Available online at www.sciencedirect.com
SCIENCE DIRECT®
Linear Algebra and its Applications 417 (2006) 301–313
LINEAR ALGEBRA AND ITS APPLICATIONS
www.elsevier.com/locate/laa

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

Michael Bader *, Christoph Zenger

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

Received 7 February 2006; accepted 6 March 2006
Available online 2 May 2006
Submitted by M. Griebel

Abstract

One of the keys to tap the full performance potential of current hardware is the optimal utilization 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.

Keywords: Cache oblivious algorithms; Matrix multiplication; Space filling curves

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. Hence, even if an index is incremented only by 1, as it is done during the commonly used loop operations, the respective address in memory may jump to a far away location. Jumps in the address space, however, should be avoided on modern computer architectures, because the access to a distant element might cause a cache miss and thus take much more time than the access to a neighboring element. For matrices, a row-wise or column-wise storage scheme is most often used. Then, even

* Corresponding author. Tel.: +49 89 289 18634.
E-mail address: bader@in.tum.de (M. Bader).

0024-3795/$ - see front matter
doi:10.1016/j.laa.2006.03.018

simple algorithms such as 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 to 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 [2]. 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. We rather want to demonstrate that it is possible to construct an algorithm where memory addresses change only with step size one, which also eliminates the need for address arithmetic. We will therefore concentrate on the basic idea of this 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 courses 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 $n \times n$-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 as 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 have to be improved (see Fig. 1). Most linear algebra libraries, such as implementations of BLAS [10], therefore use loop blocking, loop unrolling, and similar techniques [7,11]. A lot of fine tuning is required to obtain optimal cache efficiency on a given hardware, and very

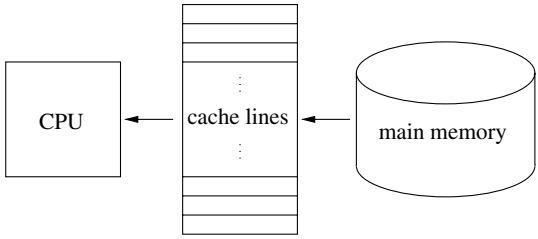

Fig. 1. A simple model of a cache. Data will be transferred from main memory into cache by one cache line at a time. Data in the same cache line will be accessed fast, even if a neighboring element caused the line to be loaded into the cache (benefit of *spatial locality*). Frequent reuse of data in the cache line will prevent the cache line to be replaced by other data (benefit of *temporal locality*). On modern hardware, a hierarchy of several caches is typically used.

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 [9]. 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* [4], 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 [3,5]. More precisely, Morton ordering (corresponding to Lebesgue's curve) was used, which further improves the data locality of the applied block recursive algorithm. In other applications 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 [12]. In data base technology, indexing based on space filling curves is also common. For matrix multiplication, however, we will show in Section 2 that Morton ordering can only optimize the temporal locality, but not the spatial locality. Therefore, 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. Peano curves (see Fig. 2) also result from a recursive construction idea, so our approach will optimize temporal locality in the same way as many block recursive multiplication schemes do. In addition, however, our scheme totally avoids jumps in the access to all three matrices involved. In that sense, its spatial locality is optimal, too. After each

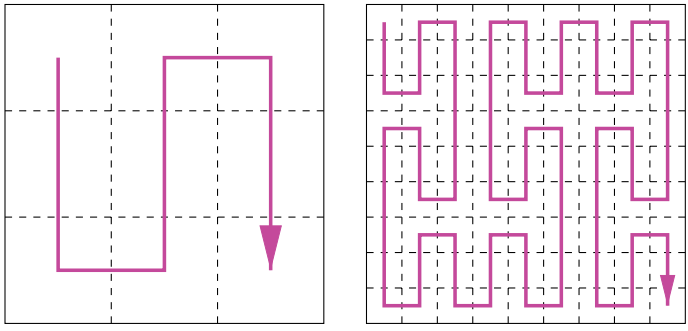

Fig. 2. Recursive construction of the Peano curve used in this article: The so-called *iterations* of the Peano curve are generated in a self-similar, recursive process. The Peano curve can be imagined as the limit curve of this process.

individual multiplication operation, the next elements to be accessed will be direct neighbors of the previous ones.

