Cache oblivious matrix multiplication using an element ordering based on the Peano curve

Michael Bader and Christoph Zenger

Institut für Informatik der TU München, Boltzmannstr. 3, 85748 Garching, Germany

Abstract

One of the keys to tap the full performance potential of current hardware is the optimal utilisation of cache memory. Cache oblivious algorithms are designed to inherently benefit from any underlying hierarchy of caches, but do not need to know about the exact structure of the cache. In this paper, we present a cache oblivious algorithm for matrix multiplication. The algorithm uses a block recursive structure, and an element ordering that is based on Peano curves. In the resulting code, index jumps can be totally avoided, which leads to an asymptotically optimal spatial and temporal locality of the data access.

Key words: cache oblivious algorithms, matrix multiplication, space filling curves
PACS:

0 Introduction

The most important data structures used in linear algebra algorithms are vectors and matrices or, more general, multidimensional arrays. The elements of these arrays have to be mapped to a linear memory space, such that all elements are stored in an address interval. Typically, algorithms are organized as loops over the array indices. However, even if an index is incremented only by 1, the respective address in memory may jump to a far away location. Jumps in the address space should be avoided on modern computer architectures, because the access of a distant element might cause a cache miss and thus take much more time than the access of a neighbouring element. For matrices, a row-wise or column-wise storage scheme is most often used. Then, even simple algorithms like the multiplication of two square matrices will cause frequent jumps in the address space. A lot of research is devoted to modify standard algorithms to overcome these problems, and improve performance on modern hardware.

In contrast, many fundamental algorithms in computer science are based on data structures which do not allow jumps in the address space. The most famous and extensively studied example is the Turing machine where the read/write head can only move by one position in every step. Another example is the push down automaton. Its basic data structure, the *stack* (in the original German notation “Keller”) was introduced in a famous paper by Bauer and Samelson [1]. Only two operations are allowed on a stack: *push* to store data on top of the stack, and *pop* to retrieve the topmost data. If either the band of a Turing machine or the stack of a push down automaton is directly mapped to the memory of a computer, it is very clear that the memory access will always remain local without any jumps.

This raises the question if algorithms in linear algebra can also be based on data structures that avoid jumps in the address space. In this paper we investigate the probably most basic nontrivial algorithm of linear algebra, the multiplication of two square matrices. We want to emphasize that it is not the aim of this paper to produce the fastest algorithm for matrix multiplication on a specific computer architecture. In contrast we want to demonstrate that it is possible to construct an algorithm where memory addresses change only with stepsize one, which also eliminates the need for address arithmetic. We will therefore concentrate on the basic idea of the algorithm, and present some of the nice properties that result from this approach.

1 Matrix Multiplication

The multiplication of two matrices, $AB = C$, is not only one of the most important (sub-)tasks in linear algebra, but it is probably also one of the most frequently used algorithms in introductory lessons to programming. We can safely assume that most students in mathematics, computer science, or engineering, at one time or another, had to program it as an exercise. We can also assume that 99 % of the resulting algorithms are similar to algorithm 1.

Algorithm 1 multiplication of two n-by-n matrices

```
for i from 1 to n do
   for j from 1 to n do
      C[i,j] := 0;
      for k from 1 to n to
          C[i,j] := C[i,j] + A[i,k] * B[k,j];
      end do;
   end do;
end do;
```

Depending on the programming language that is used, the elements of the matrices A, B, and C, will be stored in row-major or column-major order, or even using a pointer-based scheme like in C/C++ or Java. As we already pointed out, the resulting programs will show a rather disappointing performance on most current computers due to the bad use of cache memory.

To improve cache performance, the temporal and spatial locality of the access to the linearized matrix elements has to be improved. Most linear algebra libraries, like implementations of BLAS [9], therefore use techniques like loop blocking, and loop unrolling [6,10]. A lot of fine tuning is required to reach optimal cache efficiency on a given hardware, and very often the tuning has to be repeated from scratch for a new machine. Recently, techniques have become popular that are based on a recursive block matrix multiplication [8]. They automatically achieve the desired blocking of the main loop, and the tedious fine tuning is restricted to the basic block matrix operations. Such algorithms are called *cache oblivious* [3], emphasizing that they are inherently able to exploit a present cache hierarchy, but do not need to know about the exact structure of the cache.

