Jerzy Stefan RESPONDEK[*]

DOUBLE POINTER SHIFTING WINDOW C++ ALGORITHM FOR THE MATRIX MULTIPLICATION

Summary

The main objective of this article is to make use of the pointer effectiveness in the matrix multiplication task. To achieve this we proposed an advanced fast pointer-oriented matrix multiplication numerical recipe. The main obstacle to this aim is iterating through a matrix column, because it requires jumping over a separate tables. As a solution to this trouble I proposed a shifting window in a form of a table of auxiliary double pointers. The ready-to-use C++ source code is presented. Finally, we performed thorough time execution tests of the new C++ matrix multiplication algorithm. That tests proved the high efficiency of the proposed algorithm.

Key words: *Numerical Recipes, C++, Numerical Algebra, Linear Algebra, Matrix Multiplication, Pointers, Smart Pointers, Iterators*

Introduction

The C++ programming language is based on the C programming language and enables to create the object-oriented software with the speed of the C programs. The C++ was designed by Bjarne Stroustrup ([3],[4]). The C programming language was designed by Kernighan, Ritchie ([2]) as a highly effective tool to operate system designing. Apart from the operating systems, the majority of the severe contemporary software is still programmed in the C++ language. The high efficiency of the software coded in C++ follows, to a large extent, from C++ pointers. The pointers appear also in other general purpose languages but in the C++ programs they are usually the sole part of the algorithms code. The pointer-oriented C++ code closes the programming style to the direct assembler programming causing its unmatched effectiveness.

[*]dr Jerzy Stefan Respondek, Bielsko-Biała School of Finance and Law; Silesian University of Technology Faculty of Automatic Control, Electronics and Computer Science.

The main objective of this article is to make use of the pointer effectiveness in one of the fundamental numerical algorithms, i.e. in the matrix multiplication. The main obstacle to this aim is iterating through a matrix column, because it requires jumping over a separate tables. As a solution to this trouble I proposed a shifting window in a form of a table of auxiliary double pointers.

Neither of the available monographs on numerical recipes makes use of the pointers, even if the algorithms they present are coded in C++. This probably follows from the fact that in the past the numerical algorithms were coded with the use of such languages as Fortran, Algol and Pascal. In those languages the pointers are used in a very limited range. Thus in this article we want to show how we can improve the execution time thanks to involving the C++ pointers in the numerical algorithms by the example of the matrix multiplication. We proposed the pointer-oriented matrix multiplication algorithm 0 that turned out to be significantly more efficient than the classic one.

The paper is organized as follows: chapter 1 provides the necessary theoretical background for the matrix multiplication problem and storing the matrices as a series of dynamic tables, in chapter 3 we present the general paradigm of the pointer programming, in chapter 4 we present the classic matrix multiplication algorithm, in chapter 5 we present the new fast C++ matrix multiplication algorithm, in chapter 6 we carry out its thorough performance tests, in chapter 7 we give some conclusions.

1. Theoretical background

1.1 Matrices from the mathematical point of view

The sole matrix definition can be found e.g. in the book Bellman [1] p.37. The $m \times n$-dimensional matrix A we will denote as $A = \left[a_{ij} \right]_{\substack{i=1,..,m \\ j=1,..,n}}$.

The objective of the article is to present a fast C++ algorithm for matrix multiplication. Thus let us first refer to the formal definition of the matrix multiplication.

1.1.1 Matrix Multiplication Definition (Bellman [1] pp.39)

Let us be given two matrices $A=\left[a_{ij}\right]_{\substack{i=1,..,m\\ j=1,..,n}}$, $B=\left[b_{ij}\right]_{\substack{i=1,..,n\\ j=1,..,l}}$. *As a result of the multiplication of the* A *matrix by the* B *matrix we define the matrix* $C=\left[c_{ij}\right]_{\substack{i=1,..,m\\ j=1,..,l}}$ *with entries expressed by* (1)*:*