In Section 2, we will demonstrate the general idea of a Peano-based multiplication algorithm for 3×3-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.

2. Multiplication of 3 × 3-matrices

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

Algorithm 2. Multiplication of two $n \times n$-matrices (revisited)

```
// matrix C is assumed to be initialized
for all triples (i,j,k) in {1..n} × {1..n} × {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. Now, 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 3×3-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_r of the matrix C are computed as a sum of three products,

$$c_r = \sum_{(p,q)\in\mathscr{C}_r} a_p b_q, \tag{2}$$

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

$$\mathscr{C}_0 = \{(0,0),(5,1),(6,2)\}, \text{ or} \tag{3}$$

$$\mathscr{C}_4 = \{(1,5),(4,4),(7,3)\}. \tag{4}$$

Following Algorithm 2 to compute the matrix–matrix product, we have to perform the following two steps:

(1) Initialize all $c_r := 0$ for $k = 0, \ldots, 8$.
(2) For all triples (p, q, r) where $(p, q) \in \mathscr{C}_r$, and $r = 0, \ldots, 8$, execute:

$$c_r \leftarrow c_r + a_p b_q.$$

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 that we try to avoid jumps in the indices p, q, and r.

To find suitable serializations, we can use a graph representation. The nodes of the graph are given by the triples (p, q, r) 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 Fig. 3, 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 reuse a matrix element, or we move to its direct neighbor. 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 to the matrix elements, we get both optimal spatial locality and 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 2×2-matrices. A 2×2-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 Fig. 4. 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 Fig. 5, 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 as the 3×3 scheme does. A serialization that ensures both temporal and spatial locality cannot be found for the 2×2 case. This also holds if other orderings are allowed (Hilbert, for example).

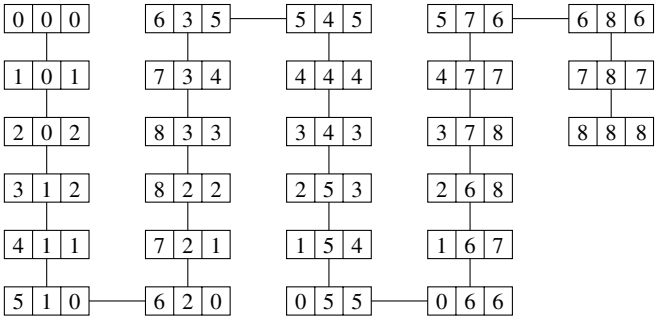

Fig. 3. Graph representation of the operations of a 3×3 matrix multiplication.

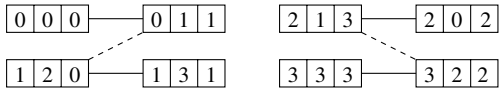

Fig. 4. Graph representation of the operations of a 2×2 matrix multiplication using Morton ordering of the elements. The dashed connections preserve locality, but do not allow reuse of matrix elements. A serialization that corresponds to Fig. 3 is not possible.

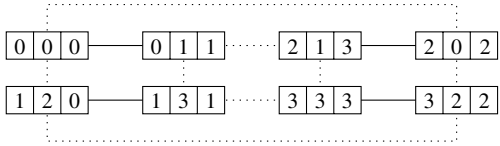

Fig. 5. Graph representation of a 2×2 matrix multiplication using Morton ordering. Here, nodes are connected, if at least one element may be reused (temporal locality). However, spatial locality is preserved only along the solid edges. A resulting serialization would achieve temporal locality only.

