Scientific Computing Mandatory Assignment 1

Magnus Erik Hvass Pedersen
University of Aarhus, Student #971055
Danish Technical University, Student #053375
September 2005

1 Introduction

The purpose of this document is to verify attendance of the author to the *Scientific Computing* course, at the Danish Technical University (DTU).

A study of cache usage and its impact on computational performance of matrix multiplication is presented, where experiments are conducted on various combinations of algorithms for performing such multiplication, and data-structures for storing the matrices. Although the basic theory is briefly recapped, the reader is still assumed to be familiar with the project description, basic linear algebra, and various aspects of computer architecture, memory layout, and the programming languages C and C++ in particular.

After a brief introduction to what matrix multiplication is all about, we move straight to the development and implementation, which is carried out in phases. First, an object-oriented framework is implemented, allowing for easy experimentation with combinations of different data-structures for storing the matrices, and different traversal algorithms for computing the matrix product. This framework however, induces a significant overhead in computational time, so a specialized implementation is made, utilizing the knowledge obtained from experimenting with the object-oriented framework. Finally, this more efficient algorithm will be used in a study of block-decomposition of matrix multiplication, in an attempt to improve cache performance even further.

2 Matrix Multiplication

Assume we are given an $(n \times p)$ matrix A, meaning it has n rows and p columns, and assume we are given another $(p \times m)$ matrix B. Then denote by C, the matrix of size $(n \times m)$ resulting from taking the product of matrices A and B. Recall the definition of matrix multiplication:

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} \tag{1}$$

where we denote by x_{ij} the element at the i'th row and j'th column of some matrix X. Note that the entry c_{ij} in the matrix product of A and B, is therefore just the dot-product of the vector corresponding to the i'th row of matrix A, and the vector corresponding to the j'th column of matrix B. Keeping this in mind, will help explain why the direct computation of the matrix product, can be very inefficient.

In the following, no assumptions will be made on whether these matrices are sparse or dense, diagonal in some way, or generally exhibit any structure that may be exploited in the multiplication. That is, we know nothing about the matrices A and B, except for their sizes.

3 Implementation

Implementation is carried out in *MS Visual C++ .NET* and compiles to an executable for *MS Windows.* The source-code relies heavily on template classes for easy specialization and change of numeric datatypes (e.g. floating points, integers, etc.). Socalled *assertions* are used throughout for easy debugging, and do not compile into the final *release-build.* Assertions also makes it easier to get an overview of valid parameters, states, etc., thus helping maintenance of the source-code.

3.1 Direct Computation

An obvious kind of storage for a matrix, is a double-array, that is, an array of arrays. So for example for the `float`-datatype, we would declare something like:

```
float** A, B, C;
```

for matrices A, B, and C,[1] and the direct way of computing the matrix product from Eq.(1), is then a socalled nested loop:

```
for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
        for (int k=0; k<p; k++)
            C[i][j] += A[i][k] * B[k][j];
```

where it is assumed that all entries in the matrix C, have initially been set to zero. The time-complexity is $O(n \cdot m \cdot p)$.

3.2 Alternative Computation

There are many alternative ways of traversing the matrices A, B, and C. The most obvious ones result from simply interchanging the order of iterating the variables `i`, `j`, and `k`. This gives $3 \cdot 2 \cdot 1 = 6$ possible ways of traversing the matrices. But there are many more. For example, we may also traverse the indices in a backwards manner, meaning there are a total of $6 \cdot 4 \cdot 2 = 48$ possible combinations for computing the resulting matrix C. Any other ordering will also do, as long as the resulting matrix C essentially equals Eq.(1). but we will not try all of these, but rather focus on the six ways that arise from simply interchanging the traversal order of indices `i`, `j`, and `k`.

[1]Note that no actual storage has been allocated, these are just declarations of variables.

3.3 Efficiency Factors

Neither of these alternative ways of computing the matrix product, have lower asymptotic time complexities than the direct method from section 3.1, and moreover, the number of operations are more or less equal, regardless of how the matrices are traversed. So how can it be, that we wish to investigate and compare the performances of the different traversals, when they have the same number of operations?

