Studienarbeit

External Matrix Operations for the STXXL

Raoul Steffen
R [hyphen] Steffen [at] gmx [dot] de

Advisers:
Dr. Johannes Singler
Prof. Dr. Peter Sanders

April 30, 2011

Abstract

This thesis discusses the matrix container template I implemented as part of the STXXL library for very large data sets.[1] Because it is designed for matrices too big to be held in internal memory, algorithms and data structures are chosen to be efficient for external memory operation.[2]

Transposition, addition, and scalar multiplication are easy; therefore their description is kept brief. Matrix multiplication is algorithmically interesting; several approaches and algorithms exist for matrix multiplication in general as well as for external matrix multiplication in effort to make it more efficient.

For that reason, extensive discussion including theoretical analysis and practical tests focus on matrix multiplication.

1 Motivation

1.1 Memory Hierarchy

Computer memory is organized in several layers, from large and slow to fast and small memory, called the *memory hierarchy.* The largest layer is used as main storage; memory indexing (addressing) refers to this layer. Faster layers are used as *cache*; they hold copies of data to provide faster access.

To reduce overheads, slower layers are not accessed for single bytes or words but for increasingly larger portions of consecutive values at once. These portions are called *cache lines.*

Compared to using only large memory, accessing a single element becomes even slower, because it is copied into cache together with the whole cache line it is part of. The effort amortizes if a significant fraction of the cache line is accessed before it is removed from cache. This effect is called *spatial locality.*

The use of fast memory becomes more useful when data copied to it is accessed more often before it is removed from cache. This effect is called *temporal locality.*

In case of internal memory algorithms, DRAM is used as largest layer. The cache layers are neither addressed nor managed explicitly by the software. Instead, copying to and from them is managed transparently by hardware mechanisms. That way, the behavior of large and fast memory is approximated.

In case of external memory algorithms, the largest layer to store most of the data consists of disks. In that case, the largest DRAM layer functions as first cache layer with the feature and burden of explicit addressing and management. The disk blocks become the cache lines of that layer. Typical sizes are around 16MiB (i. e. 2^{19} words at 64 bit per word).

1.2 I/O Model

The I/O-complexity of the algorithms is analyzed using the Multihead Model by Aggarwal and Vitter. [3] The model counts the number of reads and writes of data blocks of B bytes size (cache lines or disk blocks) to secondary storage, and the necessary units of time for it. It allows for D concurrent I/Os in one unit of time; typically, D is the number of disks.

[1]For more information about the STXXL project, see: http://stxxl.sourceforge.net/

[2]External memory algorithms are often also called out-of-core algorithms.

Choosing smaller values for B allows to analyze accesses from other cache layers to DRAM or to the next larger cache layer, too. An algorithm is called *cache oblivious*, if its I/O-complexity does not depend on B, that is, if $B \cdot \text{I/Os}(B) \in O(1)$.

1.3 Characteristic of Matrix Handling

Computer memory is organized using one-dimensional indexing (addressing) of storage units. To store a two-dimensional matrix, the two dimensions of the matrix have to be laid out on the one dimension of memory addresses. The layout pattern defines a layout function $L : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$. Historically and for human readability, the mostly used layout functions are row-major and column-major.

$$\begin{aligned} L_{\textit{row-major}}(i, j) &= i \cdot \text{width} &&+ \quad j \\ L_{\textit{column-major}}(i, j) &= i &&+ \quad j \cdot \text{height} \end{aligned}$$

These work great for matrices small enough to fit in L1 cache, but as matrices grow larger a problem arises. Consider a $n \times n$ matrix M stored in row-major and a cache with a cache line size of s elements not large enough to hold (most of) the matrix. Traversal of one column j of M (i. e. accessing the elements at $(1, j), (2, j), \ldots (n, j)$) requires loading either M completely into the cache, or a whole cache line (s elements) per accessed element. The impact of this problem increases with s. Standard matrix multiplication requires traversal of one row and one column of the input matrices per element of the output matrix. Transposition can be seen as reading a matrix stored in row-major by columns.

In C++ the standard container to store a large number of elements is the template std::vector<>. With stxxl::vector<> the STXXL provides a container for even larger numbers of elements. stxxl::vector<> stores the elements in external memory, caching disk blocks of several MB size. That means s is be in the magnitude of 10^6–10^7 for 64 Bit values.

This work studies alternative layouts and matrix multiplication algorithms to reduce the mentioned problem and implements a layout function and some matrix multiplication algorithms as new parts of the STXXL.

2 Matrix Layout

This section describes different layout functions and shortly discusses their suitability for different operations.

2.1 Row-Major and Column-Major

I consider row-major only, since column-major can be seen as its transposed duality.

The row-major layout arranges one row after another, beginning with the top row. It is a very easy layout and well suited for accessing single lines. Accessing single columns however, can require one disk access per element or loading all of the matrix (whichever is better). Because of this, row-major is very unsuited for complex matrix operations e.g. multiplication.

The situation is different for matrices that fit in L1 cache. Single words are loaded from L1, so there is no penalty for loading unneeded elements. On the other hand, the simplicity of access patterns for row- and column-traversal enables prediction mechanisms in hardware and compilers to employ efficient prefetching as well as loop unrolling and interleaving. [4]

$$\begin{array}{lcccc} L^{n\times m}_{\textit{row-major}}(i,j) & = & i\cdot m & + & j \\ L^{n\times m}_{\textit{column-major}}(i,j) & = & i & + & j\cdot n \end{array}$$

2.2 Nested Layouts

Layouts can be nested to create further layouts.

Let L_{outer} and L_{inner} be layout functions, $n \times m$ a matrix size and $n' \times m'$ a submatrix size. Assuming $n' \mid n$ and $m' \mid m$, divide the matrix into submatrices. The resulting $N \times M$ matrix (with $N = n/n'$, $M = m/m'$) whose elements are submatrices is called supermatrix. Lay out the supermatrix using L_{outer} and the elements in the submatrices using L_{inner}.

If $n' \nmid n$ or $m' \nmid m$, additional rows respectively columns containing zeros are added to the matrix. This is called *static padding* and induces some overhead in space and computation, but the overhead grows only linear with n, m, n', and m' and is negligible for matrices with $n, m >> n', m'$.

$$L^{n\times m}_{nested}(i,j) = \overbrace{L^{N\times M}_{outer}(\lfloor i/n' \rfloor, \lfloor j/m' \rfloor)}^{\text{number of submatrix}} \cdot n'm' + \overbrace{L^{n'\times m'}_{inner}(i \bmod n', j \bmod m')}^{\text{number in submatrix}}$$

where

$$N = \lceil n/n' \rceil$$
$$M = \lceil m/m' \rceil$$

2.3 Submatrix Layout

Submatrix layout [8, 10] (also called block layout or block storage, but not to be confused with disk blocks), is a nested layout with square submatrices (i. e. $n' = m'$). With a disk block size of B elements, n' is chosen so that $B \mid n'^2$, i. e. disk blocks do not cross submatrix boundaries. Inner and outer layout are row-major.

Submatrix layout allows to load a whole submatrix into internal memory for internal processing without loading further elements. This is still not very good for column traversal, but helps for transposition and multiplication.

$$L_{submatrix}(i,j) = \overbrace{(\lfloor i/n' \rfloor \cdot M + \lfloor j/n' \rfloor)}^{\text{number of submatrix}} \cdot n'^2 + \overbrace{(i \bmod n') \cdot n' + j \bmod n'}^{\text{number in submatrix}}$$

2.4 Morton Order Layout

Morton Order layouts are recursive layouts. I focus on Z-Morton Order as representative of the different Morton layouts, as it already introduces all of the concept. Chatterjee et al. [4] present other variants and further details.

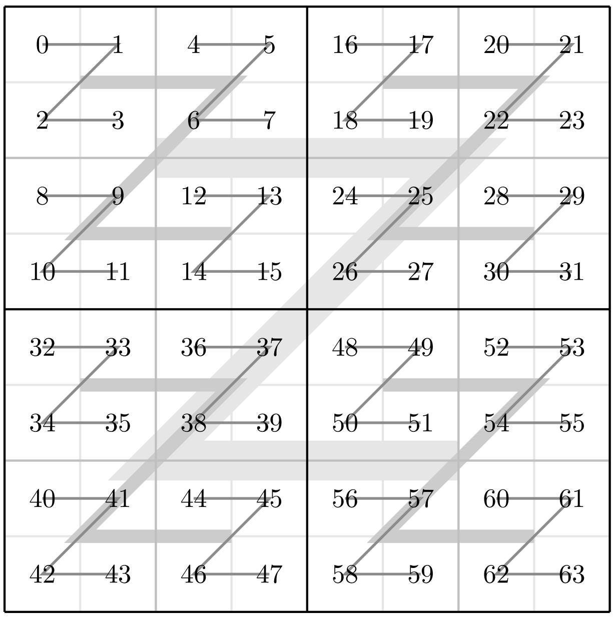

Figure 1: Z-Morton Order layout for a 8×8 matrix

$L_{morton}^{2^k \times 2^k}$ is obtained by recursively nesting

$$L_{inner} = L_{morton}^{2^{k-1} \times 2^{k-1}}$$

and $L_{outer} = L_{row\text{-}major}^{2\times 2} = [(0,0),(0,1),(1,0),(1,1)]$

requiring $n' = m' = n/2 = m/2 = 2^k \in 2^{\mathbb{N}}$

Figure 1 illustrates the resulting Layout. Other variants of Morton Order layouts differ in L_{outer}.

Morton order keeps the elements of submatrices close together on all levels. It prefers neither rows nor columns, allowing for relatively efficient traversal of both. It also naturally leads to cache-oblivious algorithms: By following the recursive layout, on each level the submatrix small enough to fit will reside in cache.

The one major drawback is the restriction to square matrices whose order is a power of two. This can require a lot of static padding (adding rows and columns of zeros respectively); up to almost three times the original size for square matrices and arbitrarily much for irregular ones. According to Chatterjee et al. [4] the disadvantages of padding can be reduced drastically by flagging blocks of zeros, thus saving read, write and arithmetic overhead. I found that this does not always help that much (see section 4.5.3).

$$L_{Z\text{-}Morton}(i,j) = \sum_{\alpha=0}^{\infty} \left(\left\lfloor \frac{i}{2^\alpha} \right\rfloor \mod 2 \right) \cdot 2^{2\alpha+1} + \left(\left\lfloor \frac{j}{2^\alpha} \right\rfloor \mod 2 \right) \cdot 2^{2\alpha}$$

2.5 Further Nested Layouts

Other nested layouts can be beneficial.

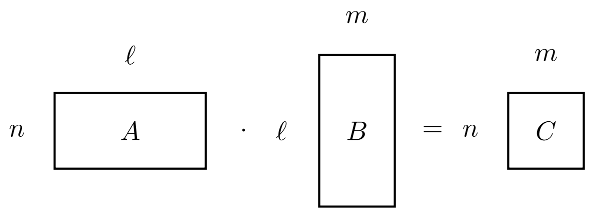

Figure 2: Names of dimensions involved in matrix multiplication.

