CS-1998-06

Architecture-efficient Strassen's Matrix Multiplication: A Case Study of Divide-and-Conquer Algorithms

V. Paúl Pauca Xiaobai Sun Siddhartha Chatterjee
Alvin Lebeck

Department of Computer Science,
Duke University, Durham, NC 27708.

May 6, 1998

Architecture-efficient Strassen's Matrix Multiplication: A Case Study of Divide-and-Conquer Algorithms

V. Paúl Pauca * Xiaobai Sun * Siddhartha Chatterjee †
Alvin Lebeck *

May 7, 1998

Abstract

Many fast algorithms in arithmetic complexity have hierarchical or recursive structures that make efficient implementations on high performance computers with memory hierarchies non-trivial. In this paper we present our findings on efficient implementation of Strassen's algorithm[17] for the ubiquitous operation of matrix multiplication as a model for a class of recursive algorithms. In comparison to the conventional multiplication algorithm, Strassen's algorithm requires more storage space and exhibits poorer data locality. Although recent years have seen better representation and better implementations of the algorithm, the characterization of the optimization in implementation issues and hence the automatic optimization strategies are lacking. We present our schemes for optimizing data locality and reducing storage space in an increased scope of computation sequences with arithmetic and numerical constraints. Moreover, our characterization of the optimization schemes are based on the use of algorithmic operators and leads to automatic optimization for a class of recursive algorithms. We present experimental results and comparisons with the corresponding subroutine in BLAS and the recursive version of Strassen's algorithm.

1 Introduction

Strassen's algorithm [17], among other bilinear algorithms [10], for matrix multiplication is a classic representative of hierarchical algorithms which are challenging in the aspect of memory-efficient implementation and compiler-aided optimization. Its arithmetic cost is $O(n^{\log_2 7})$ for the product of two $n \times n$ matrices, in comparison to $O(n^3)$ for the conventional algorithm, but its implementation on computers with hierarchical memory is not well

*Department of Computer Science, Duke University, Durham, NC 27708.
†Department of Computer Science, CB #3175, Sitterson Hall, The University of North Carolina Chapel Hill, NC 27599-3175.

characterized and optimized as that for the conventional algorithm [5]. In this paper we use Strassen's algorithm as a model to identify and characterize issues relevant to automatic code generation and optimization of a large class of hierarchical algorithms that include FFT algorithms [12, 16, 15] and multipole or multipole-like algorithms[3] used in various application problems. The state of the art for implementing Strassen's algorithm and multipole algorithms is that end-users manually implement locality enhancements on an *ad hoc* basis, with few formal models to follow.

Strassen's algorithm and some of its variants [7] have lately been investigated in their own right as serious and competitive alternatives to the conventional algorithm in high performance codes for matrix multiplication. See [1, 4, 6] and the recent work in [11]. Most existing implementations are based on the recursive version of Strassen's algorithm. These address issues such as padding and dynamic peeling for extension to rectangular matrices, memory reuse for intermediate results, and determination of the crossover point between Strassen's algorithm and the conventional algorithm on a given architecture. The orderings considered are basically the depth-first ordering and the breath-first ordering of the product tree or their combinations [13]. With depth first ordering, the sequence for computing the seven products is determined by heuristic strategy for storage reuse. An important result is an algebraic representation of the algorithm involving recursive Kronecker products [13], which leads to many program generation strategies and to non-recursive implementation for vector computations.

In this paper we view Strassen's algorithm under a different light. First of all, the memory architecture of interest is hierarchical [9]. On computers with hierarchical memory, the data locality of an algorithm affects the cache performance significantly [9, 14]. Therefore, our optimization objects are mainly data locality enhancement and working storage reduction. For data locality analysis and improvement, we present a formal description of temporal data locality at the algorithm level and provide systematic strategies to find optimal computation sequences with respect to both temporal data locality and storage space. We note that the best algorithmic data locality can be explored or undermined to different degrees by a given memory architecture, feasible data layouts and computation sequences at program level. Secondly, we point out that the optimization in data locality is not an optimization problem without constraints. For example, better cache performance may come from redundant operations. We present arithmetic and numerical constraints on the optimization. Finally, our formal characterization of the optimizations for Strassen's leads to compiler-aided automatic optimization and can extend to a class of recursive algorithms.

The remainer of this paper is organized as follows. In §2 we review the conventional description of Strassen's algorithm and describe our main idea to improve data locality. In §3 we introduce the operator representation of Strassen's algorithm and present an optimization framework for deriving information from the algorithm operators and characterizing optimization strategies. Based on the framework, we give in §4 non-recursive variants of Strassen's algorithm. Preliminary experimental results on the IBM RS6000, DEC Alpha and Pentium II architectures are presented in §4.2.