The reason is, that computers are built in ways that favour some kinds of memory usage over others. Also, certain instruction-set patterns are favourable, because they improve the numeric processor's ability to perform pipelining, prefetching, etc. All of which are modern tricks to improve performance, but are completely transparent to the application programmer – unless an explicit performance comparison is made, or the algorithms are implemented directly in socalled assembly language, which corresponds in a one-to-one manner, to the machine's own instruction set.

In this report, our two main concerns are with memory organization, that is, how we structure the memory storing the matrices. And secondly, the traversal of these data-structures, as discussed above.

3.3.1 Memory Organization & Cache

On the machine-level, memory can be thought of as a large string of consecutive bits, of which we can allocate substrings of various lengths, provided they are available. In the programming languages C and C++, this layout is replicated, and arrays of some datatype can be allocated, that may be indexed as mathematical arrays, with the index starting at zero. Arbitrary lookup of elements in an array, must take constant time, that is, the time complexity must be $O(1)$, so the elements of an array, can generally be assumed to be stored consecutively in the physical memory also.

Unfortunately, memory bandwidth is much less than what the numeric processor can consume, and several layers of socalled cache memory, has therefore been introduced between the processor and the physical memory. The quicker cache memory, the more expensive it is, and hence the smaller it usually is. The fastest kind of memory is of course the socalled registers onboard the processor itself, of which there are only relatively few. Anyway, cache memory also consists of consecutive strings or arrays of bits, so the actual memory is swapped in and out of cache, in entire arrays of some given size.

There are many aspects of code optimization for fully utilizing the memory caches, but here we will focus on just one of these, namely whether the matrices can be held in the cache or not. If a matrix can not be kept in cache due to its size and the particular pattern of memory access, then it is important that we address it in an order that maximizes the number of socalled cache hits, meaning the number of times data which we are trying to fetch, is actually held by the cache. The worst case scenario, is that the cache never holds any of the data that we need, which is known as thrashing.

3.4 Numerical Precision

Depending on the application in which the matrix multiplication is to be used, another important issue might be numerical precision of the floating point operations. Naturally, digital and finite computers are only capable of storing approximations to real-valued numbers with a finite precision, meaning that eventually, rounding errors will occur when operating on arbitrary numbers. One may typically choose between socalled single-point precision floating point numbers, or double-precision. Some machine architectures even have extended precision floating point numbers. But the problem of rounding errors is merely postponed, and with large enough matrices, certain applications could still be sensitive to rounding errors.

The problem can be illustrated by studying section 3.1, in which two arithmetic operators are in play, namely multiplication and (inplace) addition. When using floating point numbers, both of these operators may yield imprecise output, requiring some form of rounding to be employed. For each entry in c_{ij}, that algorithm merely calculates the dot-product of the i'th row of A with the j'th column of B, as noted above. However, a rounding error in the first entry, may accumulate throughout the calculation, thus causing rounding errors on top of rounding errors, which may eventually become too substantial – again, depending on the requirements of the application.

A simple solution for lowering the rounding error in this kind of numerical computation, is to organize the calculations in a manner similar to a binary tree, which can then be implemented recursively. In this way, each rounding error may only propagate through a logarithmic number of operations, instead of a linear number of operations.

However, some arithmetic processing units have extended precision floating point registers, so keeping the summation variable in such a register, may be beneficial in some cases. At any rate, using binary organization of the computation, can be expected to impact performance, both in terms of the number of operations, the additional memory needed, and general memory access affecting cache usage. It is expected that all of this will mean a performance degradation, although it is possible that it may result in performance improvement instead, however unlikely it may sound. Bottom line is, that one should be wary of concluding anything about performance and precision without having actual experiments to back it up.

3.5 Object-Oriented Design

The abstract base-class for matrix storage is `LMatrix`, which is specialized into `LMatrixDoubleArray` and `LMatrixSingleArray`. The former uses a double-array for storage, as described in section 3.1, whereas the latter uses a single array, and provides appropriate mapping of indices, so the matrix element lookup remains transparent to the user of those classes. All these are socalled template-classes, making it easy to change the datatype of the matrix elements.