For example, one can save small submatrices –called tiles– in row-major layout and arrange those in Morton Order among each other. That way one can exploit hardware prefetching and loop unrolling by the compiler as well as the locality of Morton order on higher levels.

Another example would be to use submatrices in Morton Order (for locality) and arrange those in row major, overcoming the power-of-two and regularity restrictions.

For any nesting of layouts, as long as all inner sizes are powers of two (they do not have to be square), the addresses can be kept in one unsigned integer. Addresses can then be added/subtracted and transformed to/from coordinates using bit arithmetic in few instructions. [2, 7]

3 Matrix Multiplication Algorithms

In this section I introduce some algorithms for matrix multiplication and discuss their efficiency regarding I/O. I start with the inefficient naive algorithm, continue with simple, yet quite efficient blocked ones and finish with an advanced, Strassen-like algorithm.

Research and use of I/O-efficient algorithms for matrix operations has been there since the 1950s. One of the earliest publications is by Rubinstein and Rutledge [8], in which the authors present implementations for the UNIVAC enabling it to handle matrices of order up to 310 that could only fit in external memory at that time.

3.1 Naive Standard Matrix Multiplication

For matrices A, B, C of size $n \times \ell$, $\ell \times m$, $n \times m$ respectively (cf. figure 2), $C = A \cdot B$ is defined as:

$$c_{i,j} = \sum_{k=1}^{\ell} a_{i,k} \cdot b_{k,j} \qquad \text{for } i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}$$

This leads to a simple algorithm: For each element in C, calculate the sum accordingly, iterating over i, j and k. By iterating over k in the innermost loop, only one element of intermediate result has to be stored from one iteration to the other. Thereby it can be kept in fast memory (usually registers or fast cache) and the least number of read accesses to slower memory is necessary.

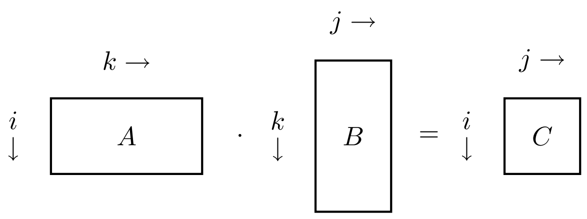

Figure 3: Coordinate (iteration) variables as used in naive standard matrix multiplication and standard matrix multiplication with reordered loop nesting. Naive standard matrix multiplication iterates k in the innermost loop, whereas standard matrix multiplication with reordered loop nesting iterates j in the innermost loop.

3.2 Standard Matrix Multiplication with Reordered Loop Nesting

Naive standard matrix multiplication involves three counters along the three dimensions of the multiplication (cf. figure 3). To obtain all combinations, they are iterated in nested loops. For matrices stored in row-major order, the costly traversal of columns can be saved simply by using a different order of nesting. [11]

Naive standard matrix multiplication iterates k in the innermost loop, thus accessing a column of A and a row of B. As discussed before, accessing columns is costly.

By iterating j in the innermost loop, a row of B and a row of C are accessed. The save of a column access comes at the cost of two row accesses, because now the elements of C have to be reread in order to sum up on them. This is worth it as soon as at least one level of cache is crossed.

3.3 Recursive Standard Matrix Multiplication

To calculate $C = A \cdot B$, divide $A = \left(\begin{smallmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{smallmatrix}\right)$ and $B = \left(\begin{smallmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{smallmatrix}\right)$ into submatrices[3]. Then $C = \left(\begin{smallmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{smallmatrix}\right)$ with

$$C_{i,j} = A_{i,1} \cdot B_{1,j} + A_{i,2} \cdot B_{2,j} \ .$$

Calculate the products recursively. The sums can be calculated together with the second product instead of afterwards, thus saving write and read of this summand.

This algorithm works well with recursive matrix layouts such as Morton Order (see section 2.4), establishing good locality of reference and being cache-oblivious. It has the disadvantage of temporarily storing intermediate results and thereby filling the caches and causing additional accesses and I/Os.

3.4 Blocked Matrix Multiplication

The upcoming algorithm takes advantage of the submatrix layout. The order of submatrices n' is chosen as large as possible, such that $3n'^2 \leq M$[4]. That

[3]The submatrices' sizes do not have to be equal, they need only suffice the products.

[4]To overlap I/O and computation, $6n'^2 \leq M$ would be necessary

Preadditions	Recursive Multiplications	Postadditions
	$P_1 = A_{11} \cdot B_{11}$	
	$P_2 = A_{12} \cdot B_{21}$	
	$P_3 = A_{21} \cdot B_{11}$	$C_{11} = P_1 + P_2$
	$P_4 = A_{22} \cdot B_{21}$	$C_{21} = P_3 + P_4$
	$P_5 = A_{11} \cdot B_{12}$	$C_{12} = P_5 + P_6$
	$P_6 = A_{12} \cdot B_{22}$	$C_{22} = P_7 + P_8$
	$P_7 = A_{21} \cdot B_{12}$	
	$P_8 = A_{22} \cdot B_{22}$	

Preadditions	Recursive Multiplications	Postadditions	
$S_1 = A_{21} + A_{22}$			
$S_2 = S_1 - A_{11}$	$P_1 = A_{11} \cdot B_{11}$	$C_{11} =$	$U_1 = P_1 + P_2$
$S_3 = A_{11} - A_{21}$	$P_2 = A_{12} \cdot B_{21}$		$U_2 = P_1 + P_4$
$S_4 = A_{12} - S_2$	$P_3 = S_1 \cdot T_1$		$U_3 = U_2 + P_5$
	$P_4 = S_2 \cdot T_2$	$C_{21} =$	$U_4 = U_3 + P_7$
$T_1 = B_{12} - B_{11}$	$P_5 = S_3 \cdot T_3$	$C_{22} =$	$U_5 = U_3 + P_3$
$T_2 = B_{22} - T_1$	$P_6 = S_4 \cdot B_{22}$		$U_6 = U_2 + P_3$
$T_3 = B_{12} - B_{12}$	$P_7 = A_{22} \cdot T_4$	$C_{12} =$	$U_7 = U_6 + P_6$
$T_4 = B_{21} - T_2$			

Figure 4: Recursive standard matrix multiplication (above) and Strassen-Winograd matrix multiplication (below). [4]

way, three submatrices can be held in internal memory, just enough to perform internal matrix multiplication on them. This has the drawback that the amount of required internal memory is fixed with the layout; it can not be changed for existing matrices without reordering them.

The supermatrix is multiplied with the naive algorithm, with internal memory submatrix multiplication again by the naive algorithm. This essentially results in a rescheduling of the naive standard algorithm that requires much less I/Os (see sections 3.6.1 and 3.6.4).

A more detailed view of this algorithm has been given by McKellar. [6]

3.5 Strassen-Winograd Matrix Multiplication

The main idea of the Strassen-Winograd matrix multiplication algorithm is to replace one of the eight recursive multiplications of recursive standard matrix multiplication by additions using algebraic identities, thereby decreasing its operation count from $\Theta(n^3)$ to $\Theta(n^{\operatorname{ld} 7}) \approx \Theta(n^{2.8})$. [9] Winograd improved the algorithm by identifying common subexpressions; his variant needs only 15 instead of 18 additions. [4]

As in recursive standard matrix multiplication, each matrix is divided into four submatrices, but now the submatrices have to be of equal size. For this constraint to hold on every level of recursion, $m, n, \ell \in 2^{\mathbb{N}}$ is required. $m, n, \ell \in t \cdot 2^{s+\mathbb{N}}$ for some constants $s, t \in \mathbb{N}$ suffices for s levels of Strassen-Winograd. In fact –as we will see in section 3.6.5– a not to small base case is advantageous.

In the preaddition phase, eight additional matrices are calculated. Then the recursion phase forms seven products, from which the four submatrices of the result can be computed in the postaddition phase, having seven additions. See figure 4 for details.

Strassen-Winograd works well with recursive layouts such as Morton Order. Other than naive standard matrix multiplication, it seems to gain only little advantage of the locality of those layouts compared to row-major layout. [4] This effect can be explained with the preadditions and postadditions. Elements of the original matrix are only accessed to perform additions, which are much less sensible to the layout than multiplications. However, the intermediate results have the same size as the submatrices (half the height and width). When the base case is reached and multiplications are performed eventually, their input matrices have become so small that locality within one matrix hardly matters. [4]

Through this mechanism, preadditions and postadditions render Strassen-Winograd cache-oblivious.

3.6 I/O and Operation Count

For each presented algorithm, the I/O and operation count depending on n, m and ℓ can be calculated algebraically, allowing to determine the theoretically best for each matrix size. Of course, several other parameters can influence the actual resource usage, so the theoretical values should be used as reference points only.

For the course of this section assume $n = m = \ell$ (i. e. square matrices) and $n^2 > M$ and let
M be the size of internal memory,
B be the size of one disk block.
For an algorithm A, let
$i_A(n)$ be the number of block I/Os,
$a_A(n)$ be the number of element additions,
$m_A(n)$ be the number of element multiplications
performed by A to multiply two square matrices of order n.

Lemma:

$$f(n) = an^2 + bf\left(\frac{n}{2}\right) \wedge f(d) = c \wedge b \neq 4 \wedge \frac{n}{d} \in 2^{\mathbb{N}}$$

$$\implies$$

$$\begin{aligned} f(n) &= an^2 \sum_{i=0}^{\operatorname{ld}\frac{n}{d}-1} \left(\frac{b}{4}\right)^i + b^{\operatorname{ld}\frac{n}{d}} c \\ &= \frac{\frac{4ad^2}{b-4} + c}{d^{\operatorname{ld} b}} n^{\operatorname{ld} b} - \frac{4a}{b-4} n^2 \end{aligned}$$

Proof by induction and geometric sums.

In I/O count formulas, a will be the number of I/Os caused by preadditions and postadditions per level of recursion and b the number of recursive calls. d is the order of the base case, c its cost.

3.6.1 Naive Standard Matrix Multiplication

Consider row-major layout and LRU replacement for this algorithm. Assume $M \geq n + 3B$, i. e. at least one row of A can be held in internal memory.

For each row of C the algorithm holds the corresponding row of A in internal memory and traverses all columns of B. For each element of B either one block or, if one block spans more than one row, the whole row containing that element has to be loaded.

$$i_{naive}(n) = \overbrace{n}^{\text{for each row of } C}(\underbrace{2\frac{n}{B}}_{\text{read A, write C}} + \overbrace{n^2 \min\left\{1, \frac{n}{B}\right\}}^{\text{load B column wise}}) = \frac{1}{B}\left(\min\{B, n\} \cdot n^3 + 2n^2\right)$$

Even with other layouts and replacement strategies, this algorithm is I/O inefficient. Assume $M \leq n^2/2$, i. e. at most half of one matrix fits in internal memory. For each row of C, all elements of B are needed, thus at least $n^2/2$ elements have to be loaded n times. This results in at least $\frac{n^3}{2B}$ I/Os.

$$a_{naive}(n) = n^2(n-1)$$
$$m_{naive}(n) = n^3$$

3.6.2 Standard Matrix Multiplication with Reordered Loop Nesting

Consider row-major layout and LRU replacement for this algorithm. Assume $M \geq n + 3B$, i. e. at least one row of C can be held in internal memory.