$$c_{ij}=\sum_{k=1}^{n}a_{ik}b_{kj},\quad i=1,...,m,\ j=1,...,l \tag{1}$$

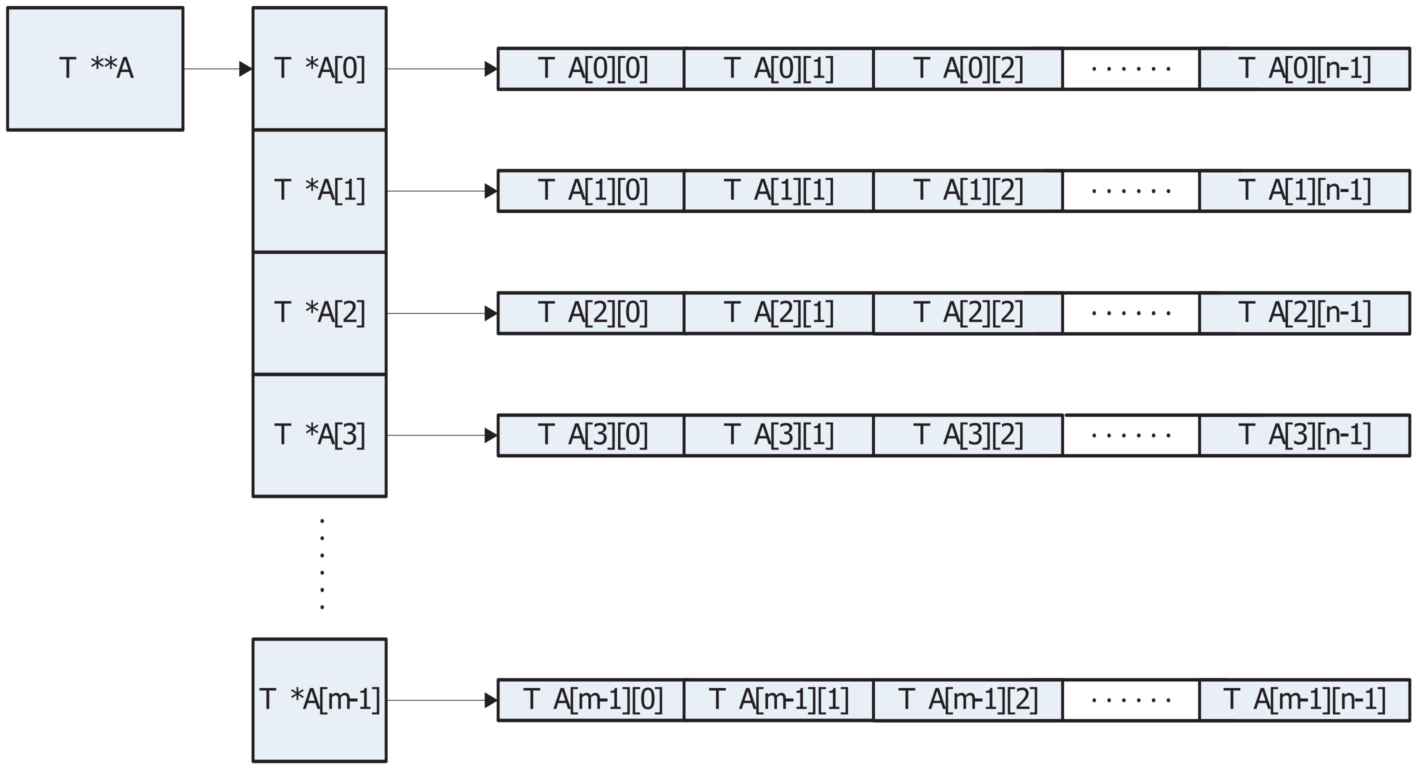
1.2 Storing the matrices in the memory as a series of dynamic tables

The most frequently used and flexible way to store the matrix in the memory is the so-called double pointer structure. This topic is presented in the most general case, for the arbitrary order of the pointers, in the [5]. Let us assume that we want to store an $m\times n$-dimensional A matrix of the general data type T in a memory. The appropriate solution is:

- declare the double pointer of the type T**,
- assign to it the address of the m-element table of the T* type pointers,
- assign to each of the above single pointers the addresses of the n-element T-type tables.

We illustrated the idea of storing the matrices as a series of dynamic tables in figure 1.

Figure 1. Representation of the matrices in the memory as a series of dynamic tables



The proper C++ type definition code has the following form:

```
      template <class T> struct TMatrix
      {
            T **data;
            int rows,cols;
      };
```

where *data* is the double pointer indicating the matrix data while *rows* and *cols* are the number of matrix rows and columns, respectively. The task of storing in a memory a $m \times n$-dimensional matrix A of the data type T can be performed by the following C++ code:

```
      TMatrix A;

      A.rows = m, A.cols = n;
      A.data = new T* [m];
      for(int i=0;i<m;i++)
            A.data[i] = new T [n];
```

Such a way of a computer matrix representation is used by all professional mathematical packages, like Matlab® and similar. It has two important advantages:

- The dimensions of the allocated matrix can be dynamically determined during the program execution. Thus there is no need to reserve an additional amount of memory during the code compilation; we use exactly as much memory as we need.
- We gain fast random access to each element of the allocated matrix. In order to get the value of the a_{ij} element of the A matrix the only code we have to write is `A[i][j]`. It is both concise and efficient. It does not need to perform any index multiplication. On the contrary, in the simple 1D table matrix representation, to get the a_{ij} element we have to code: `A[i*n+j]` which is neither convenient nor efficient.

2. The C++ pointer-based programming paradigm fundamentals

The pointers in C++ play a much greater role than in other programming languages. The fundamental C reference book [2] devotes a broad separate chapter 5 to the pointer notion. The most important advantage of the pointers is the high efficiency of the sequential table operations coded with their use. In Table 1 we showed, by the exemplary task of summing the table elements, its two opposite implementations: by the table indexing and by the pointer iteration.