Several approaches have been presented that use an element ordering based on space filling curves [2,4]. More precisely, Morton ordering was used, which further improves the data locality of the applied block recursive algorithm. In other applications, like parallelization, or indexing in data bases, the excellent locality properties of space filling curves are well known. Zumbusch, for example, has shown that space filling curves are quasi-optimal for parallelizing codes for the numerical solution of partial differential equations [11]. However, as we will see in section 2, Morton ordering can only optimize the temporal locality during matrix multiplication, but not the spatial locality. In addition, neither of these approaches can completely avoid jumps in the address space.

In this paper, we will present an approach that uses an ordering of the matrix elements that is based on a Peano space filling curve. The Peano curve (see figure 1 also results from a recursive construction idea, so our approach will be similar to many block recursive multiplication schemes. However, our presented scheme totally avoids jumps in the access to all three matrices involved, and shows optimal spatial locality in that sense. After each individual multiplication operation, the next elements to be accessed will be direct neighbours of the previous ones.

In section 2, we will demonstrate the general idea of a Peano-based multiplication algorithm for 3-by-3 matrices. This will be extended to a block recursive matrix multiplication in sections 3 and 4, using a Peano-based indexing scheme. After some implementational issues, we will show in section 6 that the spatial and temporal locality of the element access pattern in this algorithm is asymptotically optimal for any block-recursive code.

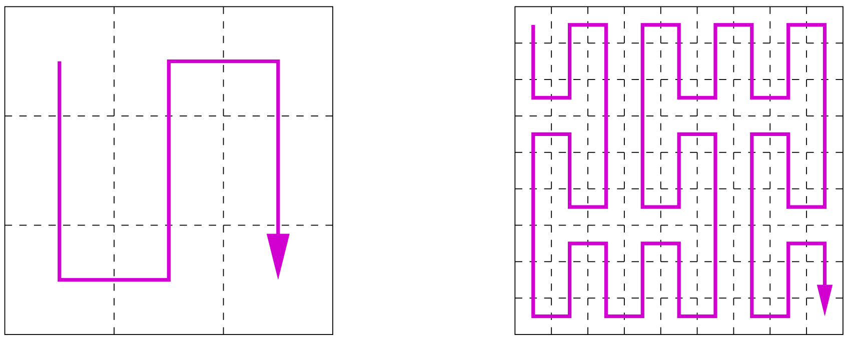

Figure 1. Recursive construction of the Peano curve: The so-called *iterations* of the Peano curve are generated in a self-similar, recursive process. The Peano curve can be imagined as the limit curve of this process.

2 Multiplication of 3-by-3 matrices

First, we take the time to re-formulate algorithm 1 into the following form:

Algorithm 2 multiplication of two n-by-n matrices (revisited)

```
// matrix C is assumed to be initialized
for all triples (i,j,k) in {1..n}x{1..n}x{1..n} do
   C[i,j] := C[i,j] + A[i,k] * B[k,j];
end do;
```

In this second algorithm, we have removed any indications on the execution order of the main loop. It may be executed in any order we find suitable, because of the commutativity. So, starting from algorithm 2, we can try to find optimal serializations of the loop, which show better locality of the element access, and can benefit from the presence of cache memory.

Let us consider the multiplication of two 3-by-3 matrices. The elements of both matrices, as well as the elements of the resulting matrix, shall be stored in a Peano-like ordering:

$$\underbrace{\begin{pmatrix} a_0 & a_5 & a_6 \\ a_1 & a_4 & a_7 \\ a_2 & a_3 & a_8 \end{pmatrix}}_{=:\,A} \underbrace{\begin{pmatrix} b_0 & b_5 & b_6 \\ b_1 & b_4 & b_7 \\ b_2 & b_3 & b_8 \end{pmatrix}}_{=:\,B} = \underbrace{\begin{pmatrix} c_0 & c_5 & c_6 \\ c_1 & c_4 & c_7 \\ c_2 & c_3 & c_8 \end{pmatrix}}_{=:\,C} \tag{1}$$

The elements c_k of the matrix C are computed as a sum of three products,

$$c_k = \sum_{(i,j)\in\mathcal{C}_k} a_i b_j, \tag{2}$$

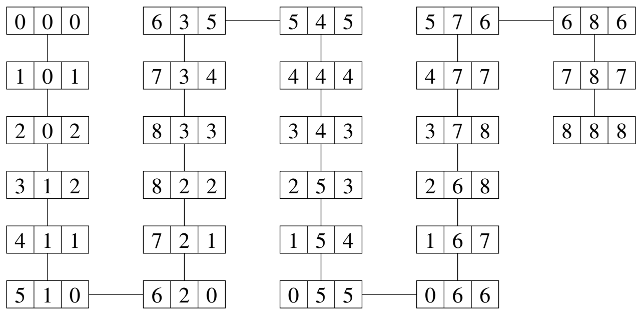

Figure 2. Graph representation of the operations of a 3-by-3 matrix multiplication.