For each row of C one row of A is multiplied with all rows of B (one element of A per row of B).

$$i_{reordered}(n) = \overbrace{n}^{\text{for each row of } C}(\underbrace{2\frac{n}{B}}_{\text{read } A, \text{ write } C} + \overbrace{\frac{n^2}{B}}^{\text{read } B}) = \frac{1}{B}\left(n^3 + 2n^2\right)$$

Compared to naive standard matrix multiplication, the factor of $\min\{B, n\}$ for column traversal disappears. Since the current row of C is assumed to fit in internal memory, there is no penalty for rereading intermediate results.

3.6.3 Recursive Standard Matrix Multiplication

Consider Morton Order or submatrix layout with small submatrices for this algorithm. With these layouts, one submatrix can be loaded without loading further elements.

Let $t \in \mathbb{N}$ maximal so that $3(tn')^2 \leq M$, i. e. three $tn' \times tn'$ submatrices fit in internal memory. To gain a useful recurrence assume $r = \frac{n}{tn'} \in 2^{\mathbb{N}}$.

$$
\begin{aligned}
i_{recursive\ standard}(n) &= 8 \cdot i_{recursive\ standard}\left(\frac{n}{2}\right) + n^2/B \\
i_{recursive\ standard}(tn') &= 3(tn')^2/B \\
&\Longrightarrow \\
i_{recursive\ standard}(n) &= \frac{\frac{4(tn')^2}{B(8-4)} + \frac{3(tn')^2}{B}}{(tn')^3} n^3 - \frac{4}{B(8-4)} n^2 \\
&= \frac{4(tn')^2}{B(tn')^3} n^3 - \frac{n^2}{B} \\
&= \frac{4}{B} r n^2 - \frac{n^2}{B} \\
&= (4r-1)\frac{n^2}{B} \\
&\approx \frac{1}{B}\left(4\sqrt{\frac{3}{M}} \cdot n^3 - n^2\right)
\end{aligned}
$$

$$
\begin{aligned}
a_{recursive\ standard}(n) &= n^2(n-1) \\
m_{recursive\ standard}(n) &= n^3
\end{aligned}
$$

3.6.4 Blocked Matrix Multiplication

Consider submatrix layout for this algorithm. The size of submatrices is chosen to fill internal memory as described in section 3.4 ($3n'^2 \leq M$).

For each submatrix of C to be written, one row of submatrices of A and one column of submatrices of B are needed.

$$
\begin{aligned}
i_{blocked}(n) &= \left(\frac{n}{n'}\right)^2 \cdot \left(2\frac{n}{n'} + 1\right) \cdot \frac{n'^2}{B} \\
&= \frac{1}{B} \cdot n^2 \left(2\frac{n}{n'} + 1\right) \\
&\approx \frac{1}{B} \cdot \left(2\sqrt{\frac{3}{M}} \cdot n^3 + n^2\right)
\end{aligned}
$$

Explicit caching and cache size fixed with the layout allow for almost a constant factor of two less I/Os compared to recursive standard matrix multiplication.

$$
\begin{aligned}
a_{blocked}(n) &= n^2(n-1) \\
m_{blocked}(n) &= n^3
\end{aligned}
$$

The I/O counts of blocked matrix multiplication and recursive standard matrix multiplication are asymptotically optimal for algorithms that compute all n^3 products $a_{i,k} \cdot b_{k,j}$, as has been proven by Hong and Kung. [5] A simplified version of the proof has been given by Toledo. [10]

3.6.5 Strassen-Winograd Matrix Multiplication

As in section 3.6.3, consider Morton Order or submatrix layout with small submatrices for this algorithm. With these layouts, one submatrix can be loaded without loading further elements.

Let $t \in \mathbb{N}$ maximal so that $\alpha(tn')^2 \leq M$, i. e. α $tn' \times tn'$ submatrices fit in internal memory. $\alpha \approx 5$ is necessary to store intermediate results during internal matrix multiplication, depending on the implementation. To gain a useful recurrence assume $r = \frac{n}{tn'} \in 2^{\mathbb{N}}$.

Strassen-Winograd has to read all eight input submatrices and write eight intermediate submatrices during preaddition phase. Postaddition phase requires to read the seven products and write four result submatrices.

$$\begin{aligned}
i_{strassen\ winograd}(n) &= 7 \cdot i_{strassen\ winograd}\left(\frac{n}{2}\right) + \underbrace{27}_{=8+8+7+4}\left(\frac{n}{2}\right)^2 /B \\
i_{strassen\ winograd}(tn') &= 3(tn')^2/B \\
&\Longrightarrow \\
i_{strassen\ winograd}(n) &= \frac{\frac{27(tn')^2}{B(7-4)} + \frac{3(tn')^2}{B}}{(tn')^{\operatorname{ld} 7}} n^{\operatorname{ld} 7} - \frac{27}{B(7-4)} n^2 \\
&= \frac{(3+\frac{27}{3})(tn')^2}{B(tn')^{\operatorname{ld} 7}} n^{\operatorname{ld} 7} - \frac{27}{3B} n^2 \\
&= \frac{12(tn')^2}{B(tn')^{\operatorname{ld} 7}} n^{\operatorname{ld} 7} - \frac{9}{B} n^2 \\
&= \frac{12}{B(tn')^{\operatorname{ld} 7 - 2}} n^{\operatorname{ld} 7} - \frac{9}{B} n^2 \\
&\approx \frac{12}{B(\frac{M}{\alpha})^{(\operatorname{ld} 7 - 2)/2}} n^{\operatorname{ld} 7} - \frac{9}{B} n^2 \\
&\overset{\alpha\approx 5}{\approx} \frac{1}{B}\left(\frac{23}{M^{0.4}} n^{\operatorname{ld} 7} - 9n^2\right)
\end{aligned}$$

Constant factors are higher than for the simple algorithms, as is typical for algorithms with preprocessing or postprocessing. Interestingly, the size of internal memory occurs not as a factor of $M^{-0.5}$ but only about $M^{-0.4}$. The reason are the intermediate results: To process a $2n^* \times 2n^*$ matrix rather than a $n^* \times n^*$ matrix internally, not only four times the data has to fit in internal memory, but also the intermediate results of one level of recursion have to be stored.

Concerning multiplication and addition counts, at first I want to show the case that multiplication is broken down to single elements. One level of recursion requires 15 preadditions and postadditions and seven recursive multiplications.

$$\begin{aligned}
a_{strassen\ winograd}(n) &= 7 \cdot a_{strassen\ winograd}\left(\frac{n}{2}\right) + 15\left(\frac{n}{2}\right)^2 \\
a_{strassen\ winograd}(\mathbf{1}) &= 0 \\
&\Longrightarrow \\
a_{strassen\ winograd}(n) &= \frac{15}{7-4}n^{\operatorname{ld} 7} - \frac{15}{7-4}n^2 \\
&= 5(n^{\operatorname{ld} 7} - n^2)
\end{aligned}$$

$$\begin{aligned}
m_{strassen\ winograd}(n) &= 7 \cdot m_{strassen\ winograd}\left(\frac{n}{2}\right) \\
m_{strassen\ winograd}(\mathbf{1}) &= 1 \\
&\Longrightarrow \\
m_{strassen\ winograd}(n) &= n^{\operatorname{ld} 7}
\end{aligned}$$

Let $o_A(n) = a_A(n) + m_A(n)$. Although $o_{strassen\ winograd}(n) \in \Theta(n^{\operatorname{ld} 7}) \approx \Theta(n^{2.8})$, obviously $o_{naive}(n)$ is much smaller for small n. This raises the question, how large n has to be for Strassen-Winograd to need less operations than standard matrix multiplication.

Consider given n. Strassen-Winograd can only be more efficient, if using it for one level of recursion (and then calling standard matrix multiplication) is more efficient than not using it at all. Thus is a necessary condition:

$$\begin{aligned}
&\frac{15}{4}n^2 + 7o_{naive}\left(\frac{n}{2}\right) < o_{naive}(n) \\
\Longleftrightarrow\ &\frac{15}{4}n^2 + 7\left(2\frac{n^3}{8} - \frac{n^2}{4}\right) < 2n^3 - n^2 \\
\Longleftrightarrow\ &15n^2 + 7(n^3 - n^2) < 8n^3 - 4n^2 \\
\Longleftrightarrow\ &12n^2 < n^3 \\
\Longleftrightarrow\ &12 < n
\end{aligned}$$

The condition becomes sufficient if for small n the recursion is broken and standard matrix multiplication is used instead. This results in the following operation counts for the refined algorithm with base case $n = 12$:

$$\begin{aligned}
a_{strassen\ winograd^*}(n) &= 7 \cdot a_{strassen\ winograd^*}\left(\frac{n}{2}\right) + 15\left(\frac{n}{2}\right)^2 \\
a_{strassen\ winograd^*}(\mathbf{12}) &= a_{naive}(12) = 12^3 - 12^2 \\
&\Longrightarrow \\
a_{strassen\ winograd^*}(n) &= \frac{\frac{15 \cdot 12^2}{7-4} + 12^3 - 12^2}{12^{\operatorname{ld} 7}}n^{\operatorname{ld} 7} - \frac{15}{7-4}n^2 \\
&\approx 2.15n^{\operatorname{ld} 7} - 5n^2
\end{aligned}$$

$$
\begin{aligned}
m_{strassen\ winograd^*}(n) &= 7 \cdot m_{strassen\ winograd^*}\left(\frac{n}{2}\right) \\
m_{strassen\ winograd^*}(\mathbf{12}) &= m_{naive}(12) = 12^3 \\
&\implies \\
m_{strassen\ winograd^*}(n) &= \frac{12^3}{12^{\operatorname{ld} 7}} n^{\operatorname{ld} 7} \\
&\approx 1.61 n^{\operatorname{ld} 7}
\end{aligned}
$$

4 Implementation in the STXXL

The implementation of the stxxl::matrix<> container template focuses on efficiency on the external level. It supports the use of BLAS libraries for efficient internal matrix multiplication and addition.

BLAS is a widely used API standard for linear algebra subroutines on real and complex floating point variables, both 32 and 64 bit (per dimension). There are several implementations available, some of which are highly efficient and tuned for specific architectures (mostly the commercial ones). There also is an open source project for an efficient, self tuning BLAS library called ATLAS. [1]

Since this work deals with external matrix multiplication, the implemented internal matrix multiplication algorithm is kept simple; for efficient multiplication an efficient BLAS library is required.

4.1 Matrix Layout

stxxl::matrix<> implements a nested layout with two levels. It is similar to submatrix layout to a certain extent. The square submatrices correspond to disk blocks (i. e. $n'^2 = B$); they are called *blocks.* The elements in the blocks are laid out in row-major or column-major. The outer, matrix level consists of indirect references to the blocks stored in row-major layout. The blocks itself are not arranged in any specific pattern because they are not supposed to have a fixed position but to be moved dynamically between internal and external memory.

