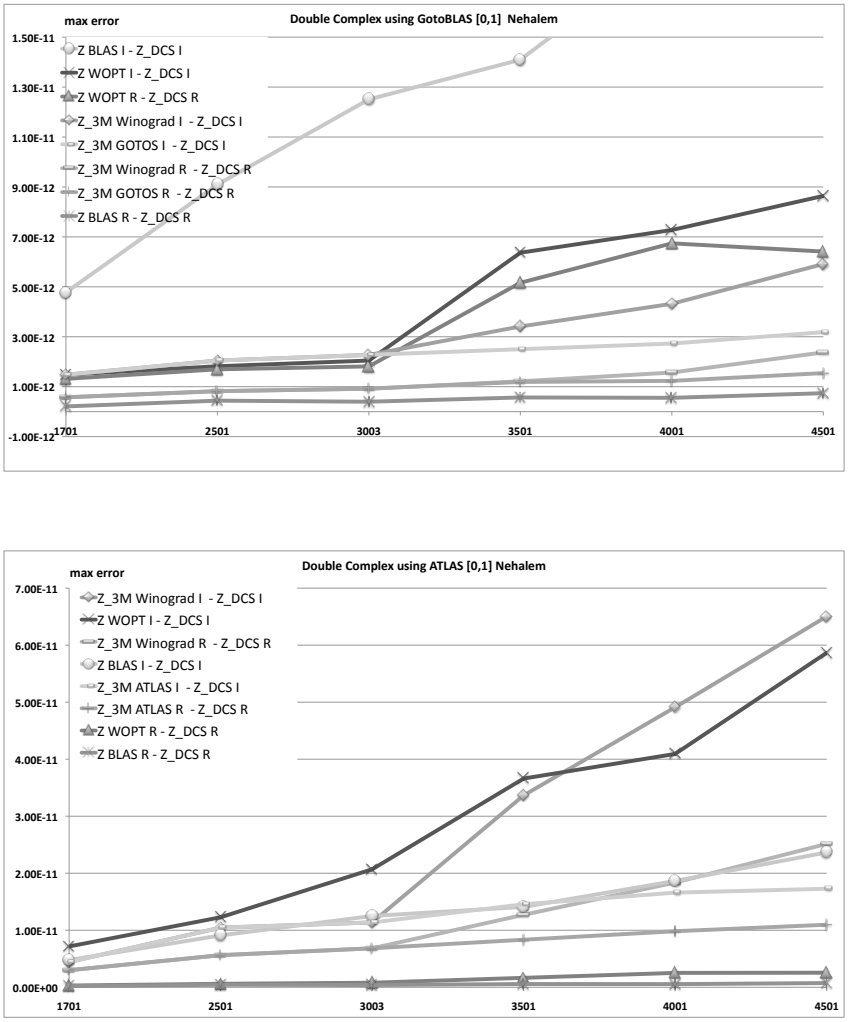

Fig. 13. 8-core 2 Xeon Nehalem Error Analysis [0,1] with respect to the DCS algorithm: (top) based on GotoBLAS and (bottom) based on ATLAS

We want to show that the relative errors (between fast and standard algorithm) are consistent across BLAS library installations. We show that a better accuracy of the kernels (GotoBLAS GEMMs) will allow a better accuracy of the fast algorithms as significant as a factor of 3 (1/3 of a digit).

Why comparing real and imaginary parts. We show that the error is different for the real and the imaginary part of the computation for several algorithms. In particular, we show that the 3M algorithm tends to have larger error on the imaginary part than the real part (which is known in the literature). However, we may notice that the 3M-Winograd variant has the same error as the BLAS GEMM (row-by-column) computation (and it is two order of magnitude faster than the BLAS GEMM). So the

application of the Winograd algorithm, alone or in combination with the 3M algorithm, has a reasonable error and it should not be discarded a priori.

Why different matrix element ranges. First, the sign of the matrix element affects the precision of any operation (cancellation of similar numbers) and the accuracy of the algorithm. It is known in the literature that Winograd's algorithm is in general stable for probability matrices ($|a| \in [0, 1]$). We show that variation of the Winograd's algorithm such as the one with fewer additions may actually loose accuracy with little performance advantage. The range of the matrix is chosen such that all the error upper bounds can be expressed as a polynomial of the matrix size (and there is no need of norm measure).