2 Strassen's Algorithm

Strassen's algorithm computes the product of two matrices of order n in less than $4n^{\log_2 7}$ arithmetic operations, compared to the $2n^3$ arithmetic operations of the conventional algorithm. Strassen's algorithm is usually described in a divide and conquer form. Let A and B be two $n \times n$ matrices, where n is an even integer. Partition A, B and $C = A \times B$ into four $n/2 \times n/2$ blocks as

$$\begin{pmatrix} C_1 & C_3 \\ C_2 & C_4 \end{pmatrix} = \begin{pmatrix} A_1 & A_3 \\ A_2 & A_4 \end{pmatrix} \begin{pmatrix} B_1 & B_3 \\ B_2 & B_4 \end{pmatrix}. \tag{1}$$

Then the four submatrices of C can be computed as follows,

$$\begin{array}{ll} & P_1 = (A_1 + A_4) \cdot (B_1 + B_4) \\ C_1 = P_1 + P_4 - P_5 + P_7 & P_2 = (A_2 + A_4) \cdot B_1 \\ C_2 = P_2 + P_4 & P_3 = A_1 \cdot (B_3 - B_4) \\ C_3 = P_3 + P_5 & P_4 = A_4 \cdot (B_2 - B_1) \\ C_4 = P_1 + P_3 - P_2 + P_6. & P_5 = (A_1 + A_3) \cdot B_4 \\ & P_6 = (A_2 - A_1) \cdot (B_1 + B_3) \\ & P_7 = (A_3 - A_4) \cdot (B_2 + B_4). \end{array} \tag{2}$$

We refer to the ten additions preceding the seven products P_j as *pre-additions* and to the eight additions afterwards as *post-additions*.

The seven products can be computed by recursive application of the above procedure. In general the recursion is switched to the conventional algorithm at some matrix size n_o at which Strassen's construction is no longer advantageous. This is the so-called *recursion truncation point* [11]. With respect to arithmetic operation complexity, the optimal n_o is approximately 16. However, actual runtime performance comparisons with the conventional algorithm give observed n_o values of at least one magnitude higher across different computer architectures [11].

The large discrepancy between the arithmetic and experimental values of the truncation point is typical to many hierarchical algorithms and reflects a poor mapping of the algorithmic hierarchy to a memory hierarchy. While the products are computed at the bottom of the recursion tree on blocks of size $n_o \times n_o$, the mapping quality of Strassen's algorithm is degraded by the pre- and post-additions of recursively changing block size down and up the recursion tree. This way of referencing data for pre- and post-adds on blocks of varying size contrasts sharply with the fixed cache block size and caching strategies on most contemporary computers [9]. In other words, the recursive computation of Strassen's algorithm exhibits poor temporal and spatial locality when performed across a system with memory hierarchy. Thus, our main idea is to express Strassen's algorithm with *all* operations on blocks of a uniform size and find computation sequences that render optimal spatial and temporal data locality as well as minimal storage requirements.

3 Optimization Framework

3.1 Operators of Strassen's Algorithm

The operations of Strassen's algorithm in (2) can be represented in matrix operators on block matrices with block size $n_o = n/2$ as follows, see references [10, 13],

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = W_{n_o}^{\mathrm{T}} \begin{pmatrix} P_1 \\ P_2 \\ \vdots \\ P_7 \end{pmatrix} = W_{n_o}^{\mathrm{T}} \left(U_{n/2} \begin{pmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{pmatrix} \diamond V_{n_o} \begin{pmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \end{pmatrix} \right), \tag{3}$$

where

$$X_{n_o} \stackrel{\mathrm{def}}{=} X \otimes I_{n_o}$$

is the Kronecker product of matrix X with the $n_o \times n_o$ identity matrix I_{n_o}, and $\diamond$ denotes the block array product, i.e., block i of $Z = X \diamond Y$ is the product of block i of X and block i of Y. The algorithmic operators U, V, and W are 7×4 matrices

$$U = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{pmatrix}, \quad V = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}, \quad W = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}. \tag{4}$$

It should be clear that U and V are the pre-add operators on the blocks of A and B, respectively, and W^{T} is the post-add operator on the computed seven products to the blocks of C. Notice that a reordering in the blocks of A, B, or C is equivalent to a permutation in the columns of U, V, or W, respectively. For consistency, we use the column-major ordering used in (1) throughout this paper.

3.2 An optimal ordering in $\{P_j\}$

The computation of the products P_j, $1 \leq j \leq 7$, involves accesses to and additions between blocks of A and B, respectively, as well as updates to blocks of C. In a serial implementation of Strassen's algorithm, the order in which the products $\{P_j\}$ are computed affects the overall performance. Among all 7! different orderings,

$$\{P_3, P_6, P_2, P_1, P_5, P_7, P_4\} \tag{5}$$

is optimal in temporal data locality, i.e. the total block reuse at each next computation step of the blocks of A, B, and C. The operator matrices in the permuted order are shown in

(6).