where each set $\mathcal{C}_k$ contains the three required index pairs. The sets $\mathcal{C}_k$ are easily obtained from the matrix multiplication algorithm. For example, we get

$$C_0 = \{(a_0, b_0), (a_5, b_1), (a_6, b_2)\}\,, \tag{3}$$

or

$$C_4 = \{(a_1, b_5), (a_4, b_4), (a_7, b_3)\}\,. \tag{4}$$

Following algorithm 2 to compute the matrix-matrix product, we have to perform the following two steps:

(1) initialize all $c_k := 0$ for $k = 0, \ldots, 8$.
(2) for all triples (k, i, j) where $(i, j) \in \mathcal{C}_k$, and $k = 0, \ldots, 8$ execute:

$$c_k \leftarrow c_k + a_i b_j$$

In step 2, the individual operations can be executed in arbitrary order. Our goal will be to find an optimally "localized" execution order of the operations, which means we try to avoid jumps in the indices k, i, and j.

To find suitable serializations, we can use a graph representation. The nodes of the graph are given by the triples (k, i, j) of the matrix multiplication. Two nodes of the graph will be connected by an edge, if there is no large index jump in neither of the three indices. A suitable serialization is then given by a path through the graph that visits each node exactly once.

In the graph given in figure 2, two nodes are connected, if the difference between two indices is not larger than one in any of the components. The graph directly provides us with an optimal serialization of the matrix multiplication. We can see that after each element operation, we either directly re-use a matrix element, or we move to its direct neighbour. There are, in fact, two such serializations, as we can traverse the graph forward or backward, starting from the triples $(0, 0, 0)$ or $(8, 8, 8)$, respectively. As there are no jumps at all in the access of the matrix elements, we get, both, an optimal spatial locality,

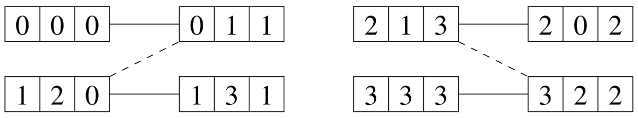

Figure 3. Graph representation of the operations of a 2-by-2 matrix multiplication using Morton ordering of the elements.

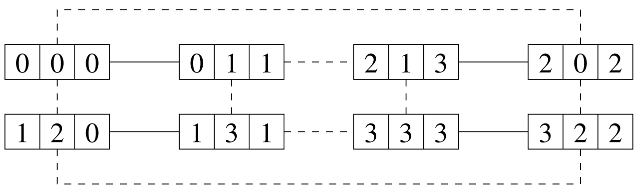

Figure 4. Graph representation of a 2-by-2 matrix multiplication using Morton ordering. Nodes are connected, if at least one element may be reused. Only the solid edges ensure spatial locality, as well.

and a very good temporal locality in the access pattern of the matrix elements. Thus, the two key requirements for good cache performance are satisfied.

It is worth to point out that a similar scheme cannot be found for a recursion based on 2-by-2 matrices. A 2-by-2 scheme similar to that in equation 1, but using Morton numbering, would look like

$$\begin{pmatrix} a_0 & a_1 \\ a_2 & a_3 \end{pmatrix} \begin{pmatrix} b_0 & b_1 \\ b_2 & b_3 \end{pmatrix} = \begin{pmatrix} c_0 & c_1 \\ c_2 & c_3 \end{pmatrix}. \tag{5}$$

The respective operation graph is given in figure 3. We can see immediately that there is no path through that graph that visits all nodes exactly once. Moreover, the dashed edges do not allow a reuse of any element. In the graph given in figure 4, we allow edges between nodes where at least one matrix block can be reused. This much weaker requirement leads to quasi-optimal temporal locality of the element access, but cannot ensure spatial locality like the 3-by-3 scheme. A serialization that ensures both, temporal and spatial locality, can not be found for the 2-by-2 case. This also holds, if other orderings are allowed (Hilbert, for example).