The `LMatrix` class also supplies functions for returning abstract references to rows and columns of a matrix. These abstract references are simply instances of the classes `LMatrixRow` and `LMatrixColumn`, each holding a reference to the matrix in question, and an index to the given row or column. Both classes override the `operator[]` function, thus allowing their user to address rows and columns as if they were vectors. This also makes it possible to address a given matrix element a_{ij} as `A[i][j]`, even though `A` is actually an object and not a double-array. Note that having these auxiliary classes representing abstract references to rows and columns, is actually necessary in C++ if we want to address elements as `A[i][j]`, because the operator `[][]` does not exist, and can therefore not be overloaded, as `operator[]` could.

3.6 Multiplication Algorithms

The matrix multiplication algorithms are implemented in a straight-forward manner, and to provide homogeneous timing results, no shortcuts are taken, though sometimes possible. For example, one could use a temporary summation variable instead of accumulating the elements directly in the destination matrix, which of course means even more memory access.

An example of one of the multiplication functions is the following, which first iterates over index i, then k, and finally j. The function is split into two, the first being:

```
template <typename T, typename U, typename V>
void Multiply_ikj(T& C, U& A, V& B, int n, int m, int p)
{
    assert(n>=0 && m>=0 && p>=0);

    InitializeZero(C, n, m);

    for (int i=0; i<n; i++)
        for (int k=0; k<p; k++)
            for (int j=0; j<m; j++)
                C[i][j] += A[i][k] * B[k][j];
}
```

which works for direct double-array as well as `LMatrix` implementations, but has no checking as to whether the supplied parameters are correct, with the exceptions that the integers must be non-negative (which could have been insured by declaring their data-types as unsigned). In this regard, `LMatrix` is supported and checked more smoothly by a wrapper function:

```
template <typename T, typename U, typename V>
void Multiply_ikj(LMatrix<T>& C, LMatrix<U>& A, LMatrix<V>& B)
{
    // Dimensionalities.
    const int n = C.GetN();
    const int m = C.GetM();
    const int p = A.GetM();

    // Ensure dimensionalities of supplied matrices are correct.
    assert(n == A.GetN() && m == B.GetM() && p == B.GetN());

    Multiply_ikj(C, A, B, n, m, p);
}
```

Note that each matrix parameter has its own template-type, which means that we can combine matrices with different storage and element data-types. Also note that the type-deduction of C++ will automatically take the function with the best matching parameter types.

The auxiliary function `InitializeZero()` simply intializes all the elements of a matrix to zero, and is implemented very similarly to the multiplication function:

```
template <class T>
void InitializeZero(T& A, unsigned int n, unsigned int m)
{
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            A[i][j] = 0;
}
```

Which may also be wrapped for the `LMatrix` class:

```
template <class T>
void InitializeZero(LMatrix<T>& A)
{
    // Dimensionalities.
    const int n = A.GetN();
    const int m = A.GetM();

    InitializeZero(A, n, m);
}
```

Generally note that values are always passed by reference. This is important not only for the outcome matrix C, but also for matrices A and B, so they are not duplicated on each function call, which would be highly inefficient. The parameters for matrices A and B should have been declared `const`, but this is not possible because a dereferencing such as `A[i][j]` returns a non-const reference to the respective element of the matrix.

One of the problems with this object-oriented approach, is the overhead arising from the virtual functions in the `LMatrix` sub-classes. Socalled iterators could be used to solve this inefficiency problem, as well as the redundancy problem where each of the six considered traversal approaches are implemented very similarly, with only the order of iteration differing. But using iterators seems to be overkill for this project. Besides, the C++ compiler used, does not properly optimize the code for template-functions, which means that we have to hard-code a specialized function anyway, when we find out which memory layout and traversal order is the most efficient.

3.6.1 Matrix Multiplication Using Binary Dot-Product

One advantage of the object-oriented approach, is that it provides us with a very simple way of ordering the computation of the dot-products in a binary manner, so as to lower the floating point rounding errors. The template function for computing the binary dot-product is recursively implemented as follows:

```
template <typename T, class T1, class T2>
T BinaryDotProduct (T1& x, T2& y,
                    unsigned int n,
                    unsigned int i = 0)
{
    if (n == 1)
    {
        return x[i] * y[i];
    }
    else if (n >= 2)
    {
        // Split array into two sub-arrays of size m1 and m2,
        // and call recursively.

        const unsigned int m1 = n >> 1; // m1 = n/2
        const unsigned int m2 = n-m1;

        return BinaryDotProduct<T>(x, y, m1, i)
             + BinaryDotProduct<T>(x, y, m2, m1);
    }
    else // n == 0
    {
        return 0;
    }
}
```