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}, V = \begin{pmatrix} 0 & 0 & 1 & -1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ -1 & 1 & 0 & 0 \end{pmatrix}, W = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}. \tag{6}$$

Data reuse can be observed from consecutive rows of the algorithmic operators. Specifically, a nonzero element at the (i, j) entry in either U, V, or W, followed by a nonzero at the $(i+1, j)$ entry indicates that the corresponding block A_j, B_j or C_j is used in consecutive steps i and $i+1$. Formally, total block data reuse at the next step can be characterized by the following formula,

$$\text{total\_reuse}(X) = \sum_{i=2}^{6} |X(i,:) \circ X(i+1,:)| e, \tag{7}$$

where $\circ$ denotes point-wise multiplication and $e = (1, 1, 1, 1)^T$. With the original ordering, total block data reuse on A is $\text{total\_reuse}(U) = 2$. In contrast, with the optimal ordering in (5), see (6), it is $\text{total\_reuse}(U) = 6$. Similar observations can be made on the block data reuse of B and C. It is not difficult to check that the maximum block reuse is 50%, and it is achieved by the optimal ordering for the three operator matrices in (6) simultaneously.

We note an interesting property of the optimal ordering in $\{P_j\}$. Each of the operator matrices in this ordering is *reflective* in the sense that the 4×4 bottom submatrix is the 4×4 top matrix with columns and rows in the reversal order. We will show later that this reflective ordering is also optimal in re-use of storage space for intermediate results with multi-level Strassen's algorithm. From now on, we assume the ordering in (5) for the operator matrices.

The algorithmic data reuse can not be realized if the block size is too big to allow sufficiently many blocks to be stored in the cache. For example, $P_6 = (A_2 - A_1) \cdot (B_1 + B_3)$ and is added to C_4. If the cache is not big enough for 6 blocks, then the algorithmic data locality is reduced. The loss of algorithmic data locality requires that the algorithmic data locality be tunable to architecture parameter(s), such as the cache size. Our tuning approach is to first express Strassen's algorithm with operations on the blocks of uniform size and then find computation sequences that are optimal with respect to temporal data locality and storage space.

3.3 Recursion Unfolding

To represent multi-level Strassen's algorithm in operations on blocks of uniform size, we define first a recursive ordering in the blocks of an array X in recursive 2×2 partition. If

$k = 1$, then

$$L[X] \stackrel{\text{def}}{=} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{pmatrix},$$

where X_i are the blocks of X in a 2×2 partition ordered in column major (1). Using this notation, equation (3) for the one-level Strassen's algorithm becomes

$$L[C] = W_{n/2}^{\mathrm{T}} \left(U_{n/2} L[A] \diamond V_{n/2} L[B] \right).$$

If $k > 1$, we define $L^{i+1}[X] = L[L^i[X]]$, recursively, with $L^0[X] = X$. The k-levels Strassen's algorithm can be mathematically expressed as the following operations on the blocks in a $2^k \times 2^k$ partition, see [13],

$$L^k[C] = (\bigotimes^k W^{\mathrm{T}})_{n_o} \left((\bigotimes^k U)_{n_o} L^k[A] \diamond (\bigotimes^k V)_{n_o} L^k[B] \right), \tag{8}$$

where $n_o = n/2^k$ is the block size at the truncation level k, and $\bigotimes^k X \stackrel{\text{def}}{=} X \otimes \cdots \otimes X$ with X appearing k times on the Kronecker products.

The mathematical representation of Strassen's algorithm given in (8) suggests a rich set of computation sequences for obtaining the result $L^k[C]$. Next, we show that optimization over this set must account for several key issues such as arithmetic cost, numerical error, and storage space.

3.4 Constrained Optimizations over Computation Sequences

The scope of mathematically equivalent computation sequences increases significantly with the number of recursion levels unfolded in the form (8). For illustration, we consider the case $k = 2$, the two-level Strassen's algorithm. The algorithmic representation in (8) describes two pre-add streams followed by a post-add stream,

$$\begin{array}{ll} \text{Pre-add stream on } A\text{:} & (U \otimes U)_{n_o} L^2[A] \\ \text{Pre-add stream on } B\text{:} & (V \otimes V)_{n_o} L^2[B] \\ \text{Post-add stream on } C\text{:} & (W \otimes W)_{n_o}^{\mathrm{T}} P_{(1:49)}, \end{array} \tag{9}$$

where $n_o = n/4$. There are a number of computation sequences for each of computation stream. For example, pre-add in A have three classes of candidate sequences,

$$\begin{array}{ll} (i) & (U \otimes U)_{n_o} L^2[A] \\ (ii) & (I_7 \otimes U)_{n_o} (U \otimes I_4)_{n_o} L^2[A] \\ (iii) & (U \otimes I_7)_{n_o} (I_4 \otimes U)_{n_o} L^2[A]. \end{array} \tag{10}$$

All three sequences compute the same block array of sums over blocks of A. Every sum in the array is computed as specified in a chosen sequence. Following convention, we refer to the right-most computation before the blocks of A, B, and C as the computation at level-1, and to left-most computation at the end of the 49 products as the computation at level-k. Similarly in what follows, *fine* partition will refer to the bottom $2^k \times 2^k$ partition, and *coarse* partition to the $2^{k-1} \times 2^{k-1}$ partition of A, B, and C. Pre-add are top-down operations and post-add are bottom-up operations. We discuss next the optimization with constraints in the increased scope of computation sequences.

The operator $U \otimes U$ of sequence (i) in (10) for pre-add in A merges the additions of two levels and exposes the dependency of each product P_j, $1 \leq j \leq 49$, on the blocks of A. It requires minimum temporary storage but invokes redundant computations. On the other hand, the operator of sequence (ii) or (iii) is decomposed into two factors (or two levels) and exploits the re-usability of common subexpressions (intermediate sums), but entails more storage space for intermediate sums.