The blocks' fixed size allows for flexible allocation of disk space and especially internal space to load and store them. Furthermore, routines that operate on block level do not have to deal with different block sizes and become simpler. Row-major and column-major are the only layouts supported by the BLAS API standard, thus using BLAS libraries for internal matrix multiplication requires use of those layouts; else additional transformation would be required.

Blocks are represented by reference objects that can store internal and external memory addresses. The blocks are accessed via acquire and release calls to a block scheduler (cf. section 4.6) that takes care of swapping. It is also possible for a block to have no valid storage location at all, i. e. to contain no data, being not *initialized.* In the context of matrices, this is interpreted as containing only zeros. When matrices are newly created or explicitly set to zero, they do not have to be written to and read from disk but their blocks are simply made not initialized.

4.2 on Algorithms

Algorithms that work recursively need to partition input and output matrices into submatrices. In stxxl::matrix<>, references to the blocks are copied to the submatrix representations, whereas the blocks remain untouched. This way, partitioning causes no I/Os and no further indirections. Because recursive calls overwrite the output matrix, they have to take intermediate results stored in it into account. Most multiplication algorithms therefore are implemented with a multiply-and-add semantic, i. e. $C = A \cdot B + C$.

Alternatively, the recursive multiplication uses temporary output matrices, separating recursion and addition, but requiring additional I/Os to perform the addition.

Instead of copying the references, one could introduce another level of indirection, storing references and intervals to the matrices of references, but copying supports cache-obliviousness as discussed in section 3.5.

Parallelism is applied on the block level. If a BLAS library is used, exploiting parallelism is left to it. Usually, parallelizing as coarse as possible is desired, and matrix multiplication can easily be parallelized, e. g. by running the recursive calls in parallel. However, parallelizing above block level conflicts with the applied prefetching mechanism (see section 4.6). Due to dynamic load balancing, the order of block accesses can change. To use a great part of the cache for prefetching, it is necessary to schedule many block accesses in advance. Because matrix multiplication is computation intensive, the scheduled use corresponds to a long computation time that then can not be easily partitioned for load balancing.

As we will see in section 5.4.1, block operations do take long enough for effective parallelization.

4.3 Transposition

Transposition is implemented as a zero I/O operation. It works on matrix level only. The matrix of references to blocks is transposed and the block layout switched between row-major and column-major. Ease of this operation comes at the cost of additional switches for the block layout in the block level routines that is very low compared to rewriting the whole matrix.

4.4 Addition, Subtraction, Scalar Multiplication

There is a simple two-level non-recursive algorithm for addition, subtraction and scalar multiplication. On the matrix level it iterates over all blocks. Since these operations require nothing but a single scan, they are naturally cache-oblivious.

If possible, a BLAS library is used for block level addition and subtraction. Unfortunately the BLAS API defines no matrix addition. Instead, vector addition is used if the blocks use the same layout. If no BLAS library is available, a single cache-oblivious scan is employed on block level. In case of different block layouts, a scan over the elements of the output block is employed, but the input blocks may be accessed cache-inefficiently.

4.5 Multiplication

The implemented matrix multiplication algorithms differ only on matrix level; they use the same subroutines for block multiplication and addition. Multiplication of blocks is done either by a BLAS library or standard matrix multiplication with reordered loop nesting (see section 3.2). Not initialized blocks (see section 4.1) are handled explicitly, saving I/Os and computation.

None of the algorithms implements explicit caching, so they do not need to deal with the size of internal memory. They rely on the block scheduler (see section 4.6) to do (offline) caching in a best-effort manner.

4.5.1 Naive Matrix Multiplication

As discussed in section 3.1, naive matrix multiplication needs the minimal number of write accesses (cf. figure 7). It calculates the inner product that forms one block of the result at once and does not store any intermediate results on disk.

4.5.2 Recursive Matrix Multiplication

Recursive matrix multiplication uses naive matrix multiplication as base case for small inputs. The recursive calls are reordered to enhance temporal locality: Both calls that affect the same part of the output matrix are grouped together. The next call always uses the same part of one of the input matrices.

4.5.3 Strassen-Winograd Matrix Multiplication

The Strassen-Winograd matrix multiplication algorithm requires to partition the matrix into quadrants of equal size. Since we operate on matrix level, that means height and width in blocks have to be even. If they are not, the matrix is *dynamically padded* during partitioning with a row or column of zero blocks respectively. The zero blocks are not initialized, thus do not cause additional I/Os directly.

Contrary to the conclusion found by Chatterjee et al. [4], padding still causes algebraic and I/O overhead, because the preadditions create non-zero blocks nevertheless (see figure 5). For Strassen-Winograd to be effective, there must not be too much padding. The effect has not been investigated; no theory about a good decision mechanism is known. The implementation just pads whenever necessary when Strassen-Winograd is requested. For matrices that require lots of padding, this may be inefficient.

Strassen-Winograd matrix multiplication uses recursive matrix multiplication as base case for small inputs.

Preadditions There are two implementations of the preaddition phase. The first is straightforward. It calculates the intermediate sums one after the other. The second interleaves calculation of the intermediate sums. It calculates the group of all first blocks of $S_1, \ldots, S_4$, then the group of all second blocks and so on. In the same way, $T_1, \ldots, T_4$ are calculated. Because the grouped blocks share common summands, doing so increases temporal locality in accessing the input matrices and thus helps reducing I/Os.

$$A : \begin{array}{|c|c|}\hline * & 0 \\ \hline 0 & 0 \\ \hline\end{array} \qquad B : \begin{array}{|c|c|}\hline * & * \\ \hline * & * \\ \hline\end{array}$$

input: A				recursive multiplications					input: B			
$*$	0	0	0						$*$	$*$	$*$	$*$
A_{11}	A_{12}	A_{21}	A_{22}						B_{11}	B_{12}	B_{21}	B_{22}
$+$				A_{11}	$*$	●	$*$	B_{11}	$+$			
	$+$			A_{12}	0	$\cdot$	$*$	B_{21}			$+$	
		$+$	$+$	S_1	0	$\cdot$	$*$	T_1	$-$	$+$		
$-$		$+$	$+$	S_2	$*$	●	$*$	T_2	$+$	$-$		$+$
$+$		$-$		S_3	$*$	●	$*$	T_3		$-$		$+$
$+$	$+$	$-$	$-$	S_4	$*$	●	$*$	B_{22}				$+$
			$+$	A_{22}	0	$\cdot$	$*$	T_4	$-$	$+$	$+$	$-$

input: A				recursive multiplications					input: B			
$*$	0	0	0						$*$	$*$	$*$	$*$
A_{11}	A_{12}	A_{21}	A_{22}						B_{11}	B_{12}	B_{21}	B_{22}
$+$				A_{11}	$*$	●	$*$	B_{11}	$+$			
	$+$			A_{12}	0	$\cdot$	$*$	B_{21}			$+$	
		$+$		A_{21}	0	$\cdot$	$*$	B_{11}	$+$			
			$+$	A_{22}	0	$\cdot$	$*$	B_{21}			$+$	
$+$				A_{11}	$*$	●	$*$	B_{12}		$+$		
	$+$			A_{12}	0	$\cdot$	$*$	B_{22}				$+$
		$+$		A_{21}	0	$\cdot$	$*$	B_{12}		$+$		
			$+$	A_{22}	0	$\cdot$	$*$	B_{22}				$+$

Figure 5: Illustration of preadditions of the Strassen-Winograd algorithm (above) compared to the recursive standard algorithm (below).
'0' represents zero blocks, '*' non-zero blocks. Block multiplications have to be performed whenever two non-zero blocks are to be multiplied ('•'). Though preadditions, Strassen-Winograd does four block multiplications while recursive standard multiplication needs only two. Other scenarios with zero blocks show similar results.

Postadditions There are two implementations of the postaddition phase. The first mixes postadditions with the recursive multiplications, exploiting the multiply-and-add semantic to save separate additions and thereby I/Os. That schedule requires to do only six of the seven postadditions explicitly and needs temporary storage the size of only one quadrant of the result matrix to store intermediate results from the recursive multiplications.

The second interleaves calculation of the sums the same way as interleaved preadditions.

4.5.4 Multi Level Strassen-Winograd

When the Strassen-Winograd algorithm is used for multiple levels of recursion, intermediate results calculated in the outer preaddition phase are used by the inner preaddition phase and so on until the base case is reached. Because the intermediate results are the same size as the input matrices on each level, they are written to and read from disk to pass them from one level to the other. Similarly, the postaddition phases pass intermediate results from inner to outer levels. Because matrix additions need to access each element just once and do not depend on the order of operations on single elements, I thought of a possibility to save I/Os by not storing intermediate results on disk.

Given some granularity, e. g. single elements or single blocks, start the innermost level of preadditions. Whenever a grain of an intermediate result is needed, calculate it just before its use. This eliminates need to store many grains and thus use slow memory for them.

Unfortunately, every intermediate result is used a constant number of times by each level of recursion, causing it to be requested and thus computed a number of times growing exponentially with the number of levels of inner recursions. It is possible to avoid the exponential computation cost though. Grouping together all operations on one level of recursion that use a particular grain of intermediate result allows to calculate it only once. Use of the results of all operations in the group immediately afterwards however, forces to join a constant number of groups in the next lower level. The result is a group size growing exponentially with the number of levels of outer recursions.

The number of grains of intermediate results that need to be stored at once increases linearly with the size of groups. Processing all levels of recursion at once, with the base case of the multiplication as grains would require just as much temporary storage as normal recursion. To save I/Os only so many levels should be processed at once that all intermediate results between those levels can still be held in internal memory. By this restriction cache-obliviousness is lost.

stxxl::matrix<> features two experimental implementations of Strassen-Winograd on multiple levels, one with single elements as grains, one with blocks as grains. They are implemented using recursive template-based structures, where the levels to process are the template parameter. The outer levels hold seven structures of the next lower level, one for each recursive multiplication. They perform preadditions and postadditions and hand the results to the next level. A specialization forms the innermost level. It stores intermediate results and calls recursive multiplication.

To use these structures, one object of the desired level has to be created and "fed" with portions of input. After the recursive multiplications, the result can be read in portions. The portions are collections of those grains that are used by one group of preadditions. The feeding mechanism keeps the groups of preadditions and postadditions together and tells inner levels when portions of intermediate results are ready.

4.6 Caching and Prefetching through the Block Scheduler

All implementations of matrix algorithms use a block scheduler. The block scheduler takes care of swapping the blocks in and out including caching, prefetching, and associated memory management. The block scheduler was developed for use by the matrix algorithms, but has been written generically, to be usable by any algorithm with a deterministic, data-oblivious block access pattern.

Different scheduling strategies are available, one of which can be used online. By doing a simulation run, the sequence of block accesses can be recorded as a prediction sequence without doing any I/O. The prediction sequence can afterwards be used for offline scheduling.

Wether a block will be initialized[5] is tracked during simulation. Other than that, the computing algorithm has to reproduce the same sequence of block accesses during simulation as it will on the real run independent of the blocks' contents. A simulation flag of the block scheduler allows to skip expensive computations.