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 matrix multiplication, where the matrices are recursively divided into 3×3 block matrices, each block matrix being of odd dimensions. 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 reuse our multiplication scheme from Section 2.

These requirements are perfectly met by a standard Peano curve. Each 3×3-matrix, as well as each 3×3 block matrix, will be numbered according to one of the four schemes given in Fig. 6. For block matrices, the nine subblocks are, again, numbered by one of the four schemes. Fig. 6 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.

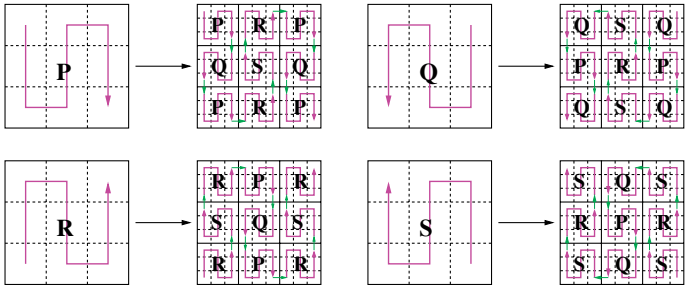

Fig. 6. Recursive block numbering scheme based on a Peano curve.

4. Recursive Peano multiplication

The Peano numbering of larger matrices is based on subdividing the matrix recursively into 3×3-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 algorithmic schemes similar to the following one:

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

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

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

Similar to this $P \overset{+}{\leftarrow} PP$ operation, we now have to examine 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.

The ordering of the matrix blocks in the $P \overset{+}{\leftarrow} PP$ block multiplication corresponds to that for 3×3-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 3×3 matrix multiplication is

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

Fig. 7 shows the respective serialization graph. Again, we can instantly see the two possible serializations—one forward, one backward. In fact, the scheme is identical to the $P \overset{+}{\leftarrow} PP$ scheme

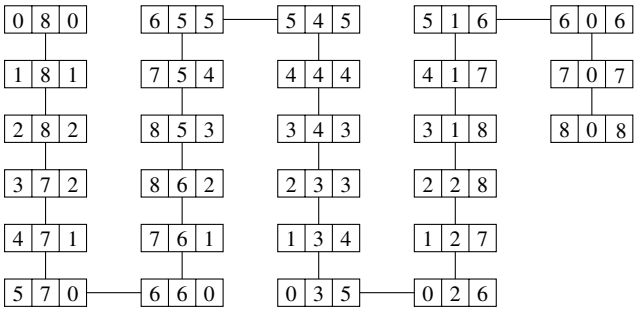

Fig. 7. Graph representation of the operations of a 3×3 matrix multiplication of type $Q \overset{+}{\leftarrow} QP$.

given in Fig. 3 up to the access order for the second index (which corresponds to the elements b_q). The second index is serialized in inverse order, which means that for this index we run the scheme backwards.

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 reused—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 following results:

(1) Each scheme leads to a graph representation similar to those in Figs. 3 and 7. Thus, there are two optimal serializations for each scheme.
(2) All of the serializations follow exactly the structure as that for $P \overset{+}{\leftarrow} PP$. Just as in the serialization for $Q \overset{+}{\leftarrow} QP$, we only have run the access pattern backwards 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 seriali-

Table 1
Serializations for the eight different block multiplication schemes

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

A "plus" (+) 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).

zation 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 such as 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.

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);
  };
}
```