We present our findings in Figure 12–13. We believe, this is the first attempt to show how in practice all these algorithms work and how they affect the final result and error. We do not justify the blind use of fast algorithms when accuracy is paramount; however, we want to make sure that fast algorithms are not rejected just because of an unfounded fear of their instability.

In practice, it's prohibitively expensive to provide statistics based on DCS algorithm (described in Section 5.1): to gather the experimental results presented in Figure 12–13 took about 2 weeks. To collect, say ten points, it will take about 20 weeks. First, it is not really necessary. Second, if we take a weaker reference (i.e., an algorithm with better accuracy because it uses better precision arithmetic double precision instead of single precision), then we may estimate the statistics of the average error and show that there is no contradictions (with the result presented in the paper Figure 12–13) and in the literature.

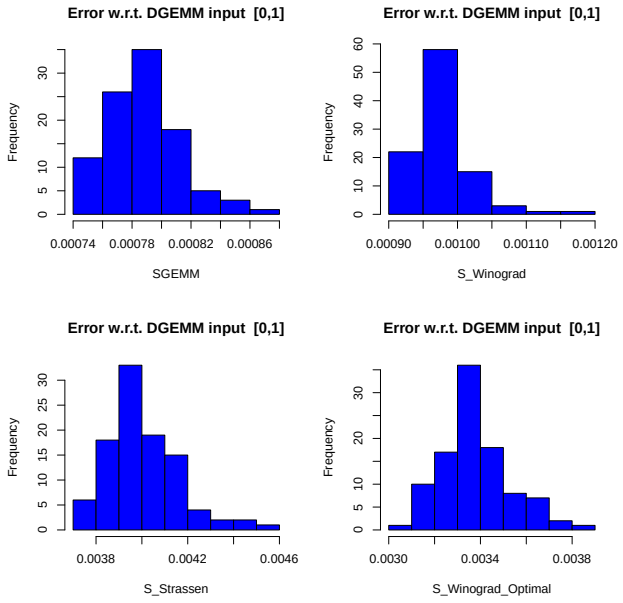

Fig. 14. 2-core 2 Opteron, problem size N=6500, absolute error statistics w.r.t. DGEMM for matrices with input [0,1]: distribution of the maximum error for GEMM, WINOGRAD, STRASSEN, and WOPT/WIDEAL in single precision w.r.t. DGEMM by 100 runs. WINOGRAD is 3 times more accurate on average that WOPT and 0.3 times less accurate than SGEMM

Consider the DGEMM as the reference (more precise arithmetic but no more accurate algorithm than the DCS algorithm in double precision) and we compare the error committed by the SGEMM, Winograd's, Strassen's and the optimized Winograd's algorithm all computed in single precision. What we can measure is the statistic of the fast algorithms w.r.t. the more accurate DGEMM. We run 100 MMs and we collected the maximum absolute error for matrices of sizes $N = \{3500, 4000, 4500, 5000, 5500, 6000, 6500\}$, we report here only for 6500, see Figure 14–15.

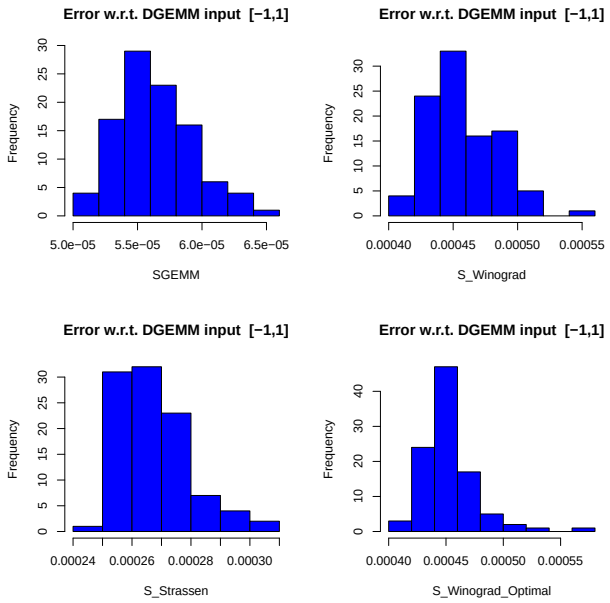

Fig. 15. 2-core 2 Opteron (x86_64) N=6500 absolute error statistics w.r.t. DGEMM for matrices with input [-1,1]: distribution as above. WINOGRAD and WOPT have on average the same error and WINOGRAD is 10 times less accurate than SGEMM (1 digit) and STRASSEN is 5 times less accurate than SGEMM (1/2 digit)