4.6.1 Access Model

Scheduling is based on a simple access model. Before accessing a block, an acquire call for that block has to be made. It is then held accessible in internal memory, until a release call for it is received. A parameter of the release call states if its contents have been changed. This results in a sequence of acquire and release calls for different blocks.

Each time a release follows an acquire call, an implicit timestep is assumed; otherwise the calls are seen as concurrent and assigned the same timestamp. Different lengths of time are not taken into account, but each timestep is considered to be significant for scheduling.

It makes no sense to have an acquire call before a release call with no timestep in between because if the acquired block is not used in the meantime they could be swapped, allowing to use internal memory previously reserved for the released block to be used for the other.

4.6.2 LRU Scheduling

Least Recently Used (LRU) is the only online strategy. The implementation uses an addressable FIFO queue to determine the swapped in block b that has been least recently used. Whenever a swapped out block a shall be acquired, b is cleaned (i. e. written if dirty), its internal memory assigned to a and a is read.

No overlapping between I/O and computation is possible.

[5] i. e. contains valid data, see section 4.1

4.6.3 LFD Scheduling

Longest Forward Distance[6] (LFD) is read-optimal. It is not I/O-optimal, because it does only consider wether a block is dirty when it is compared to another block with the same next-use-time.

The LFD implementation is based on the *next-use-time*, i. e. the timestamp of the next acquire call for a block. It uses the prediction sequence to find the next-use-time to every release call for a block. The next-use-time is the base of the priority assigned to every block, supplemented with its dirty flag. Swapping works just as with LRU, with the only difference that the block to swap out is determined by an addressable priority queue using the mentioned priority.

4.6.4 LRU Scheduling with Prefetching

LRU with prefetching follows from a simple idea: Swap in[7] all blocks that will be accessed up to a time[8] as far as possible in the future. Do that as soon as possible.

Following this idea inevitably leads to LRU replacement:
Let t_n be the time of the current or last processed call, and let t_s be the time of the call up to that is known that all accessed blocks are already swapped in or are being swapped in. All calls until t_s inclusively can thus be processed without issuing any further reads or writes. For a block b let $t_{next\_use}(b)$ be the next time after t_n at that b is acquired ($t_{next\_use}(b) = \infty$ if b is not acquired any more).

Consider scheduling before reaching the end of the prediction sequence. t_s can be moved forward, issuing background read and write requests, until $t_s + 1$ points to an acquire call for a swapped out block and no internal memory is or can be made available. Whenever a block b is released, one of the following cases applies:

case $t_{next\_use}(b) \leq t_s$**:** b has to be kept swapped in because it will be accessed soon.

case $t_{next\_use}(b) > t_s + 1$**:** b can be swapped out, making its reserved internal memory available. With it, t_s can be be moved forward. b is the only block not accessed during $(t_n, t_s]$ that is swapped in; thus b will have been least recently used at t_s.

case $t_{next\_use}(b) = t_s + 1$**:** Can not happen. As mentioned above, $t_s + 1$ points to an acquire call for a swapped out block, but b is swapped in.

By issuing read and write requests to run in background, overlapping of I/O and computation is possible. Computation only waits for I/O if a block shall be acquired but the read request to swap it in has not finished yet.

Waiting for I/O can only happen if the algorithm has been I/O-bound for a long period of time: The read request has been issued when t_s reached the current call. Now t_n points to the current call. Since only blocks that are accessed between t_n and t_s are held in cache $t_n \leq t_s - \langle \textit{cache size} \rangle$ holds. Thus, at least as many blocks as fit in cache have been used since issuing the read request.

[6] Longest Forward Distance is also known as Longest Future Distance.

[7] Blocks that are not initialized of cause need not to be read. However, internal memory has to be reserved to for them.

[8] "Time" in this context means a position in the prediction sequence.

5 Test Results and Findings

5.1 Setup

Tests have been run on two computers:

i10pc125 with four Intel Xeon-X5550 processors (eight cores) and eight disks. Intel mkl was used as BLAS library. The installed OS was OpenSuse 11.1 Professional.

i10pc126 with four AMD Opteron MagnyCours-6168 (48 cores) and two disks. AMD acml was used as BLAS library. The installed OS was Ubuntu 10.10 Server

Not all of the machines RAM was used. Caching (including prefetching and acquired blocks) has been limited by the block schedulers mechanisms to use up to a certain amount of internal memory[9]. The amount is specified with the respective test cases. Some more internal memory was used by administrative data structures, but only in the magnitude of megabytes. The BLAS libraries may also allocate some internal memory, but since they are only called for single blocks, their usage should not exceed the magnitude of megabytes either.

Disc access has been performed via DMA, thus no caching of reads and writes appears outside the disks hardware, in particular not by the operating system.

During the tests, the machines were otherwise idle. The block schedulers cache has been cleared before and at the end of multiplication, so that all input came from disk and all output went to disk.

In all experiments, the matrices consisted of 64 bit floating point variables (C++s build in type *double*).

If not stated otherwise, block order was 1568, resulting in a block size of 18.8 MiB. The algorithm variants in use were Strassen-Winograd with interleaved preadditions and postadditions mixed in recursion, and multi level Strassen-Winograd up to three levels grained on blocks.

Matrix orders have been limited to powers of two times block order because of the efficiency restrictions stated in section 4.5.3.

5.2 Hardware Considerations

As discussed by Toledo [10], it is reasonable to assume that computers today have an external memory by internal memory ratio between 10 and 1000. This is because of the price ratio fluctuating around 100.[10] Computers with less internal memory could have it increased, and thus their performance boosted, relatively cheap regarding their total cost.

Given this ratio, one can conclude that of a matrix of size $n, m \geq 1000$ or more, at least one row or column fits into internal memory. Similarly, of a matrix in submatrix layout with $N, M \geq 1000$ (i. e. 1000 submatrices per row/column) a whole row or column of submatrices fits into internal memory. Under these conditions some operations like matrix-vector multiplication or row-major to block layout conversion can easily be implemented to be I/O-efficient.

[9]Exempt are the tests with pure internal multiplication. In that case all three matrices have to be held in internal memory.

[10]This does of cause not apply to higher levels of cache.

5.3 Labels Used in This Section

overall-time Time taken by the whole multiplication, including flushing the output from cache.

I/O-time Sum per disk of time a read or write is processed.

parallel-I/O-time Time at least one read or write is processed on any disk.

I/O-wait-time Time that the computation had to wait for disk accesses to finish.

computation-time = overall-time − I/O-wait-time

parallel-I/O-ratio = I/O-time / parallel-I/O-time, i. e. the average number of I/Os run in parallel.

5.4 Block Timings

First of all, tests with very small blocks (32×32 elements) had been run. Matrix order and amount of internal memory have been adjusted, so that the number of blocks per matrix and that fit in internal memory remained equal to the real tests. The longest of these tests took only some minutes for very large numbers of blocks, showing that administrative overhead is small compared to the actual arithmetic.

The following results for block operations are derived from multiplying matrices of order 100352 on i10pc125, once with the recursive standard algorithm and once with the Strassen-Winograd algorithm using 15GiB internal memory.

5.4.1 Multiplication and Addition

The recursive standard matrix multiplication algorithm took 26388 seconds computation-time and performed 262144 block multiplications. Thus it takes about *0.101 seconds per block multiplication.* Strassen-Winograd performed 117649 block multiplications and 529914 block additions in 17211 seconds computation-time. This leaves 5368 seconds for the additions, resulting in *0.010 seconds per block addition.*

Additions show to be only ten times faster than multiplications, not over a thousand times faster as would be expected from theoretical operation counts. This effect may be explained with parallelizing on block level, so inter-thread communication has to be done for every block addition and multiplication. Another possible explanation is that for both operations, the operands have to be fetched from outer to inner levels of cache. Even though addition requires only a single scan, its throughput can not exceed the speed of the memory the operands are in at the beginning of the addition.

5.4.2 I/Os and Compute-Boundedness

I/O-time of recursive standard matrix multiplication was 9748 seconds for 54896 I/Os and of Strassen-Winograd 38314 seconds for 182860 I/Os. Thus it takes *0.178 or 0.210 seconds per I/O* respectively. The difference probably arises from Strassen-Winograds larger memory consumption increasing disk head movement and thereby seek time.

Parallel-I/O-time of recursive standard was 6429 seconds, resulting in a parallel-I/O-ratio of 1.516. Parallel-I/O-time of Strassen-Winograd was 9703 seconds, resulting in a parallel-I/O-ratio of 3,949. Remember that i10pc125 has 8 disks, thus parallel-I/O-ratios of 8 are theoretically possible.

The above result for the recursive standard algorithm is not surprising, since it is clearly compute-bound, as can be seen in figure 8.

The Strassen-Winograd algorithm is not compute-bound all the time, because it has a significant I/O-wait-time. It is also not I/O-bound all the time, else it would have a much higher parallel-I/O-ratio and its parallel-I/O-time would match the overall-time. The obvious reason is that there are I/O-bound phases and compute-bound phases.

The phases are easily identified: Preadditions require four input and four output blocks to perform four additions. With the previously calculated times, the I/Os take 0.210 seconds on 8 disks in parallel ($(4+4) \cdot 0.210\text{s}/8$), while the additions take 0.040 seconds. Preadditions require sequential access to all blocks of the submatrices, so they cannot run from cache in the outer levels of recursion. Postadditions behave similar. Thus, those preadditions and postadditions are highly I/O-bound.

As submatrices become smaller on inner levels of recursion, they can stay in internal memory and multiplications can be performed without further I/Os. Also, the number of multiplications is $n^{ld7} \approx n^{2.8}$ for $n \times n$ block submatrices. Even for 2×2 block submatrices seven multiplications and 15 additions are necessary (0.857 seconds) while only 16 parallel I/Os (0.420 seconds). For larger submatrices, multiplication becomes even more compute-bound. Multiplications including preadditions and postadditions on submatrices that fit in internal memory thus are highly compute-bound.

5.5 Matrix Multiplication

Figure 6 shows block operation counts for recursive standard matrix multiplication and Strassen-Winograd variants. As expected, the number of multiplications done by the Strassen-Winograd variants grows asymptotically slower than for the standard algorithm.

Recursive standard multiplication does no explicit additions at all because they are hidden in the multiply-and-add semantic (cf. section 4.2). Strassen-Winograds addition count asymptotically approaches five times the multiplication count as predicted in section 3.6.5.

The addition count of the multi level variant does not grow that evenly because of its limitation to particular levels. The break at 25088 (four levels of recursion) corresponds to the three-level limit plus one level base case. Its multiplication count differs slightly from that of regular Strassen-Winograd because of its different base case.

Figure 7 shows I/O counts for different algorithms. Theoretical I/O-counts are 3.5–5 times higher than the measured values probably because the assumed number of intermediate results to be stored at once is to high.

The break at order 12544 corresponds to the point the amount of data becomes to large for internal memory. Strassen-Winograd fills the internal memory with smaller instances due to its need for temporary storage, thus starts of with more I/Os at order 25088. Its slightly smaller growth ($\Theta(n^{\mathrm{ld}\,7})$ instead of $\Theta(n^3)$) is hardly visible.