We give the comparison in arithmetic complexity first. For each $X \in \{U, V, W^{\mathrm{T}}\}$, let the function adds$(X)$ be the number of additions X invokes, i.e. define

$$\mathrm{adds}(X) \stackrel{\mathrm{def}}{=} \bar{e}^{\mathrm{T}}(|X|e - \bar{e}),$$

where $|X|$ is X with each element replaced by its absolute value, and e and $\bar{e}$ are vectors with all elements equal to 1. It can be checked easily that adds$(U) = 5$. The operator in sequence (i) invokes

$$\mathrm{adds}(U \otimes U) = (\bar{e}^{\mathrm{T}}|X|e)^2 - (\bar{e}^{\mathrm{T}}\bar{e})^2 = 12^2 - 49 = 95,$$

additions over $n_o \times n_o$ blocks. We have for sequence (ii) and sequence (iii), respectively,

$$\begin{aligned} &\mathrm{adds}(I_7 \otimes U) + \mathrm{adds}(U \otimes I_4) = 7 \cdot 5 + 5 \cdot 4 = 55, \\ &\mathrm{adds}(U \otimes I_7) + \mathrm{adds}(I_4 \otimes U) = 55. \end{aligned}$$

The difference in arithmetic cost between the level-merging sequence and the level-wise sequences is evidently too big to ignore. As a matter of fact, the level-merging sequence leads to a transformed multiplication having more arithmetic operations than the conventional multiplication! The tie in arithmetic cost between sequence (ii) and sequence (iii) is not unsurprising. While sequence (ii) exploits the common expressions *across* a block in the coarse partition, sequence (iii) exploits the common expressions *within* a block in the fine partition, which may be intuitively expected to be better. Figure 1 illustrates common subexpressions across and within blocks of sequences (ii) and (iii) respectively for $k = 2$.

We are also concerned about the change in numerical property of the algorithm with the alternation of computation sequence. The level-wise operations specified in sequence (ii) and (iii) guarantee that all the sums are computed pairwise. The rounding error in a computed sum $e^{\mathrm{T}}X$, where X has n rows, by pairwise summation is bounded by $c \log_2(n) e^{\mathrm{T}}|X|$, which is sharper than the bound for non-pairwise summation, $c(n-1)e^{\mathrm{T}}|X|$. Since the operator

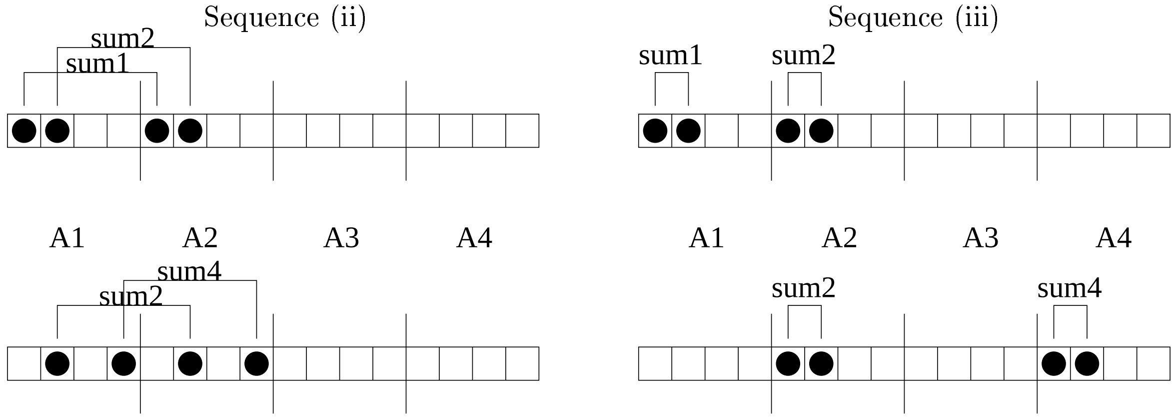

Figure 1: Sequence (ii): sum_2 (across the coarse partition) in $e_9(U \otimes U)L^k[A]$ is used again in $e_{9+1}(U \otimes U)L^k[A]$. Sequence (iii): sum_2 (within the coarse partition) in $e_9(U \otimes U)L^k[A]$ is used again in $e_{9+7}(U \otimes U)L^k[A]$.

$U \otimes U$ in sequence (i) does not specify a summation method to use, an implementation of the sequence may result in a much larger bound on the rounding errors in the sums. Based on the unfolding form (8), we can show that if the pre-add streams and the post-add stream are computed in level-wise sequence (ii) or (iii), the rounding error bound remain the same as the one given by Bini and Lotti [2] and Sun et al. [18], although the error propagation changes with the choice of a sequence for each stream and their combination. Notice that in the recursive version, the pre-add are computed as in sequence (ii) and the post-add are computed as in sequence (iii). The above analysis on arithmetic complexity and rounding error analysis tells us that for multiple-level Strassen's algorithm, a level-merge between any two levels results in more arithmetic operations and a larger rounding error bound.