And this is then used in the following function for performing matrix multiplication:

```
template <typename T>
void Multiply_BinDotProduct(LMatrix<T>& C,
                            LMatrix<T>& A,
                            LMatrix<T>& B)
{
    // Dimensionalities.
    const int n = C.GetN();
    const int m = C.GetM();
    const int p = A.GetM();

    // Ensure dimensionalities of supplied matrices are correct.
    assert(n == A.GetN() && m == B.GetM() && p == B.GetN());

    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            C[i][j] = ArrayOps::BinaryDotProduct<T>(A.Row(i),
                                                     B.Column(j), p);
}
```

3.7 Testing

To test the correctness of the implementation, we manually check the outcome of multiplying the following matrix:

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}$$

with:

$$B = \begin{bmatrix} 6 & 7 & 8 \\ 9 & 10 & 11 \end{bmatrix}$$

and ascertain that the following holds:

$$C = AB = \begin{bmatrix} 9 & 10 & 11 \\ 39 & 44 & 49 \\ 69 & 78 & 87 \end{bmatrix}$$

This checking is done for each of the configurations (i.e. combination of memory layout and order of traversal) experimented with below.

3.8 Experimental Settings

As mentioned above, modern computers have several layers of cache, where the fastest are also the most expensive, and hence the smallest. In these experiments we are mainly interested in uncovering the performance difference on large matrices, arising from switching the iteration order.

3.8.1 Expectations

We expect that the iteration order which traverses the elements of the matrices in the order that they are stored, will have the best performance on large matrices, due to their cache friendliness. And vice versa. More specifically, this means that iterating over i, then k, then j, as in the `Multiply_ikj()` algorithm above, is expected to have the best performance, along with the algorithm iterating first over k, then i, and then j. The reader may check that both of these will iterate the elements of the matrices A and B in the order that they are stored. Smaller matrices may be kept more or less entirely in cache, which means the performance difference is expected to be negligible, for the different orderings of index-iteration.

3.8.2 Data-Sets

We conduct experiments on three sets of matrices, one with smaller sizes that can probably be held entirely in cache, one set of medium sized matrices that can be partially held in cache, and finally a set of large matrices, which can definitely not be contained in cache. Because multiplication of the small and medium sets are performed too quickly for us to accurately measure the duration, we perform and measure the time it takes to perform 100000 multiplications of the smaller matrices, 100 multiplications of the medium sized matrices, and just a single multiplication of the large matrices. The figures presented in the tables below are then the total time usage divided by the number of multiplications performed. Notice that not only will some or all of the matrices be in cache after their initialization, but this kind of repeated multiplication, naturally also affects the ratio between cache hits and misses. However, what we wish to find out, is the overall performance tendencies, and not the exact time usage, so these experiments will suffice.

3.8.3 Memory Requirements

Since double-precision floating points take up 8 bytes per variable, the A matrices consume $16 \cdot 32 \cdot 8 = 4096$, $64 \cdot 128 \cdot 8 = 65536$, and $2048 \cdot 1024 \cdot 8 = 16777216$ bytes for the different matrix sizes. The B matrices take up $8 \cdot 32 \cdot 8 = 2048$, $512 \cdot 128 \cdot 8 = 524288$, and $512 \cdot 1024 \cdot 8 = 4194304$ bytes respectively. The C matrices use $16 \cdot 8 \cdot 8 = 1024$, $64 \cdot 512 \cdot 8 = 262144$, and $2048 \cdot 512 \cdot 8 = 8388608$ bytes. This means, that for the largest matrices, we need approximately 28 MB of memory in total, which is far beyond the size of the memory cache, which is probably not more than a half MB, if that. Double-arrays require additional storage for pointers to the rows of the matrices also, but this is comparatively negligible. Double-arrays however, have the advantage that they do not require the very large amounts of memory to be consecutively available in memory, as the storage of each matrix is split up into its rows.

3.8.4 Matrix Initialization

Apart from the manual testing of the correctness of the algorithms, the experimentation with time usage is carried out on matrices where the elements are double-precision floating point numbers, that are all initialized randomly and uniformly within the range $(-1, 1)$. The pseudo-random number generator is Ran1 from [1, Chapter 7.1]. The random seed is not changed for the different runs, which means the contents of the matrices remain the same for all experiments of the same sizes.