3 Peano indexing of larger matrices

The multiplication scheme presented in section 2 can be easily extended to the multiplication of 5×5 or 7×7 matrices. In fact, it can be used for any matrix multiplication, as long as the matrix dimensions are odd numbers. However, to improve the temporal locality of the data access, it is necessary to use a block recursive approach. Hence, our approach will be based on a blockwise

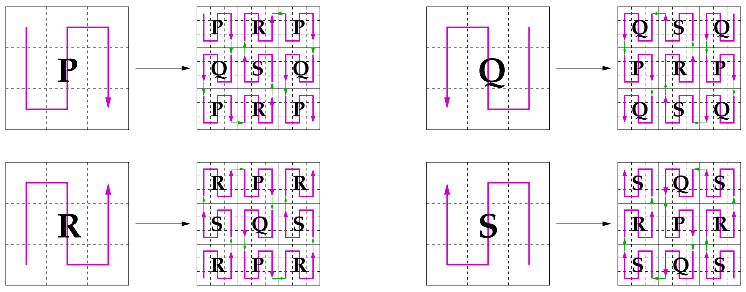

Figure 5. Recursive block numbering scheme based on the Peano curve

matrix multiplication, where the matrices are recursively divided into 3-by-3 matrix blocks. Therefore, the indexing scheme for larger matrices has to fulfill the following basic requirements:

- The range of indices within a matrix block should be contiguous. Once an enumeration of the matrix elements enters a matrix block, it has to enumerate all elements before moving to the next block.
- The indexing scheme should be somehow recursive or self-similar. such that we can re-use our multiplication scheme from section 2

These requirements are perfectly met by a suitable Peano curve. Each 3-by-3 matrix, as well as each 3-by-3 block matrix, will be numbered according to one the four schemes given in figure 5. For block matrices, the nine subblocks are, again, numbered by one of the four schemes. Figure 5 also illustrates what numbering schemes are chosen for the nine subblocks, respectively. We get a recursive numbering scheme that leads to a contiguous numbering of all matrix elements. The numbering exactly follows a so-called *iteration* of the Peano curve.

In the following, we will only discuss the case where the matrix size is a power of 3. However, for both the block recursion and the size of the smallest blocks, 5×5 or 7×7 schemes may be used, as well. In fact, any $n_x \times n_y$ scheme is applicable, where n_x and n_y are odd numbers. Therefore, the presented scheme can be modified to work with any matrices of odd dimension.

4 Recursive Peano multiplication

The Peano numbering of larger matrices is based on subdividing the matrix recursively into 3-by-3 blocks. Consequently, we will use a blockwise matrix multiplication to implement the multiplication of larger matrices. Equation 6 is an example for such a blockwise multiplication. The matrix blocks are named according to their numbering scheme, and indexed with the name of the global matrix, and their Peano index within the matrix blocks.

$$\underbrace{\begin{pmatrix} P_{A0} & R_{A5} & P_{A6} \\ Q_{A1} & S_{A4} & Q_{A7} \\ P_{A2} & R_{A3} & P_{A8} \end{pmatrix}}_{=:\,A} \underbrace{\begin{pmatrix} P_{B0} & R_{B5} & P_{B6} \\ Q_{B1} & S_{B4} & Q_{B7} \\ P_{B2} & R_{B3} & P_{B8} \end{pmatrix}}_{=:\,B} = \underbrace{\begin{pmatrix} P_{C0} & R_{C5} & P_{C6} \\ Q_{C1} & S_{C4} & Q_{C7} \\ P_{C2} & R_{C3} & P_{C8} \end{pmatrix}}_{=:\,C} \tag{6}$$

We get the following operations on the matrix blocks:

$$\begin{aligned} P_{C0} &:= P_{A0}P_{B0} + R_{A5}Q_{B1} + P_{A6}P_{B2} \\ Q_{C1} &:= Q_{A1}P_{B0} + S_{A4}Q_{B1} + Q_{A7}P_{B2} \\ R_{C5} &:= P_{A0}R_{B5} + R_{A5}S_{B4} + P_{A6}R_{B3} \\ S_{C4} &:= Q_{A1}R_{B5} + S_{A4}S_{B4} + Q_{A7}R_{B3} \end{aligned} \tag{7}$$

plus five similar equations for P_{C2}, R_{C3}, P_{C6}, Q_{C7}, and P_{C8}. The sums will be computed by an algorithmic scheme like

$$\begin{aligned} &P_{C0} := 0 \\ &P_{C0} \overset{+}{\leftarrow} P_{A0}P_{B0} \qquad \text{(short notation for } P_{C0} := P_{C0} + P_{A0}P_{B0} \\ &P_{C0} \overset{+}{\leftarrow} R_{A5}Q_{B1} \\ &P_{C0} \overset{+}{\leftarrow} P_{A6}P_{B2} \end{aligned} \tag{8}$$