3.5 Minimal Memory Requirement

The search for optimal computation sequences with respect to data locality and storage requirement for multi-level recursion unfolding is now confined to level-wise sequences. Notice first that the data reuse in the intermediate levels is also the reuse of common expressions (intermediate results). For example, for summations at each level $l+1$ from results at level

l with operator U, we have

$$\mathrm{sum}_{i,l+1} = e_i^{\mathrm{T}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \mathrm{sum}_{1,l} \\ \mathrm{sum}_{2,l} \\ \mathrm{sum}_{3,l} \\ \mathrm{sum}_{4,l} \end{pmatrix}, \quad i = 1:7. \tag{11}$$

The intermediate results require additional storage. The tradeoff with cost in memory space is a typical feature of arithmetically fast divide-and-conquer methods.

Since the intermediate results are used partially at any computation step, we consider the minimum memory requirement for computing and reusing the intermediate results. Notice that although the seven sums at level-l in (11) share the four sums at level l, $\mathrm{sum}_{1:4,l+1}$ do not reference $\mathrm{sum}_{3,l}$ and the sums $\mathrm{sum}_{4:7,l+1}$ do not reference $\mathrm{sum}_{2,l}$. In other words, $\mathrm{sum}_{2,l}$ needs not to be computed until $\mathrm{sum}_{3,l}$ is no longer referenced, and the memory for $\mathrm{sum}_{3,l}$ can be reused for $\mathrm{sum}_{2,l}$. In such ordering, it is sufficient to keep three sums instead of fours sums from the previous level. Formally, the potential in memory reduction can be explored by checking pairwise orthogonality among the columns of the operator matrix and exploited by a proper ordering in the sums at level $l+1$. For example, $U(:,2)$ and $U(:,3)$ are orthogonal and the memory reduction for sums with U is achieved by the reflective ordering (5). It can be also checked that keeping three blocks is also necessary to avoid repeated evaluations. By induction, at any step of computing sums with U at level l, the minimum memory requirement for common sums at levels $< l$ is 3^{l-1} blocks. Any requirement for less memory results in a tradeoff with repeated arithmetic operations. The analysis applies to sequence (iii), the difference is in the stride of accessing the blocks at each level. By using the reflective ordering for a k-level recursion unfolding implementation, we can achieve reduction in memory requirement for pre-add by a factor $(3/4)^k$ compared to the recursive version. Finally, since post-add depends on the availability of pre-add, see (3), the total storage requirement is minimized when the two pre-add streams take a same level-wise sequence and the post-add takes the alternative level-wise sequence. We refer this as the level-sequence matching strategy.

In summary, we have explored an expanded space of computation sequences over blocks of a uniform size, which is to be determined by a given hierarchical memory system. We have presented the reflective ordering for maximal reuse of data and common expressions. We have provided the level-wise operation strategy to meet the arithmetic and numerical constraint. The total storage requirement is minimized by using the reflective ordering in pre-add at each level combined with the level-sequence matching strategy.

4 Experimental Tests

The recursion unfolding schemes for performance improvements presented in §3 should be fully exploited with automatic code generation using compiler techniques, and are as such beyond the scope of this paper. We illustrate in this section the potential of our approaches in the improvement of actual CPU and wall-clock timings in comparison to those of recursive Strassen's algorithm in depth first ordering. We present experimental performance results obtained with three non-recursive variants that incrementally apply the concepts introduced in §3 and progressively improve the overall performance. The general strategy is referred to as delayed additions strategy (DAS) since it consists in delaying and advancing pre- and post-addition of blocks of uniform size, with respect to the recursive version of Strassen's algorithm. The three non-recursive variants are outlined as follows.

- DAS. The first non-recursive implementation takes the level-wise computation sequence (ii) in (10) for pre-adds and sequence (iii) for post-adds. In other words, DAS does *all* computations on blocks of uniform size $n_o \times n_o$ subject to the arithmetic and numeric constraint. The ordering in $\{P_j\}$, $j = 1 : 7^k$ is heuristic but nearly optimal in data reuse and memory reuse.

- OPT-DAS. The second implementation employs the optimal ordering in $\{P_j\}$ given in (6) for maximal data reuse and memory reuse.

- BLOCK OPT-DAS. The third implementation assumes a block recursive data layout scheme on the matrices A, B and C, which are layed out columnwise in the previous two implementations.

We refer as STRASSEN to our implementation of the recursive Strassen's algorithm in depth-first ordering. We permit the GAXPY [8] operation $C = C + A * B$ and therefore can not use the storage for C to store intermediate results at interior levels. In other words, the intermediate results require additional storage. The internal storage is $3 \cdot 4^k$ blocks at minimum for STRASSEN and DAS, and $2 \cdot 3^k + 4^k$ blocks for OPT-DAS and BLOCK-OPT-DAS. For DAS based implementations, we use block data storage for all intermediate results, which are invisible to the user.