Multi level Strassen-Winograd shows leaps in I/O requirement when it needs another level of recursion because then it has to store the intermediate results.

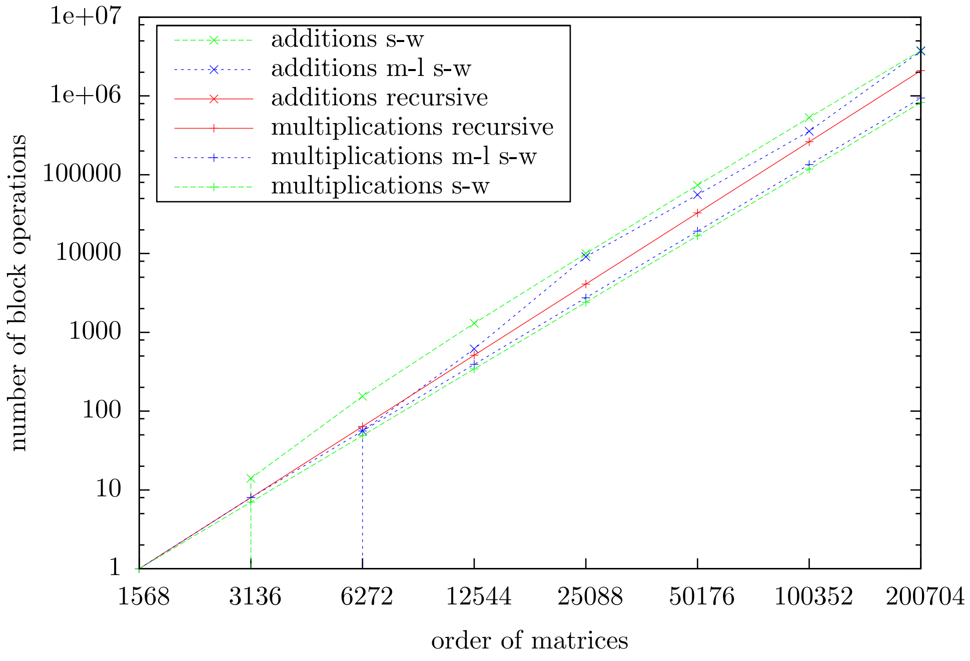

Figure 6: Number of block operations for different matrix multiplication algorithms by matrix order. Results have been obtained on i10pc125 using 15 GiB internal memory.

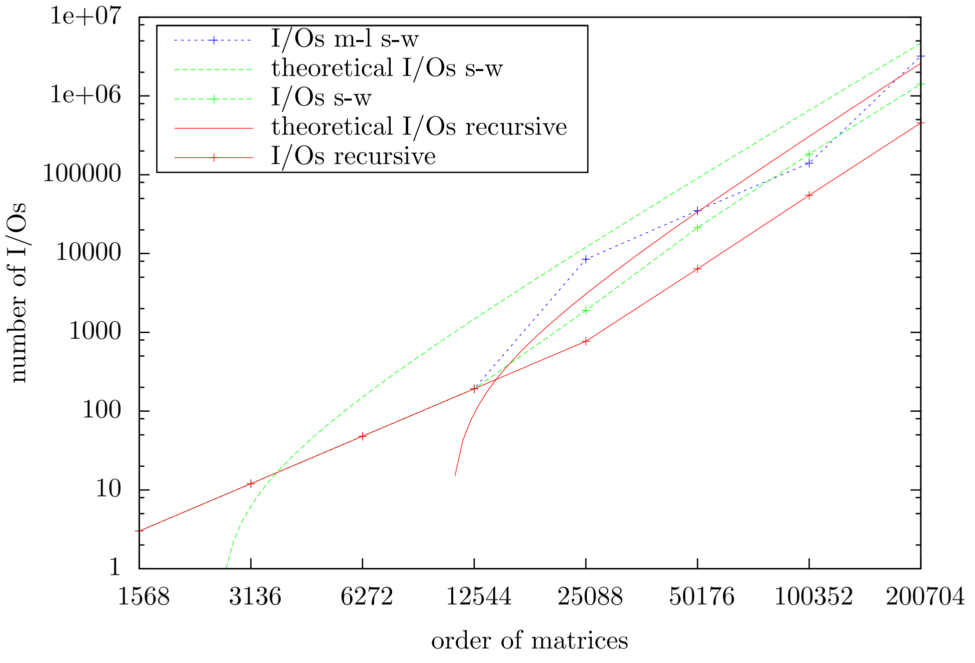

Figure 7: Number of I/Os for different matrix multiplication algorithms by matrix order. Results have been obtained on i10pc125 using 15 GiB internal memory.

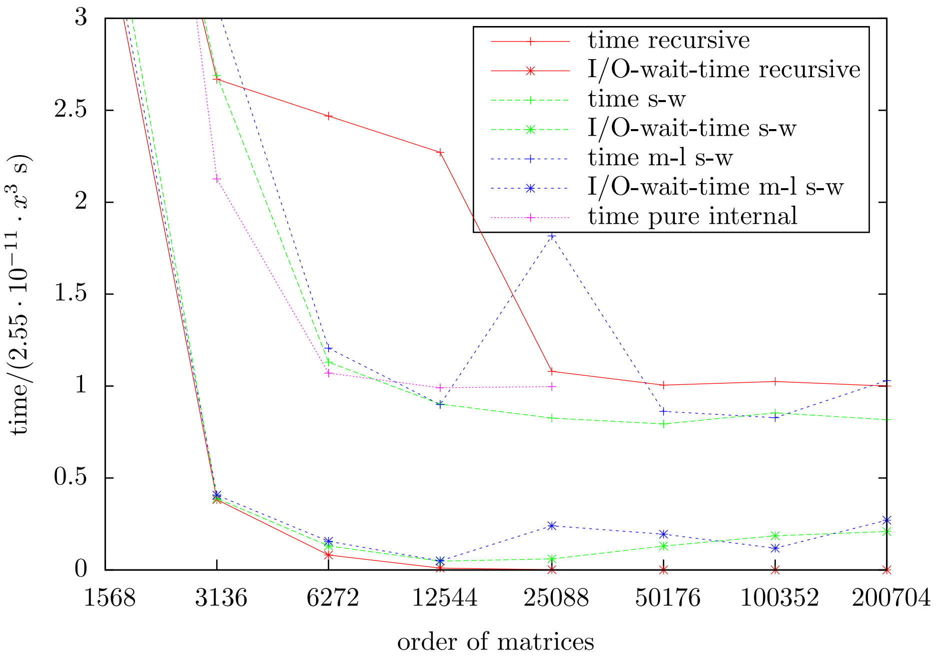

Figure 8: Times for different matrix multiplication algorithms by matrix order. Results have been obtained on i10pc125 using 15 GiB internal memory. The pure internal multiplication could use up to 64 GiB internal memory

Outside of the leaps, it grows only linear with the input size, confirming that it has to access the input and output only once and store no intermediate results on disk for those levels.

Figures 8 and 9 show overall-time, overlapping-I/O-time, and I/O-wait-time for different algorithms. The recursive standard algorithm is completely compute-bound. Strassen-Winograd has a significant I/O-wait-time due to its I/O-bound preaddition and postaddition phases. It is still faster because it saves a lot in computation. For matrix side length 100352, the multi level variant's save in I/Os results in decrease of I/O-wait-time. The gain is reduced by the higher computation-time. Anyway, it is still a little faster for this instance.

Conceptually assuming n^3 floating point operations, Strassen-Winograd did 48 GFLOPS for matrix order 200704 on i10pc125.

5.6 Scheduling

As can be seen in figure 10, LRU caching causes more I/Os than LFD caching. However, the advantage fades for very large matrices. Furthermore, for some access sequences, LFD inhibits prefetching, as can be seen in figure 11. Access sequences like that occur for example on intermediate results in successive preaddition phases.

As discussed in section 5.4.2, Strassen-Winograd has phases of compute-boundedness (multiplications) and phases of I/O-boundedness (preadditions

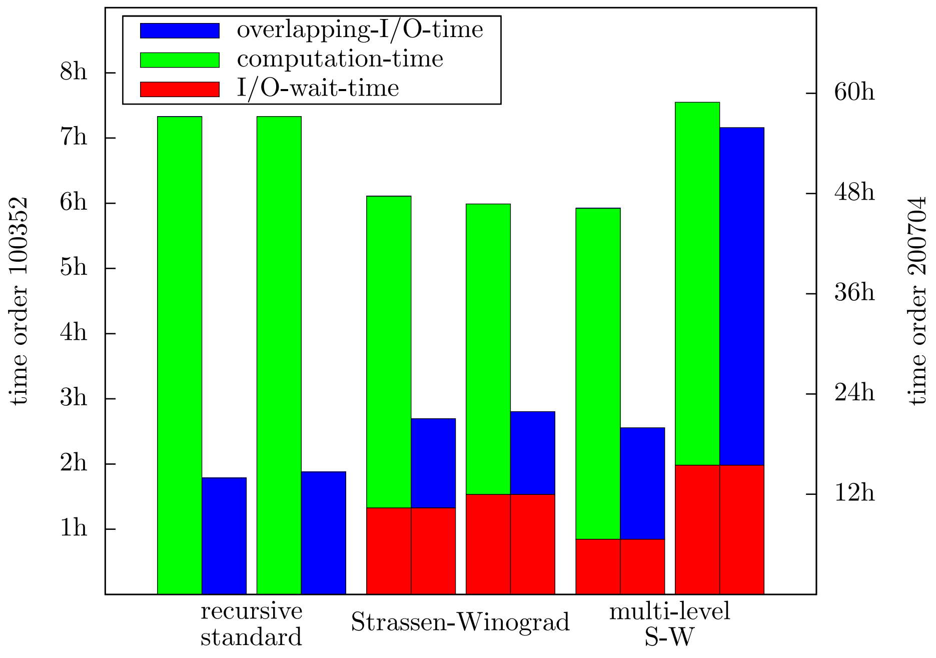

Figure 9: Times for different matrix multiplication algorithms. Left values are for order 100352, right values for order 200704. Results have been obtained on i10pc125 using 15 GiB internal memory.