3.8.5 Experimental Source-Code

The basic source-code for conducting these experiments, is as follows:

```
const int kN = 16, kM = 8, kP = 32, kNumRuns = 100000;

LMatrixDoubleArray<double> A(kN, kP), C(kN, kM);
LMatrixDoubleArray<double> B(kP, kM);

std::clock_t beginTime, endTime;

InitializeRandom(A, kN, kP);
InitializeRandom(B, kP, kM);

beginTime = std::clock();
for (int i=0; i<kNumRuns; i++)
{
    Multiply_ijk(C, A, B);
}
endTime = std::clock();

std::cout << "(n, m, p) = (" << kN << ", " << kM
          << ", " << kP << ")\n";

std::cout << "Time spent: "
          << (double) (endTime-beginTime)/kNumRuns
          << " milli-seconds per multiplication" << std::endl;
```

Where the particular kind of storage and algorithm is hard-coded, and thus needs to be manually changed into something else for each experiment, which of course also requires a recompile.

3.9 Experimental Results, Double-Arrays

The results of running these experiments on double-arrays and with the six permutations of the order in which the indices of the matrices are traversed, are presented in table 1. From this it can be clearly seen, that our expectations in regards to performance, were indeed correct, and the traversal order of i, k, and j, is the best-performing. Interestingly however, this is not true for the smallest matrices.

3.10 Experimental Results, Single-Arrays

The results of conducting the same experiments, only this time with single-dimensional arrays for storage, are shown in table 2. The tendencies are the

(n,m,p)	$(16,8,32)$	$(64,512,128)$	$(2048,512,1024)$
ijk	0.09273	100.84	37063
ikj	0.09283	93.53	23904
jki	0.09002	98.54	44363
jik	0.09063	98.94	28611
kij	0.09173	93.63	25046
kji	0.09123	102.54	45525

Table 1: Time usage in milli-seconds for multiplying matrices of the given sizes (n, m, p), and with traversal of the matrices according to the listed order of indices i, j, and k. Storage for matrices is double-arrays of double-precision floating point numbers.

same as for double-arrays, but much more polarized. Here the *ikj* and *kij* orderings are still the most efficient, yet slightly worse than for double-arrays; but the other index traversal orderings are 7-8 times worse than they were for double-arrays.

The reason why the *ikj* and *kij* single-array versions are slower than the double-array ones, is clearly because additional arithmetic operations are required to map from two-dimensional to one-dimensional coordinates. Because the lookup function is a virtual function, the compiler is unable to optimize and possibly even remove this mapping by transforming the loops, and this is believed to be the reason for that particular difference in performance.

In regards to the gross performance decrease when the elements of the matrices are used in a different order than they are stored in the single-dimensional arrays, the answer is not so clear cut. However, there is no need to partake in any guessing, as we have no further use for those algorithmic variants anyway.

(n,m,p)	$(16,8,32)$	$(64,512,128)$	$(2048,512,1024)$
ijk	0.09543	103.24	199657
ikj	0.09533	96.23	24725
jki	0.09403	111.66	361770
jik	0.09403	104.04	202491
kij	0.09764	98.64	25636
kji	0.09733	114.46	366707

Table 2: Time usage in milli-seconds for multiplying matrices of the given sizes (n, m, p), and with traversal of the matrices according to the listed order of indices i, j, and k. Storage for matrices is single-dimensional arrays of double-precision floating point numbers.

3.11 Experimental Results, Binary Dot-Product

Recall the matrix multiplication algorithm from section 3.6.1 which used a so-called binary dot-product, that is, the summation in the dot product, was organized as a binary tree, with the objective being to improve the numerical precision. When using matrices in which the elements are double precision floating point numbers, this kind of dot product is only relevant for very very large matrices. In fact, the compiler being used here, has a poorer numerical precision for binary summation (dot-product), when implemented as a template function, than for computing it in a straight-forward manner.