If we just consider the ordering of the matrix blocks, we can see that there are exactly eight different types of block multiplications:

$$\begin{aligned} &P \overset{+}{\leftarrow} PP \qquad && P \overset{+}{\leftarrow} RQ \\ &Q \overset{+}{\leftarrow} QP && Q \overset{+}{\leftarrow} SQ \\ &R \overset{+}{\leftarrow} PR && R \overset{+}{\leftarrow} RS \\ &S \overset{+}{\leftarrow} QR && S \overset{+}{\leftarrow} SS. \end{aligned} \tag{9}$$

Similar to this $P \overset{+}{\leftarrow} PP$ operation, we now have to examination the other seven types of block multiplications. A close examination reveals that no additional operation type will arise. Thus, we have a closed system of eight multiplication schemes.

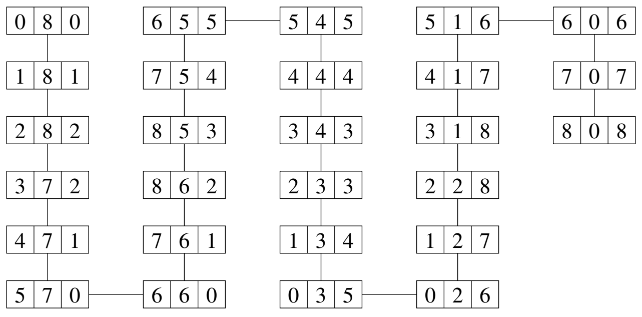

Figure 6. Graph representation of the operations of a 3-by-3 matrix multiplication of type $Q \overset{+}{\leftarrow} QP$.

The ordering of the matrix blocks in the $P \overset{+}{\leftarrow} PP$ block multiplication corresponds to that for 3-by-3 matrices. Hence, we may carry over the serialization introduced in section 2. However, we still have to find serializations for the seven other types of multiplications. We will demonstrate this for the block operation $Q \overset{+}{\leftarrow} QP$. The respective 3-by-3 matrix multiplication is

$$\begin{pmatrix} a_6 & a_5 & a_0 \\ a_7 & a_4 & a_1 \\ a_8 & a_3 & a_2 \end{pmatrix} \begin{pmatrix} b_0 & b_5 & b_6 \\ b_1 & b_4 & b_7 \\ b_2 & b_3 & b_8 \end{pmatrix} = \begin{pmatrix} c_6 & c_5 & c_0 \\ c_7 & c_4 & c_1 \\ c_8 & c_3 & c_2 \end{pmatrix} \tag{10}$$

Figure 6 shows the respective serialization graph. Again, we can instantly see the two possible serializations—one forward, one backward. The scheme is also very similar to that issued by figure 2. We recognize that we just have to invert the access order for the second index (which corresponds to the elements b_i).

Now, it is interesting to note that a $Q \overset{+}{\leftarrow} QP$ scheme will often follow a $P \overset{+}{\leftarrow} PP$ scheme. While the $P \overset{+}{\leftarrow} PP$ multiplication ends by processing the $(8, 8, 8)$ triple, the following $Q \overset{+}{\leftarrow} QP$ multiplication starts with the $(0, 8, 0)$ triple. We realize that

(1) The central P-block is directly re-used. Hence, the element b_8 will be the last accessed element of the $P \overset{+}{\leftarrow} PP$, and directly used again as the first element of the $Q \overset{+}{\leftarrow} QP$ scheme.
(2) In the global order, element 0 of the two Q-blocks will follow directly after the respective elements 8 of the two P-blocks.

Consequently, there is no index jump in neither of the three indices, if a $Q \overset{+}{\leftarrow} QP$ scheme starting at $(0, 8, 0)$ follows after a $P \overset{+}{\leftarrow} PP$ scheme ending at $(8, 8, 8)$.