4.1 Test-Bed Platforms

All our implementations are tested on three different computer architectures with different memory configurations. Table 1 contains relevant features of the test-bed platforms considered. Notice the relative differences in level-1 and level-2 cache sizes as well as those with super-scalar and dynamic execution capabilities. In contrast to the RS6000, the Alpha architecture is considered fast but unsophisticated since it relays more in the speed of its CPU than on architecture optimizations such as dynamic execution [9]. Notice the relative

	IBM RS6000	DEC AlphaStation 600	INTEL Pentium II
CPU	RS6000	Alpha 21164	Pentium II
L1 Cache	256 KB	8 KB	8 KB
L2 Cache		96 KB	256 KB
Super-scalar	Yes	Yes	Yes
Dynamic exec	Yes	No	Yes

Table 1: Specifications of test-bed platforms.

difference between level-1 cache sizes. The speed of the CPU emphasizes the latency between the computer's cache and memory and hence the enhancement on data locality.

The compiler and library option settings used in our comparisons are given in Table 2.

RS6000	compiler:	AIX C Set ++
	options:	-O2
	libraries:	blas, xlf, m
Alpha	compiler:	DEC C V5.2-023
	options:	-fast, -migrate, -non_shared
	libraries:	dxml, for, Futil, m
Pentium II	compiler:	SUN C PC3.0.1
	options:	-pentium -O2
	libraries:	blas, sunmath, F77, M77, m

Table 2: Compilation details for all platforms.

4.2 Experimental Results

The experimentation is set up in two main parts. Firstly we show the impact of uniform block size and computation sequencing at the cache level by comparing the implementations of Strassen's algorithm to the BLAS implementation of the conventional algorithm on small matrices. Secondly, we observe their effect on larger matrices at the page level. We use only two levels of recursion or recursion unfolding with Strassen's algorithm and then switch to DGEMM of BLAS [5]. Note that the ratio in arithmetic cost between 2-level Strassen's algorithm and the conventional algorithm is a constant about $(7/8)^2$ when the block size at the truncation level is big enough. Thus, the performance is altered mainly by data locality. For the cache level performance, we illustrate the crossover points where the implementations of 2-level Strassen's algorithm start to outperform DGEMM of BLAS in CPU time. For the page level performance, we present comparisons in wall-clock time among the different implementations of Strassen's algorithm.