First, the maximum of the maximum error and the average measure of the maximum error have the expected behavior and there is not evidence of any contradicting results w.r.t. to the one already presented in this paper. Second, the standard deviation of the error is relatively small (the range of the error), thus the inference about the error we commit base only on a single trial as we do in this paper, may be off for a fraction of a significant digit, it confirms the already published results, and our estimation. Third, the statistic confirm a difference in the error between the version of Winograd's algorithms (in the paper we distinguish them as WINOGRAD and WOPT where we reduce the number of additions) where we show that reducing further the number of MAs increases the maximum error for probability matrices.

7. CONCLUSIONS

We investigated the performance of fast algorithms such as 3M/Winograd/Strassen for general purpose SMP systems (common in search engine data centers). We show that fast algorithms can be always applied (achieving 5–25% improvement) but more importantly they present a family of matrix computation algorithms where very interesting and useful optimizations can be applied (further 2–15% performance improvements).

On one side, there is no dominant algorithm for all architectures and all problem sizes. This performance variety should not undermine our overall message: fast algorithms can be used in combination with classic MM algorithms and all will thrive when used together. On the other side, a few researchers may see the same variety as discouraging because there is no clear dominant algorithm: it would be easier to rely on a single common solution.

Here we have shown, that our algorithms and optimizations are simple to apply and they extend the performance of the fastest BLAS libraries for the state of the art architectures.

Acknowledgments.

The first authors would like to thank Yahoo! to provide the opportunity, to give the time, and to provide the machines for these experiments. In particular, an heartfelt thank goes to Ali Dasdan, Kelving Fong, Jeff Harlam, Joel Carceres, Greg Ulrich and Arun Kejawal (Yahoo!), Kazushige Goto (Microsoft) and Robert van de Geijn (UT Austin), Clint Whaley (UT San Antonio), and David Wise (Indiana University).

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.

BILMES, J., ASANOVIC, K., CHIN, C., AND DEMMEL, J. 1997. Optimizing matrix multiply using PHiPAC: a portable, high-performance, Ansi C coding methodology. In *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 Transaction in Mathemathical Software 28,* 2, 135–151.

BODRATO, M. 2010. A Strassen-like matrix multiplication suited for squaring and higher power computation. In *ISSAC '10: Proceedings of the 2010 international symposium on Symbolic and algebraic computation*. ACM, New York, NY, USA. http://bodrato.it/papers/#ISSAC2010.

BOYER, B., DUMAS, J.-G., PERNET, C., AND ZHOU, W. 2009. Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm. In *ISSAC*. 55–62.

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.

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 19-th annual ACM conference on Theory of computing*. 1–6.

D'ALBERTO, P. AND NICOLAU, A. 2007. Adaptive strassen's matrix multiplication. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*. ACM, New York, NY, USA, 284–292.

D'ALBERTO, P. AND NICOLAU, A. 2009. Adaptive Winograd's matrix multiplications. *ACM Transactions on Mathematical Software 36,* 1.

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 Transaction in 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 Transaction in 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.

DUMAS, J.-G., GIORGI, P., AND PERNET, C. 2008. Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages. *ACM Trans. Math. Softw. 35,* 3, 1–42.

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

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, 422–455.

HIGHAM, N. 1990. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Trans. Math. Softw. 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 *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*. ACM Press, 32.

JÁJÁ, J. 1992. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

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, 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, 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. *Theor. Comput. Sci. 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 Transaction in Mathemathical Software 5*, 308–323.

LOOS, S. AND WISE, D. Strassen's matrix multiplication relabeled.

NICOLAU, A., LI, G., AND KEJARIWAL, A. 2009. Techniques for efficient placement of synchronization primitives. In *PPoPP '09: 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

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

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

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.

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, 101–121. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.