We now need to repeat the analysis carried out on the schemes $P \overset{+}{\leftarrow} PP$ and $Q \overset{+}{\leftarrow} QP$ for the remaining six schemes, and their combinations. We get the

block scheme	serialization		
$C \overset{+}{\leftarrow} AB$	C	A	B
$P \overset{+}{\leftarrow} PP$	$+$	$+$	$+$
$P \overset{+}{\leftarrow} RQ$	$-$	$+$	$+$
$Q \overset{+}{\leftarrow} QP$	$+$	$+$	$-$
$Q \overset{+}{\leftarrow} SQ$	$-$	$+$	$-$
$R \overset{+}{\leftarrow} PR$	$+$	$-$	$+$
$R \overset{+}{\leftarrow} RS$	$-$	$-$	$+$
$S \overset{+}{\leftarrow} QR$	$+$	$-$	$-$
$S \overset{+}{\leftarrow} SS$	$-$	$-$	$-$

Table 1
Serializations for the eight different block multiplication schemes. A + indicates that the access pattern for the respective matrix A, B, or C is executed in forward direction (from element 0 to 8). A “minus” ($-$) indicates backward direction (starting with element 8).

following results:

(1) Every scheme leads to graph representation similar to those in figures 2 and 6. Thus, there are two optimal serializations for each scheme.
(2) All of the serializations have the same structure as that for $P \overset{+}{\leftarrow} PP$. Just like in the serialization for $Q \overset{+}{\leftarrow} QP$, we have to invert the access pattern for one, two, or even all of the three indices.
(3) For each scheme, only one of the two possible serializations will be used. In addition, this will ensure that even at the connection of two schemes, there will never be an index jump.

Table 1 shows the eight different schemes, and the access pattern of the elements for all three indices.

5 Implementation

Algorithm 3 is an implementation of the recursive scheme we have developed in the previous sections. The algorithm takes three parameters—`phsA`, `phsB`, and `phsC`—to indicate the serialization scheme (see table 1). An additional fourth parameter, `dim`, specifies the size of the current matrix block.

The actual matrices—`A`, `B`, and `C`—, as well as the matrix indices—`a`, `b`, and `c`—, are defined as global variables. In a programming language like C or

C++, the index variables `a`, `b`, and `c` are dispensable. Instead, three pointers `A`, `B`, and `C` may be used that directly reference the matrix elements. The index shifts can then be executed directly on the pointers. This will also improve the performance of the algorithm considerably.

In the given algorithm, the recursion actually goes down to 1-by1 matrices. This is rather inefficient. The recursion should be stopped at least on the previous level (`dim==3`), such that the multiplication is performed on 3-by3 matrices. However, the respective algorithm would not have fit onto a single page.

The parameters `phsA`, `phsB`, and `phsC` can only assume the values `+1` or `-1`. It is therefore possible to replace the recursive function `peanomult` by a set of eight recursive function, one for each possible combination of values for `phsA`, `phsB`, and `phsC`. The index variables `a`, `b`, and `c` can then be updated by using increase or decrease operations only. This makes it much easier for compilers to optimize the generated code, and therefore leads to a massive performance gain.

6 Characterizing Data Locality and Cache Efficiency

6.1 Data Locality

To characterize the data locality of the algorithm, we will analyse the ratio between the number of algebraic operations performed and the index range that is covered by the elements accessed by these operations. For example, during the block multiplication of two 3-by-3 matrix blocks, the algorithm will perform 27 operations (counting a multiplication and the following addition as one operation). During these 27 operations, only a subset of 9 elements will be accessed in each of the three matrices involved. If we disregard approaches like Strassen's algorithm, then in fact any matrix multiplication will have to access at least n^2 elements for performing n^3 operations—otherwise, it would perform superfluous operations. Hence, after p operations, we will cover an index range of at least $p^{2/3}$ elements. This is also the optimal ratio we can achieve in the long range.

However, a naive implementation like given in algorithm 1 will access a range of n different elements during the first n operations. Even a block recursive approach will only make sure that k^2 different elements will be accessed during k^3 operations (presuming that k is a multiple of the block size). However, these elements will not necessarily belong to a contiguous range of indices. So, while the ratio $k^2/k^3 = k^{2/3}$ is obviously the best we can get, we can characterize