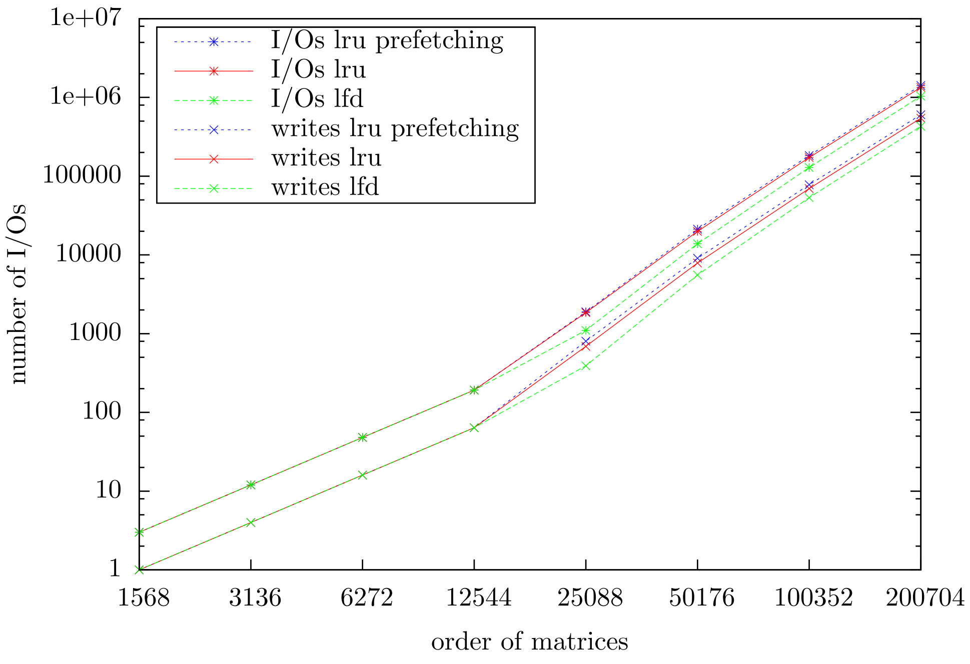

Figure 10: Number of I/Os for different scheduling algorithms by matrix order. Results have been obtained on i10pc125 using 15 GiB internal memory. The break at 12544 corresponds to the point the amount of data becomes to large for the cache.

	timestep	1	2	3	4	5	6	7	8
	accessed block	a	b	c	d	a	b	c	d
LFD	cacheblock 1	**a**	a	a	a	**a**	a	a	a
	cacheblock 2	b	**b**	**c**	**d**	$d \rightarrow b$	**b**	**c**	**d**
LRU	cacheblock 1	**a**	$a \rightarrow c$	**c**	$c \rightarrow a$	**a**	$a \rightarrow c$	**c**	c
	cacheblock 2	b	**b**	$b \rightarrow d$	**d**	$d \rightarrow b$	**b**	$b \rightarrow d$	**d**

Figure 11: Cache contents for LFD (prefetching if possible) and LRU with prefetching. "$x \rightarrow y$" indicates possibility for overlapping I/O. Notice how LFD uses only the last cache block for all missing blocks, inhibiting overlapping of I/O and computation.

and postadditions).

During compute-bound phases, the exact caching strategy is of no importance as long as it prefetches the required blocks in time. Idleness of I/O can be used to fill the cache for the next I/O-bound phase. It is advantageous to start an I/O-bound phase with the cache filled with the blocks required in the beginning of the phase. That is exactly what LRU with prefetching tries to achieve.

More interesting is what happens during I/O-bound phases. LRU replacement, even with prefetching if possible, is no good alternative because it may inhibit overlapping between I/O and computation as illustrated in figure 11.

5.7 Conclusions

We have seen that efficient external matrix multiplication is possible, though not trivial. Some internal memory for use as cache is helpful, but improved I/O bandwidth through multiple disks is more important. By using advanced algorithms, time can be saved if I/O bandwidth is high enough.

Recursive standard matrix multiplication evidently is a robust external algorithm. Its pure compute-boundedness, low overlapping-I/O-time, and low parallel-I/O-ratio indicate that it will run well even with less disks and less internal memory. On the other hand, improving internal multiplication would proportionally increase its speed without need for adjustments on I/O. Its simple access patterns would ease more coarse grained parallelization and allow for prefetching without explicit simulation.

Strassen-Winograds save in computation works well in the implementation as external algorithm. Since it is used on matrix level, the overhead by its more complicated structure is negligible. The preaddition and postaddition phases pose a problem, because the need to stream all data renders it highly I/O-bound during these phases. Even caching with lots of internal memory can only partially reduce this problem. Therefore, the speed and number of disks is most critical to the effectiveness of Strassen-Winograd. Anyway, in the test setup with mediocre tuning, Strassen-Winograd enabled to break the speed limit set to standard matrix multiplication by the speed of internal multiplication.

For matrix order 100532 the multi level variant of Strassen-Winograd successfully reduced the number of I/Os, again slightly decreasing the overall-time. For some other matrix orders however, it increased the number of I/Os and thereby overall-time. Its sensitivity to the number of recursions and its extensive need for internal memory complicate its use and require more tuning. In the current state, it is to be seen as an academic approach only.

6 Future Work

As explained in section 4.5.3, Strassen-Winograd with padding may become inefficient for matrices that require lots of padding. Another approach is dynamic peeling [10], i. e. one level of recursive standard multiplication partitioning into a main part with even side length and a single row or column respectively. One could also partition so that the main part's side lengths are powers of two. Research about this topic is missing.

As mentioned in section 5.6, neither LRU nor LFD provide optimal replacement and prefetching. Improvements may be possible by combining the idea that led to LRU with prefetching (to prefetch as early as possible, see section 4.6.4) with LFD-like replacement decisions if there is more than one block pending to replace. The replacement decision could be postponed until just before I/O falls idle to maximize the number of replaceable blocks. Another idea is to reserve a fixed partition of the cache for prefetching, so LFD can be applied to the rest but prefetching is always possible.

In section 4.2 the problematic of combining prefetching and load balancing was described. To apply more coarse grained parallelization to external algorithms, solving this problem will be helpful.

The existence of I/O-bound and compute-bound phases raises one question: Can these phases be scheduled to overlap (possibly in combination with coarse parallelism) so that complete overlapping between I/O and computation is achieved?

A Further Graphics of Test Results

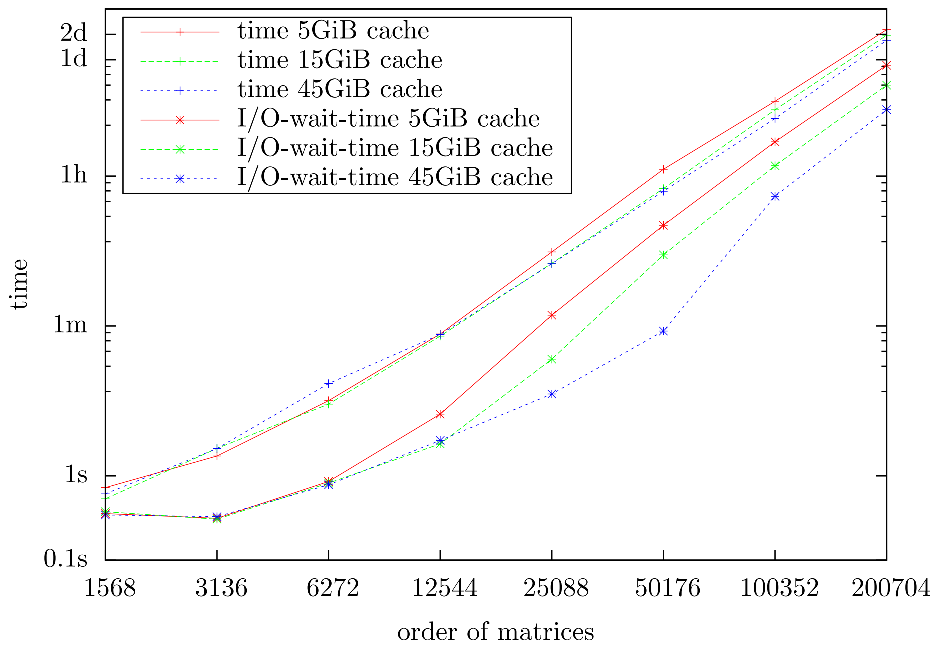

Figure 12: Times for different cache sizes by matrix order. Results have been obtained with Strassen-Winograd on i10pc125.

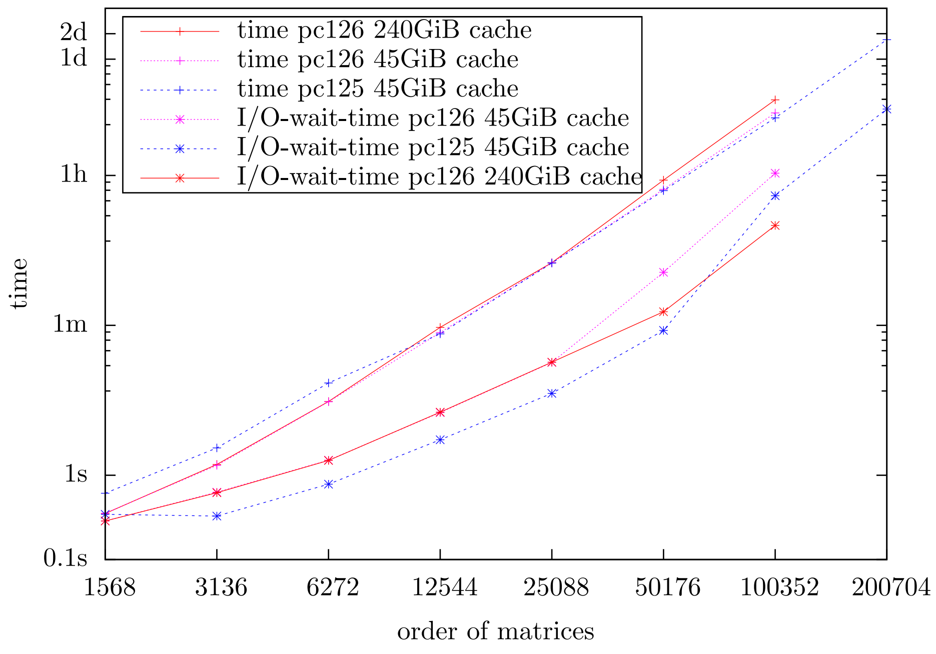

Figure 13: Times for different computers and cache sizes by matrix order. Results have been obtained with Strassen-Winograd.

Despite its larger number of processor cores, i10pc126 seems to be slower than i10pc125. Possibly the Intel mkl is just better than AMD acml.

Strangely, more cache makes the algorithm slower. Maybe the machines distributed memory combined with mid grained parallelism causes more copying between the processor groups. Or maybe administrative overhead gets that big.