4.2.1 Experiments at the Cache Level

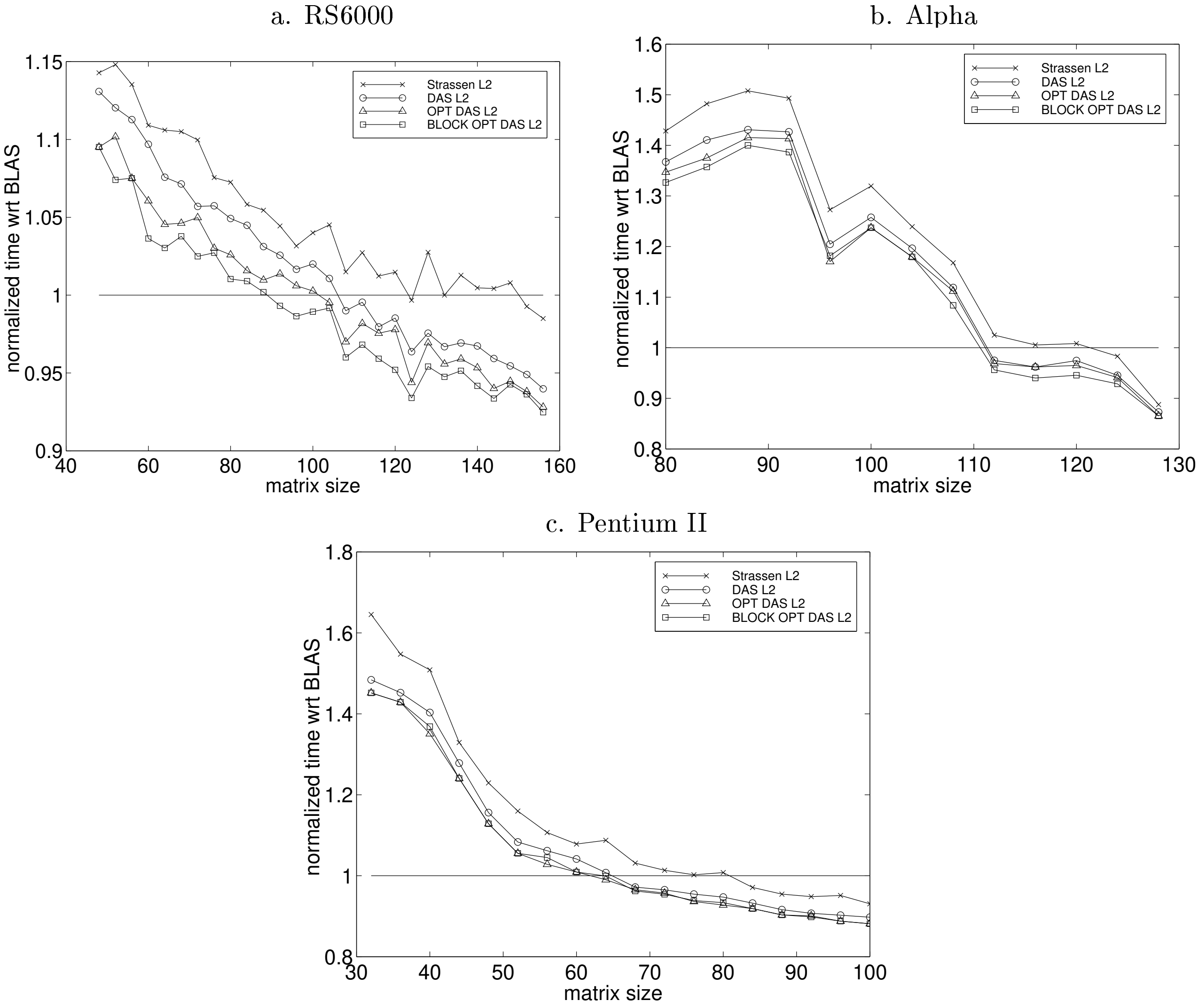

Figure 2: Normalized CPU times with respect to compiled DGEMM on the RS6000, Alpha, and Pentium II architectures. The improvement of the non-recursive variants over STRASSEN is seen in the migration of the truncation points towards zero.

Figure 2 presents the comparisons in normalized CPU time, i.e. the timing results are normalized with respect to that for a compiled DGEMM subroutine. The points below the horizontal line at 1 mean improvement over DGEMM. Notice that the crossover points on the horizontal line for DAS, OPT-DAS and BLOCK-OPT-DAS move progressively away from the crossover point of STRASSEN and toward smaller matrix size.

Table 3 summarizes the results on the crossover points obtained for all four implementations.

	RS6000	Alpha600	Pentium II
STRASSEN	152	124	76
DAS	108	112	64
OPT-DAS	104	112	60
BLOCK-OPT-DAS	92	112	60

Table 3: Crossover points with respect to a compiled DGEMM subroutine.

While the improvement of the non-recursive implementation over the recursive implementation is evident across all three architectures, we shall note that the progress in migrating the crossover point in the RS6000 is close to the best in the implementation setup as we described. The level-1 cache can hold an entire matrix (block) of size up to 90×90. The reduction in arithmetic complexity (and hence the number of memory references) and the improvement in data locality will not be significant until the matrix size is larger than 90. Our best crossover point is 92. The multi-level and level-variant caching policies of the cache system in the Alpha architecture entail more complicated data locality enhancement strategies. Likewise, using manufacture provided implementations of DGEMM would complicate the comparisons on the impact of improvement in data locality. Nevertheless, we observed a significant improvement in the cross-over point of our DAS implementations with respect to *native* BLAS DGEMM in the RS6000 architecture.

4.2.2 Experiments at Page Level

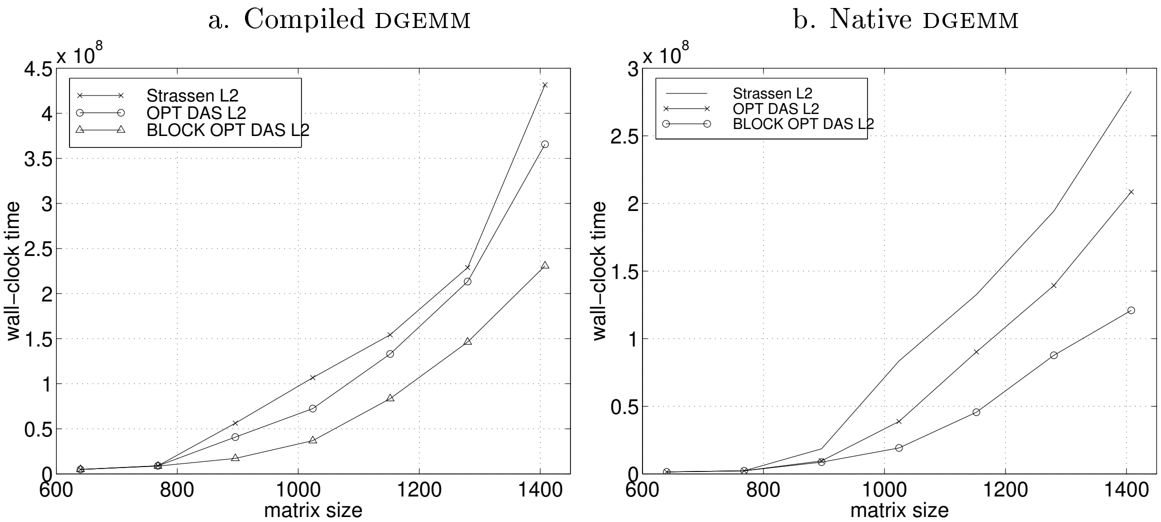

Figure 3: Wall-clock time comparisons between STRASSEN, OPT-DAS, and BLOCK-OPT-DAS. Compiled and native DGEMM were used on the left and right plots respectively. Notice that paging starts to occur for matrices of size greater than 768×768.