Algorithm 3 Recursive implementation of the Peano matrix multiplication

```
/* global variables:
 * A, B, C: the matrices, C will hold the result of AB
 * a, b, c: indices of the matrix element of A, B, and C
 */
peanomult(int phsA, int phsB, int phsC, int dim)
{
   if (dim == 1) {
      C[c] += A[a] * B[b];
   }
   else
   {
      peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA, -phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; b += phsB;

      peanomult( phsA,  phsB, -phsC, dim/3); a += phsA; c -= phsC;
      peanomult( phsA, -phsB, -phsC, dim/3); a += phsA; c -= phsC;
      peanomult( phsA,  phsB, -phsC, dim/3); a += phsA; b += phsB;

      peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA, -phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA,  phsB,  phsC, dim/3); b += phsB; c += phsC;

      peanomult( phsA,  phsB,  phsC, dim/3); a -= phsA; c += phsC;
      peanomult( phsA, -phsB,  phsC, dim/3); a -= phsA; c += phsC;
      peanomult( phsA,  phsB,  phsC, dim/3); a -= phsA; b += phsB;

      peanomult( phsA,  phsB, -phsC, dim/3); a -= phsA; c -= phsC;
      peanomult( phsA, -phsB, -phsC, dim/3); a -= phsA; c -= phsC;
      peanomult( phsA,  phsB, -phsC, dim/3); a -= phsA; b += phsB;

      peanomult( phsA,  phsB,  phsC, dim/3); a -= phsA; c += phsC;
      peanomult( phsA, -phsB,  phsC, dim/3); a -= phsA; c += phsC;
      peanomult( phsA,  phsB,  phsC, dim/3); b += phsB; c += phsC;

      peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA, -phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; b += phsB;

      peanomult( phsA,  phsB, -phsC, dim/3); a += phsA; c -= phsC;
      peanomult( phsA, -phsB, -phsC, dim/3); a += phsA; c -= phsC;
      peanomult( phsA,  phsB, -phsC, dim/3); a += phsA; b += phsB;

      peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA, -phsB,  phsC, dim/3); a += phsA; c += phsC;
      peanomult( phsA,  phsB,  phsC, dim/3);
   };
}
```

the data locality of an algorithm by the respective worst case.

For a given algorithm, we will define the *access locality function* $L_M(n)$ as the maximal possible distance between two elements of a matrix M that are accessed within n contiguous operations.

In the Peano multiplication in algorithm 3, the access patterns of the matrices ensure that index ranges are always contiguous. Thus, after p operations, we will automatically get $L(n) \in \mathcal{O}(p^{2/3})$ as an upper bound of the extent of the index range. Moreover, we can even determine the respective factor. First, we determine the longest streak of not reusing matrix blocks in algorithm 3. On matrix A, no matrix block is reused for up to nine consecutive block multiplication. For matrix C a matrix block is reused after at most three contiguous block operations, and for matrix B, it is one block multiplication at most. During recursion, two such streaks can happen right after each other. Thus, the longest streak of not reusing a matrix block is 18 operations for matrix A, 6 operations for matrix C, and 2 operations for matrix B. Consequently, after $18n^3$ block operations on matrix A, we will access approximately $18n^2$ contiguous elements of A. Thus, for matrix A, we get that

$$L_A(n) \approx \frac{18}{18^{2/3}} n^{2/3} = \sqrt[3]{18} n^{2/3}, \tag{11}$$

For matrices B and C, we receive in a similar manner that

$$L_B(n) \approx \sqrt[3]{2} n^{2/3} \qquad L_C(n) \approx \sqrt[3]{6} n^{2/3}. \tag{12}$$

With $L_A(n) \leq 3n^{2/3}$, and both, $L_B(n) \leq 2n^{2/3}$, and $L_C(n) \leq 2n^{2/3}$, the data locality functions are all very close to the theoretical optimum, $n^{2/3}$.

6.2 Cache Misses on an Ideal Cache

To characterize the temporal locality of the algorithm, we give an estimate of the number of the generated cache misses on a so-called *ideal cache* [5]. The ideal cache model assumes a computer consisting of a local cache of limited size, and unlimited external memory. The cache consists of M words that are organized as cache lines of L words each. The replacement strategy is assumed to be ideal in the sense that the cache can foresee the future. Hence, if a cache line has to be removed from the cache, it will always be the one that is used farthest away in the future.

In the following, we will compute the number of cache line transfers required to compute a matrix multiplication of two $N \times N$ matrices, N being a power of three. The recursive algorithm leads to a recursion for the number $T(N)$ of transfers:

$$T(N) = 27T\left(\frac{N}{3}\right) = 3^3 T\left(\frac{N}{3}\right). \tag{13}$$

Now, let n be the largest power of 3, such that three $n \times n$ matrices fit into the cache. Hence, $3n^2 < M$, but $3 \cdot (3n)^2 > M$, or

$$\frac{1}{3}\sqrt{\frac{M}{3}} < n < \sqrt{\frac{M}{3}}. \tag{14}$$

Let k be the number of levels of recursion, then

$$T(N) = 3^3 T\left(\frac{N}{3}\right) = \ldots = 3^{3k} T\left(\frac{N}{3^k}\right) = \left(\frac{N}{n}\right)^3 T(n). \tag{15}$$

As long as the $n \times n$ blocks are processed, each line of memory that is accessed will be transfered to the cache at most once. Due to the ideal cache replacement strategy, it will not be deleted till we move on to the next set of $n \times n$ blocks. Hence, there will be $\left\lceil \frac{n^2}{L} \right\rceil$ cache transfers per $n \times n$ block.