In the given algorithm, the recursion actually goes down to 1×1-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 3×3-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 functions, 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 (up to a factor of 2 depending on hardware).

6. Characterizing data locality and cache efficiency

6.1. Spatial locality

To characterize the data locality of the algorithm, we will first examine the spatial locality properties of the algorithm. For that purpose, we analyze 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 3×3 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. In general, 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, which makes this number the optimal ratio we can achieve in the long range.

However, a naive implementation as 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 the block size). However, these elements will not belong to a contiguous range of indices, unless a block recursive numbering scheme is used. And even then the index range will no longer be contiguous, once the k^3 operations do not exactly match a block multiplication.

Therefore, as the ratio $k^2/k^3 = k^{2/3}$ is obviously the best we can get, we can characterize the spatial locality of an algorithm by the respective worst case. For a given algorithm, we will thus define the *access locality function* $L_M(p)$ as the maximal possible distance between two elements of a matrix M that are accessed within p 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 $L(p) \approx c \cdot p^{2/3}$ as an estimate of the extent of the index range. Moreover, we can even determine the respective factor c. 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 multiplications. For matrix C, a matrix block is reused after at most six contiguous block operations, and for matrix B, it is two block multiplications at most. For matrix A, two such streaks can occur right after each other during recursion. Thus, the longest streak of not reusing a matrix block of A is 18 operations. In such a streak, we will access $18k^2$ contiguous blocks of A while performing $18k^3$ block operations. Thus, for the access to matrix A, we get that

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

For matrices B and C, the longest streaks were 6 operations for matrix C, and 2 operations for matrix B (in that case, combining two streaks of successive block operations will not lead to a longer streak than 6, or 2 respectively). Thus, we obtain in a similar manner that

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

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

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* [6]. The ideal cache model assumes a computer consisting of a local cache of limited size, and unlimited external memory (see also Fig. 1). 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

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

As long as the $n \times n$ blocks are processed, each line of memory that is accessed will be transferred 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 $\lceil \frac{n^2}{L} \rceil$ 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 transferred. 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

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

A more careful examination leads to the following approximation:

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

For comparison: the cache-oblivious block recursive approach presented in [6] leads to $\mathcal{O}(N + N^2/L + N^3/L\sqrt{M})$ cache misses. The additional term $N + N^2$ results from copy operations, where matrix blocks are copied to and from auxiliary memory blocks used to compute the block matrix products. The respective operations, and therefore the resulting cache misses, do not occur in our algorithm. Even more important is the fact that the strictly local access pattern makes it possible to give a rather sharp estimate for the involved constant factor.

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 increments or decrements of at most one, which totally eliminates the need for address arithmetic. 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 [1]. If the numbers of rows and columns of the matrices are odd numbers (adding a single row or column of zeroes where necessary), the 3×3 block recursion

can, for example, be repeated up to matrix blocks of size $p \times q$, where $p, q \in \{3, 5, 7\}$. In [1], we also demonstrate that the approach can be competitive compared to other fast implementations of matrix multiplication. Further results also indicate that larger blocks $p \times q$ might be advantageous, because cache efficiency is no longer the dominating effect once the smallest blocks fit into the first level cache. The required optimization of this block multiplication (usually for specific hardware) is beyond the scope of this paper, though.

As future work, we are planning 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 [8].

References

[1] M. Bader, Ch. Zenger, A cache oblivious algorithm for matrix multiplication based on Peano's space filling curve, in: Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics, PPAM, in press.
[2] K. Samelson, F.L. Bauer, Sequentielle Formelübersetzung, Elektronische Rechenanlagen 1 (4) (1959).
[3] 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.
[4] 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, in press.
[5] 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.
[6] Matteo Frigo, Charles E. Leiserson, Harald Prokop, Sridhar Ramachandran, Cache-oblivious algorithms, in: Proceedings of the 40th Annual Symposium on Foundations of Computer Science, New York, October 1999, pp. 285–297.
[7] 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>).
[8] F. Günther, M. Mehl, M. Pögl, C. Zenger, A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves. SIAM J. Scient. Comput., submitted for publication.
[9] F.G. Gustavson, Recursion leads to automatic variable blocking for dense linear-algebra algorithms, IBM J. Res. Develop. 41 (6) (1999).
[10] C.L. Lawson, R.J. Hanson, D. Kincaid, F.T. Krogh, Basic linear algebra subprograms for FORTRAN usage, ACM Trans. Math. Soft. 5 (1979) 308–323.
[11] R. Clint Whaley, Antoine Petitet, Jack J. Dongarra, Automated empirical optimization of software and the ATLAS project, Parallel Comput. 27 (1–2) (2001) 3–35.
[12] Gerhard Zumbusch, Adaptive Parallel multilevel methods for partial differential equations, Habilitation, Universität Bonn, 2001.