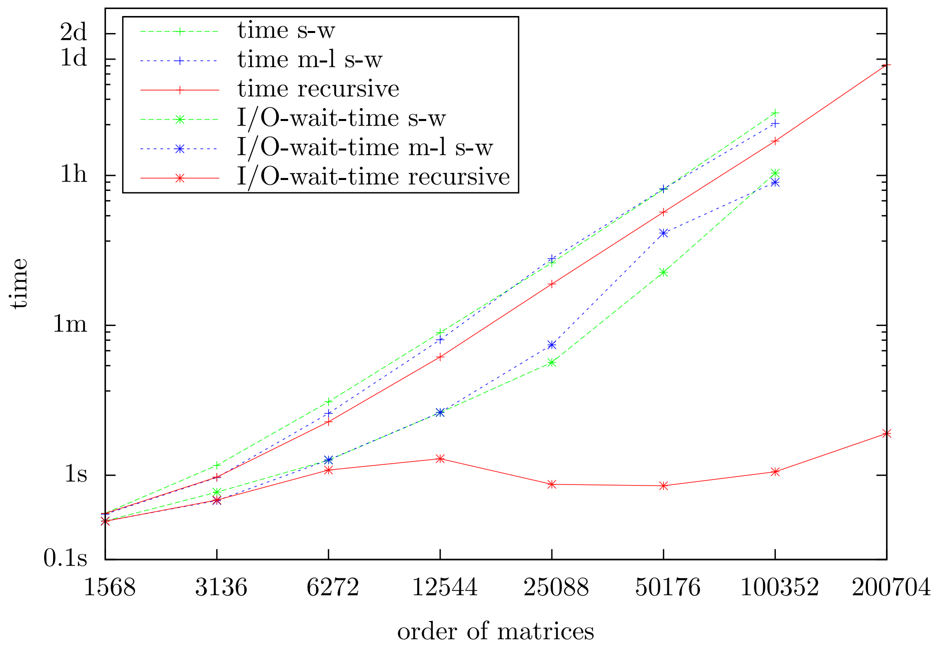

Figure 14: Times for different matrix multiplication algorithms by matrix order. Results have been obtained on i10pc126 using 45 GiB internal memory.

This tests show that the I/O with only two disks is to slow for Strassen-Winograd to be efficient. The reduced computation cost can not make up for the I/O-bound preaddition and postaddition phases.

For order 200704 Strassen-Winograd and the multi level variant could not finish, because it was not enough disk space available.

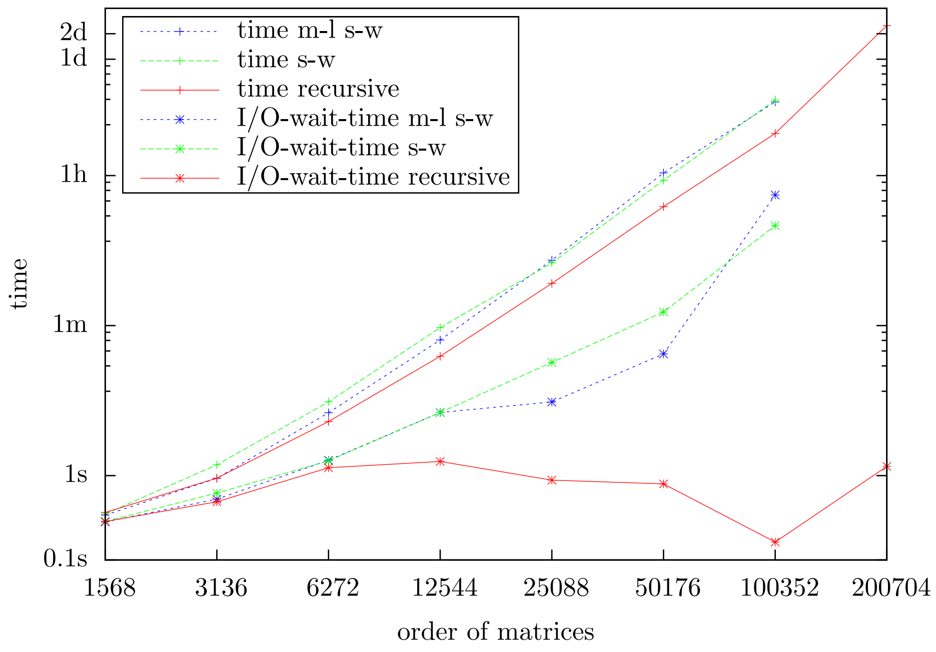

Figure 15: Times for different matrix multiplication algorithms by matrix order. Results have been obtained on i10pc126 using 240 GiB internal memory.

Again, as seen for 45 GiB cache in figure 14, I/O with only 2 disks is to slow for Strassen-Winograd to be efficient. The reduced computation cost can not make up for the I/O-bound preaddition and postaddition phases.

For order 200704 Strassen-Winograd and the multi level variant could not finish, because it was not enough disk space available. Recursive standard multiplication took longer, because even without temporary storage the disks were almost full.

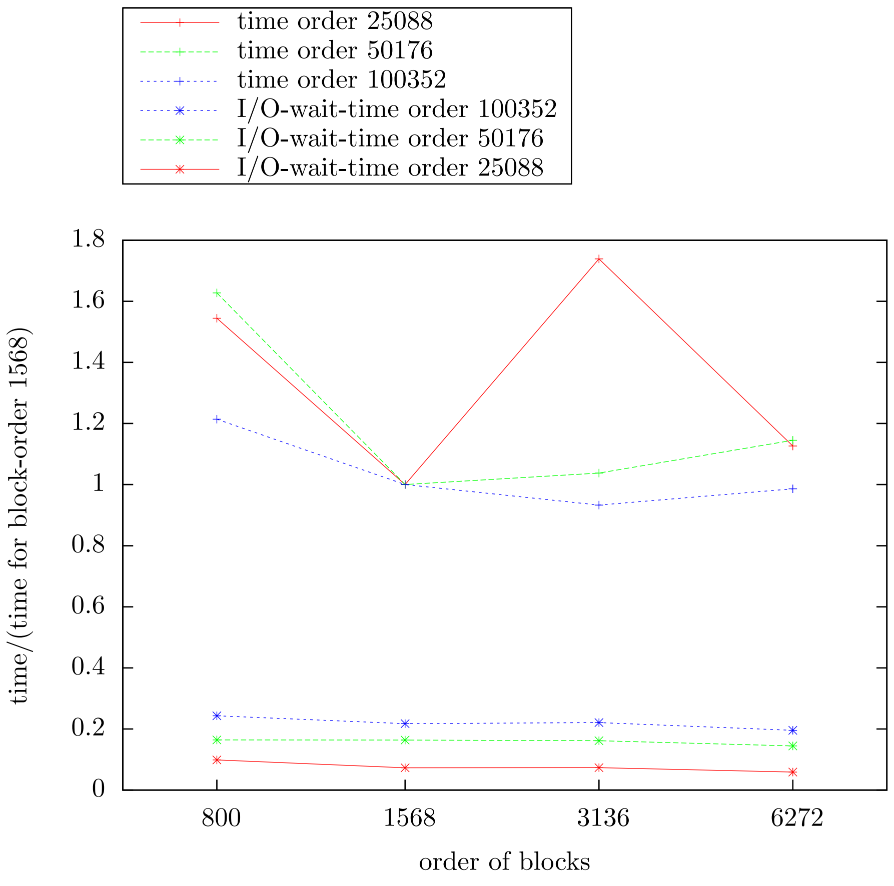

Figure 16: Times for different matrix orders by order of blocks. Results have been obtained multiplying matrices by Strassen-Winograd on i10pc125 using 15 GiB internal memory.

Larger blocks reduce overhead in disk accesses and parallelization but also reduce the number of levels Strassen-Winograd can be applied. For small and mid-sized matrices, the chosen block size seems to be best. For larger matrices, larger blocks seem to be better. The reason probably are the I/O-bound preaddition and postaddition phases, that benefit from faster disk accesses and are more significant for larger matrices.

The high value for matrix order 25088 and block order 25088 can not be explained yet and calls for further investigation.

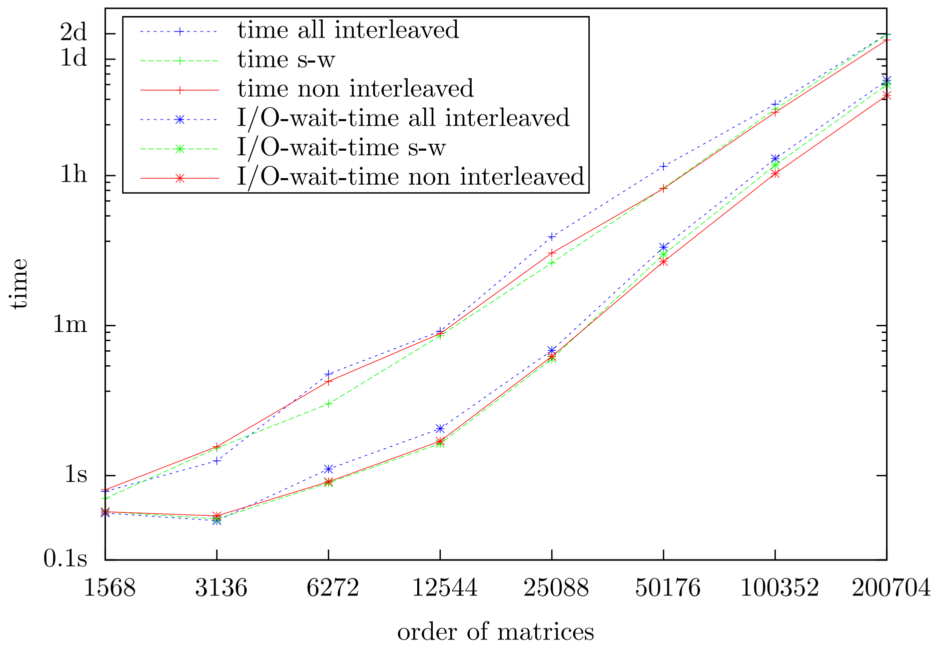

Figure 17: Times for different implementations of Strassen-Winograd by matrix order. Shown are the implementation with preadditions and postadditions interleaved, the implementation without interleaved additions and the implementation with preadditions interleaved and postadditions mixed in with recursion. Results have been obtained on i10pc125 using 15 GiB internal memory.

Interleaving postadditions seems to be no good.

Other than expected, interleaving preadditions is better for smaller matrices and worse for larger ones. That shows that the increased temporal locality helps on internal multiplication but somehow conflicts with disk access or scheduling.

References

[1] Automatically Tuned Linear Algebra Software (ATLAS). *http://math-atlas.sourceforge.net/.*

[2] Michael D. Adams and David S. Wise. Fast additions on masked integers. *SIGPLAN Not.*, 41:39–45, May 2006.

[3] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, September 1988.

[4] S. Chatterjee, A.R. Lebeck, P.K. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. *Parallel and Distributed Systems, IEEE Transactions on*, 13(11):1105 – 1123, November 2002.

[5] Hong Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 326–333, New York, NY, USA, 1981. ACM.

[6] A. C. McKellar and E. G. Coffman, Jr. Organizing matrices and matrix operations for paged memory systems. *Commun. ACM*, 12:153–165, March 1969.

[7] Rajeev Raman and David Stephen Wise. Converting to and from dilated integers. *IEEE Transactions on Computers*, 57:567–573, 2008.

[8] H. Rubinstein and J. D. Rutledge. High order matrix computations on the UNIVAC. In *Proceedings of the 1952 ACM national meeting (Pittsburgh)*, ACM '52, pages 181–186, New York, NY, USA, 1952. ACM.

[9] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969. 10.1007/BF02165411.

[10] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. *Dimacs Series In Discrete Mathematics And Theoretical Computer Science*, 1999.

[11] Ferad Zyulkyarov. Cache efficient matrix multiplication. *http://blog.feradz.com/2009/01/cache-efficient-matrix-multiplication/*, Published January 16, 2009.