The results of running the same experiments as above, but with binary dot-product, are shown in table 3. The figures designated as *normal*, are for the algorithm as given in section 3.6.1, whereas the ones designated *reverse*, use a storage abstraction for the matrix, implemented in the class `LMatrixReverse`, which stores the given matrix in its transposed form, but accesses it with reversed indices also. This means that the algorithm from section 3.6.1 may still be used, but now it uses the elements from matrix B in the same order as they're stored in memory. In either case, double-arrays are used for storage.

Storage	$(16, 8, 32)$	$(64, 512, 128)$	$(2048, 512, 1024)$
Normal	0.15472	159.22	41179
Reverse	0.16193	168.14	43873

Table 3: Time usage in milli-seconds for multiplying matrices of the given sizes (n, m, p), and using binarily ordered computation of the vector dot-products. Storage for matrices is two-dimensional arrays of double-precision floating point numbers, in either normal or reverse order (see section 3.11 for explanation).

As can be seen, these figures are significantly worse than the time usage for regular computation of the vector dot-products. Interestingly, the reverse storage order has even worse performance, even though it should have improved cache usage. The reason for this is probably the extra layer of virtual functions in `LMatrixReverse`, which perhaps requires even more computational time than what is gained by the more proper cache usage.

3.12 Non-Template Implementation

As mentioned previously, the compiler in question, does not properly optimize template classes and functions, and furthermore, the abstract storage of matrix elements through the `LMatrix` class, implies extra overhead for its virtual function calls. For these reasons, the matrix multiplication algorithm with *ikj* traversal is implemented as a non-template function using direct storage of double-arrays, as follows:

```
void Multiply_ikj(double** C, double** A, double** B,
                  int n, int m, int p)
{
    assert(C && A && B && n>=0 && m>=0 && p>=0);

    InitializeZero(C, n, m);

    for (int i=0; i<n; i++)
```

```
        for (int k=0; k<p; k++)
            for (int j=0; j<m; j++)
                C[i][j] += A[i][k] * B[k][j];
}
```

And for single-dimensional arrays:

```
void Multiply_ikj(double* C, double* A, double* B,
                  int n, int m, int p)
{
    assert(C && A && B && n>=0 && m>=0 && p>=0);

    InitializeZero(C, n, m);

    for (int i=0; i<n; i++)
        for (int k=0; k<p; k++)
            for (int j=0; j<m; j++)
                C[i*m+j] += A[i*p+k] * B[k*m+j];
}
```

3.12.1 Experimental Results

The results of running the matrix multiplication experiments with these algorithms, are shown in table 4, and prove to be a significant improvement over tables 1 and 2. In contrast to the earlier results however, here the version for single-dimensional arrays is much faster than the version for two-dimensional arrays. And it may even be possible to optimize the one-dimensional version further, by simplifying the index-mappings, although it is believed, that the compiler does that to some extent already.

Storage	$(16, 8, 32)$	$(64, 512, 128)$	$(2048, 512, 1024)$
`double**`	0.03124	32.94	9223
`double*`	0.01311	11.61	6078

Table 4: Time usage in milli-seconds for multiplying matrices of the given sizes (n, m, p), and using non-template functions, as well as direct storage of matrices in one- and two-dimensional arrays.

3.13 Non-Template Implementation & Reverse Storage

We may avoid accumulating directly on the elements of matrix C, but still access both matrices A and B in the order in which their elements are stored in memory. The trick is to store the transpose of matrix B, that is $D = B^T$, and then instead of accessing element b_{kj}, use d_{jk} which are of course equivalent, but means that with ijk traversal ordering, we would access both elements of A and D in a row-by-row order. Note that this is identical to the idea employed in section 3.11 for computing the dot-product in a manner resembling a binary tree. It means we can have a summation variable instead of using elements of matrix C over and over again, and hence also avoid explicitly initializing the elements of C to zero. The algorithm is as follows:

```
// Assumes B is stored as its own tranpose.
void Multiply_Reverse(double* C, double* A, double* B,
                      int n, int m, int p)
{
    assert(C && A && B && n>=0 && m>=0 && p>=0);

    for (int i=0; i<n; i++)
    {
        for (int j=0; j<m; j++)
        {
            double c = 0;

            for (int k=0; k<p; k++)
            {
                c += A[i*p+k] * B[j*p+k];
            }

            C[i*m+j] = c;
        }
    }
}
```

3.13.1 Experimental Results

The experimental results are shown in table 5, which are slightly better than table 4. However, given the added complexity of using this algorithm, and with only such a negligible improvement, we refrain from using it any further in this study. Also, it makes block-decomposition more difficult, as block-decomposition relies on using C for temporary storage, as we will see shortly.