At the page level, the gap between computation time and I/O time emphasizes the improvement in data locality. Even in the presence of an effective pre-fetching system, less paging activity is desirable. The overall performance in time at this level is indicated by the wall-clock time, consisting of CPU time and I/O time. For illustration, we use an Alpha architecture configured to 16MB of user data memory. This configuration allows paging to occur for matrices of size greater than 744×744 approximately. Figure 3 shows the comparisons in wall-clock timing among STRASSEN, OPT-DAS, and BLOCK-OPT-DAS. Experiments in Figure 3a and Figure 3b use the compiled DGEMM subroutine and the native DGEMM subroutine, respectively, at the truncation level. Our experiments are designed so that there is no paging at the truncation level. Thus, the difference between the two versions is in the gap between computation time and I/O time, the faster implementation of DGEMM enlarges the gap. We see from Figure 3 that when paging start to occur, STRASSEN consistently takes more time than the other implementations. The difference between OPT-DAS and BLOCK- OPT-DAS is in the data layout. Evidently, the column-major data layout of A, B and C in OPT-DAS undermines the data locality at the algorithm level. We expect the same impact of data layout of intermediate results in implementations with further recursion unfolding.

5 Remarks

Our analysis on data reuse and space reuse at the algorithm level has resulted in effective optimization strategies for performance enhancement and can lead to automatic optimization of recursion unfolding for more than two levels. As we have explained and illustrated, the data layout issue is important in realizing the maximal reuse of data (common expressions) and the maximal reuse of memory. While the potential of improving temporal data locality can be exposed independent of data layout at the algorithm level, both the temporal data locality and the spatial data locality at the programming level depend on the consistency between the computation sequence and the data layout. We recognize that while the data layout of A, B, and C at the interface level may be given, the data layout for the intermediate results is free of interface constraint and, more improtantly, has dominant impact on the realization of optimal data locality and memory reuse. Specifically, for k-level recursion unfolding, the ratio between the number of the internal blocks (for intermediate results) at minimum and the number of external blocks (i.e., the blocks of A, B and C together) is bigger than $(7/4)^{k+1}$. Our study in characterizing and optimizing the consistency between computation sequence and data layout is in progress along with the automatic code generation and optimization.

6 Acknowledgment

We thank Nikos Pitsianis and Bob Plemmons for their valuable suggestions and comments. Also we thank Jeff Chase for his support with the reconfiguration of a DEC Alpha memory system for our page-level experimental setup. This work is supported in part by NSF Grants CDA-97-2637 and CDA-95-12356, Duke University, and an equipment donation through Intel Corporations Technology for Education 2000 Program. Pauca and Sun were in part supported by the Advanced Research Projects Agency, under contract P-95006. Chatterjee was partially supported by NSF CAREER Award CCR-95-01979. Lebeck is partially supported by NSF CAREER Award MIP-97-02547. Through the use of the Maui High Performance Computing Center, this research was sponsored in part by the Phillips Laboratory, Air Force Materiel Command, USAF, under cooperative agreement number F29601-93-2-0001.

References

[1] D. H. Bailey. Extra high speed matrix multiplication on the Cray-2. *SIAM J. Sci. Stat. Comp.*, 9(3):603–607, 1988.

[2] D. Bini and G. Lotti. Stability of fast algorithms for matrix multiplication. *Numer. Math.*, 36:63–72, 1980.

[3] J. A. Board, Jr., C. W. Humphres, C. G. Lambert, W. T. Rankin, and A. Y. Toukmaji. Ewald and multipole methods for periodic n-body problems. In *Proceedings, Eighth SIAM Conference on Parallel Processing for Scientific Computing*, page CDROM. SIAM Press, 1997.

[4] IBM Corporation. *IBM engineering and scientific subroutine library guide and reference*, 1992. Order number: SC23-0526.

[5] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.

[6] C. Douglas, M. Herosx, G. Slishman, and R. M. Smith. GEMMW: a portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multipy algorithm. *J. Comp. Phy.*, 110(1-10), 1994.

[7] P.C. Fischer and R. L. Probert. Efficient procedures for using matrix algorithms. In *Automata, Languages and Programming*, pages 413–427, Berlin, 1974. Springer-Verlag.

[8] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1996.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, 1996.

[10] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, Philadephia, 1996.

[11] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of Supercomputing '96*, 1996.

[12] J. R. Johnson, R. W. Johnson, D. Rodrigues, and R. Tolimieri. A methodology for designing, modifying and implementing Fourier transform algorithms on various architectures. *Circuits Systems and Signal processing*, 9(4):450–500, 1990.

[13] B. Kumar, C.-H. Huang, , P. Sadayappan, and R. W. Johnson. A tensor product formulation of Strassen's matrix multiplication algorithm. preprint, 1996.

[14] A. R. Lebeck and D. A. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE COMPUTER*, pages 15–26, 1994.

[15] C. Van Loan. *Computational Frameworks for the Foast Fourier Transforms.* SIAM, 1992.

[16] N. P. Pitsianis. *The Kronecker Product in Approximation and Fast Transform Generation.* PhD thesis, Cornell University, 1996.

[17] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[18] X. Sun, V. P. Pauca, and N. Pitsianis. Formal error analysis of bilinear noncommutative matrix multiplications. preprint, 1998.