As a direct result of the structure of our algorithm, one $n \times n$ block will remain in the cache as it will be reused in the next block multiplication. Hence, only two blocks will have to be transfered. A regular block recursive algorithm would often have to exchange all three blocks in the cache. For the number of cache line transfers $T(n)$, we get

$$T(n) = 2 \cdot \left\lceil \frac{n^2}{L} \right\rceil, \tag{16}$$

and therefore

$$\begin{aligned} T(N) &= \left(\frac{N}{n}\right)^3 \cdot 2 \cdot \left\lceil \frac{n^2}{L} \right\rceil \leq \left(\frac{N}{\frac{1}{3}\sqrt{\frac{M}{3}}}\right)^3 \cdot 2 \cdot \left(\frac{n^2}{L} + 1\right) \\ &\in \mathcal{O}\left(\frac{N^3}{L\sqrt{M}}\right). \end{aligned}$$

A more careful examination leads to the following approximation:

$$T(N) \approx 6\sqrt{3}\frac{N^3}{L\sqrt{M}} \tag{17}$$

7 Conclusions

We have presented a block recursive algorithm for matrix multiplication that has excellent spatial and temporal locality features. Using the ideal cache model, we were able to show that the number of cache misses is of order $\mathcal{O}\left(\frac{N^3}{L\sqrt{M}}\right)$. This is asymptotically optimal for any algorithm that is based on recursive block multiplication (algorithms that use a Strassen-like approach excluded). Moreover, the index range covered by any p consecutive operations consists of at most $c \cdot p^{2/3}$ elements, where $c < 3$ is a small constant for each of the three matrices. This is very close to the theoretical minimum of $1 \cdot p^{2/3}$.

The spatial locality is also optimal in the sense that index jumps will be totally avoided; changes in the memory addresses of matrix elements are incrementations or decrementations of at most one, which totally eliminates the need for address arithmetics. While this fact cannot be fully exploited on standard computers, it may be a considerable advantage for hardware implementations of matrix multiplication.

As we already pointed out, the algorithm can be generalized to the multiplication of non-square matrices of arbitrary size. If the numbers of rows and columns of the matrices are odd numbers (adding single rows or columns of zeroes where necessary), the 3-by-3 block recursion can, for example, be repeated up to matrix blocks of size $p \times q$, where $p, q \in \{3, 5, 7\}$. Of course, the peano numbering of the matrices has to be changed accordingly. It should also be possible to generalize the multiplication scheme to certain types of sparse matrices. Related to this is a recent work on the implementation of iterative schemes for the finite element method, where the sparse matrices result from a 9-point discretization stencil. It was shown that even with adaptivity and multi-level schemes used, it is possible to use only stacks as data structures, and therefore retain optimal spatial locality of the memory access [7].

References

[1] K. Samelson, F. L. Bauer. Sequentielle Formelbersetzung, *Elektronische Rechenanlagen 1(4)*, 1959

[2] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, Mithuna Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems, in *International Conference on Supercomputing (ICS'99)*, 1999

[3] Erik D. Demaine, Cache-Oblivious Algorithms and Data Structures, in *Lecture Notes in Computer Science, BRICS, University of Aarhus, Denmark, June 27-July 1, 2002*, Springer, to appear.

[4] Jeremy Frens, David S. Wise. Auto-Blocking Matrix-Multiplication or Tracking BLAS3 Performance from Source Code. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1997.

[5] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, October 1999.

[6] Kazushige Goto, Robert van de Geijn. On Reducing TLB Misses in Matrix Multiplication. TOMS, under revision
(preprint on `http://www.cs.utexas.edu/users/flame/pubs.html`)

[7] F. Günther, M. Mehl, M. Pögl, C. Zenger. A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves. *SIAM Journal of Scientific Computing*, submitted

[8] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development 41 (6)*, 1999

[9] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage, *ACM Trans. Math. Soft., 5*, 1979, pp. 308–323.

[10] R. Clint Whaley, Antoine Petitet, Jack J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing 27(1–2)*, 2001, pp. 3–35

[11] Gerhard Zumbusch. Adaptive Parallel Multilevel Methods for Partial Differential Equations. Habilitation, Universitt Bonn, 2001.