$(16, 8, 32)$	$(64, 512, 128)$	$(2048, 512, 1024)$
0.00971	10.01	5578

Table 5: Time usage in milli-seconds for multiplying matrices of the given sizes (n, m, p), and using reverse access of the transposed matrix B. Storage for matrices is one-dimensional arrays of double-precision floating point numbers.

3.14 Matrix Block-Multiplication

The implementation for single-dimensional arrays in section 3.12 has a problem with its cache usage, even though it appears to be very fast. The algorithm accesses matrix B in a row-by-row manner, which is also the way it is stored. However, if the B matrix is sufficiently large, then the cache can not hold the elements of the matrix until they are needed again. So even though the cache will get loaded with successive parts of the matrix-row in question, then these elements will not get re-used in time for them to still be in the cache.

A trick is therefore to decompose the matrices into smaller blocks which can be held in cache, and then perform multiplication on those blocks. There are two immediate and intuitive ways of seeing that this actually works. First imagine a matrix having as its elements, smaller matrices of appropriate sizes. The second way, is to consider e.g. the algorithm from section 3.12, and merely note that we may re-order this computation any way we like, as long as each element of C finally amounts to what is required by Eq.(1) on page 1.

3.14.1 Algorithm

The idea in the block-decomposition below, is to use square matrix-blocks whenever possible, and progress the blocks in the same manner as we know to be cache-wise efficient from our previous studies, namely in the index-order of i, k, and j (see Eq.(1) for the meaning of these indices). So we split the matrix multiplication algorithm into two functions, one performing the actual multiplication on blocks, with $i \in \{n_0, \cdots, n_1\}$ and the block delimiters n_0 and n_1 satisfying: $1 \leq n_0 < n_1 \leq n$, and similarly for $j \in \{m_0, \cdots, m_1\}$ and $k \in \{p_0, \cdots, p_1\}$. The function is as follows:

```
void Do_Multiply_Block_ikj(double* C, double* A, double* B,
                           int n,  int m,  int p,    // Actual matrix sizes.
                           int n0, int m0, int p0,   // Start-indices for block.
                           int n1, int m1, int p1)   // End-indices for block.
{
    assert(C && A && B && n>=0 && m>=0 && p>=0);
    assert(n0 >= 0 && n0<n1 && n1 <= n);
    assert(m0 >= 0 && m0<m1 && m1 <= m);
    assert(p0 >= 0 && p0<p1 && p1 <= p);

    for (int i=n0; i<n1; i++)
        for (int k=p0; k<p1; k++)
            for (int j=m0; j<m1; j++)
                C[i*m+j] += A[i*p+k] * B[k*m+j];
}
```

The block-delimiters are then, as mentioned, progressed in the same order of ikj, and `Do_Multiply_Block_ikj()` is called on each block, accumulating its results to the output matrix C. We also need to ensure that the blocks do not exceed the boundaries of the matrices. The second function is therefore:

```
void Multiply_Block_ikj(double* C, double* A, double* B,
                        int n, int m, int p,
                        int blockSize)
{
    assert(C && A && B && n>=0 && m>=0 && p>=0 && blockSize>0);

    InitializeZero(C, n, m);

    for (int n0=0; n0<n;)
    {
        int n1 = std::min(n0+blockSize, n);

        for (int p0=0; p0<p;)
        {
            int p1 = std::min(p0+blockSize, p);

            for (int m0=0; m0<m;)
            {
                int m1 = std::min(m0+blockSize, m);

                Do_Multiply_Block_ikj(C, A, B,
                                      n, m, p,
                                      n0, m0, p0,
                                      n1, m1, p1);

                m0 = m1;
            }

            p0 = p1;
        }

        n0 = n1;
    }
}
```

3.14.2 Testing

Testing the blocked matrix multiplication implementation above, was done in two ways. First manually as described in section 3.7 and with block-sizes 1, 2, 3, and 4. No errors were uncovered. Then a small code-fragment for testing on the larger matrices, was added after the timing fragment in section 3.8.5:

```
    std::cout << "Checking with control matrix .. ";
    double* D = new double[kN*kM];
    Multiply_ikj(D, A, B, kN, kM, kP);
    bool identical = true;

    for (int i=0; i<kN*kM && identical; i++)
```

```
        identical = (C[i] == D[i]);

    if (identical)
        std::cout << "identical.\n";
    else
        std::cout << "Mismatch!\n";
```

Where it is assumed that the algorithm from section 3.12 is indeed correct.

Note that both the blocked and non-blocked algorithms accumulate in the same order to each element in matrix C, so the resulting floating point numbers can be expected to have identical rounding errors for the two algorithms, and we may therefore use strict comparison `==`.[2] The two algorithms merely defer the partial accumulation of elements in C for different amounts of time, which of course has no influence on the rounding errors, as the ordering is still exactly the same. No programming errors were uncovered with this checking-procedure, when performing the timing experiments below.

3.14.3 Experimental Results

Table 6 presents the results of using the blocked matrix-multiplication algorithm for the usual experiments. Because caches sizes (in bytes) are most likely powers of 2, we have chosen blocksizes that are also powers of 2. From the results, we see that finding the optimal blocksize appears to be a socalled unimodal optimization problem, and one could try and use binary search, starting with the endpoints 128 and 256, to see if the best actual blocksize is not a power of 2. But as the performance difference for these two blocksizes is so small, we will not search any further.

At any rate, the results for blocksize 256, clearly show an improvement over the non-blocked algorithm in table 4 on page 13, with only a very tiny performance degradation on the smallest sized matrices. On larger matrices still, the performance difference can be expected to be even greater.

4 Conclusion

A number of variations on naive matrix multiplication were tested, as well as different data-structures for storing matrices. In the experiments conducted here, it was found that a single-dimensional C or C++ array along with the corresponding algorithm from section 3.12, was the fastest combination. This algorithm was converted in section 3.14.1 into a blocked version, so as to increase cache re-use on very large matrices, which was shown to yield maximum performance with blocksizes of (256×256) matrix elements.

[2]Some programming languages even disallow strict comparison of two floating point numbers, exactly because of rounding errors and the semantic implications. Naturally, one may cheat and negate a less-than and greater-than comparison to achieve the same logical comparison.

Blocksize	$(16, 8, 32)$	$(64, 512, 128)$	$(2048, 512, 1024)$
2	0.05007	50.37	13229
4	0.02193	22.13	5858
8	0.01412	16.12	4356
16	0.01381	13.92	3575
32	0.01361	12.61	3144
64	0.01361	11.91	2964
128	0.01361	11.51	2874
256	0.01361	11.31	**2834**
512	0.01361	11.41	3154
1024	0.01361	11.41	5928

Table 6: Time usage in milli-seconds for multiplying matrices of the given sizes (n, m, p), and with the blocked matrix multiplication algorithm from section 3.14.1. Storage for matrices is single-dimensional arrays of double-precision floating point numbers.

This study could be extended in numerous ways. For example, one could use socalled meta-programming to improve performance for matrix expressions involving several operands, and this would also give a cleaner syntax, such as `C=A*B;` for matrix-matrix multiplication.[3] The *Blitz++* library [2] facilitates this, among others.

Another example of extending this study, is the use of alternative matrix multiplication schemes, for example socalled *Strassen* multiplication [3, p. 2875]. And yet another suggestion, would be to distribute the matrix multiplication to several processors on a shared-memory machine, or even to distribute the computation to several machines through a network. This is quite possibly the objective of the remaining projects of the *Scientific Computing* course.

References

[1] William H. Press et. al., Numerical Recipes in C, Cambridge University Press 1992, ISBN 0-521-43108-5

[2] Todd Veldhuizen, Blitz++, `http://oonumerics.org/blitz/`

[3] Eric W. Weisstein, CRC Concise Encyclopedia of Mathematics 2nd ed., Chapman & Hall/CRC 2002, ISBN 1-58488-347-2

[3]Strictly speaking, the simple expression `C=A*B;` can also be implemtend with operator-overloading, but it implies the creation of a temporary matrix for storing the product, and then copying all of these matrix-elements in the assignment to `C`, which is highly inefficient.