Table 1. Summing the table elements by two implementation paradigms

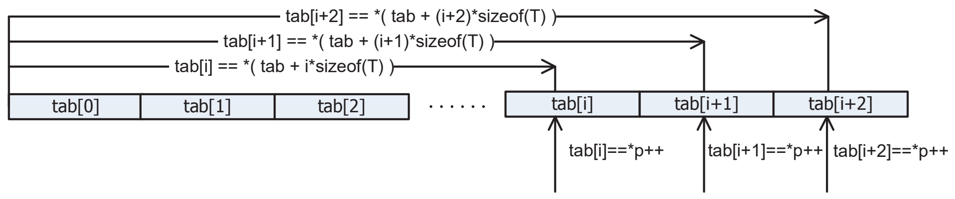
A) `float tab[1000],s=0.0;` `for(int i=0;i<100;i++)` `s += tab[i] ;`	B) `float tab[1000],*p=tab,s=0.0;` `for(int i=0;i<100;i++)` `s += *p++ ;`

From Table 1 we can conclude the following reasons of the pointer-based programming execution time boost:

- In the table-based implementation (cell A) we have to determine the memory address for each summed table element in each loop iteration again and again in compliance with the expression: &tab[i] = tab + i*sizeof(float). It is worth to notice that in the table implementation it is necessary to perform an additional integer multiplication in each loop iteration.
- In the pointer-based implementation (cell B), in order to sequentially sum up the table elements the only task we have to perform in each loop iteration (apart from the sole element addition) is to shift the pointer value to the next table element by a constant equal to sizeof(float).

We illustrated the idea of pointer-based table access compared with the classic one in Figure 2.

Figure 2. The classic table access vs. pointer based table access


tab[i+2] == *(tab + (i+2)*sizeof(T))
tab[i+1] == *(tab + (i+1)*sizeof(T))
tab[i] == *(tab + i*sizeof(T))
tab[0]
tab[1]
tab[2]
......
tab[i]
tab[i+1]
tab[i+2]
tab[i]==*p++
tab[i+1]==*p++
tab[i+2]==*p++

The main objective of this article is to apply the C++ pointer programming paradigm in order to obtain highly efficient matrix multiplication.

3. The classic matrix multiplication algorithm

The matrix multiplication definition (Bellman [1] p.39) leads directly to the following, well known, function:

```
   template <class T>
   void  Classic_Matrix_Multiply(Matrix<T>  &A,Matrix<T>
&B,Matrix<T> &C)
   {
     T temp;
     for(int i=0;i<A.rows;i++)
          for(int j=0;j<B.cols;j++)
          {
               temp=0.0;
               for(int k=0;k<A.cols;k++)
                    temp    +=    A.data[i][k]    *
B.data[k][j];
               C.data[i][j]=temp;
          }
   }
```

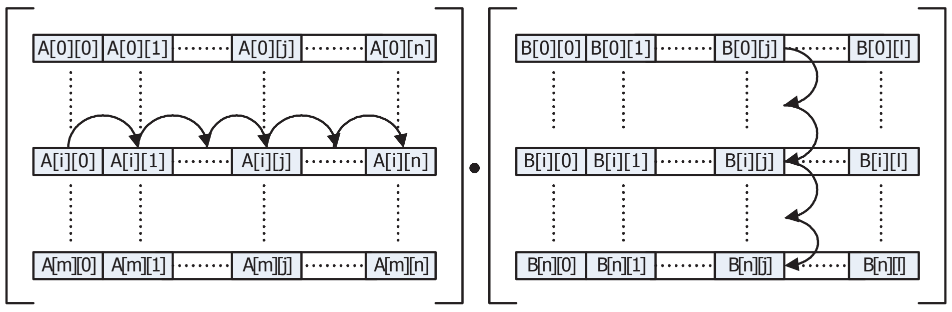
It can be noticed that at each iteration of the most nested loop we have to calculate anew the memory address of the proper elements of the multiplicated matrices, namely, the addresses of the elements `A.data[i][k]` and `B.data[k][j]`. It is highly ineffective. In the next item we present a fast C++ matrix multiplication algorithm that enables to avoid determining those two data memory addresses anew in each iteration.

4. The fast C++ matrix multiplication algorithm

4.1 Problem analysis

In item 2 we showed how to use the pointers in the sequential iterating through one dimensional table. The problem of the pointer iteration through two dimensional matrices is a more sophisticated one. We illustrated the necessary pointer paths in the matrix A by B multiplication in figure 3.

Figure 3. The pointer iteration paths in the matrix multiplication process



A[0][0] A[0][1] A[0][j] A[0][n]
A[i][0] A[i][1] A[i][j] A[i][n]
A[m][0] A[m][1] A[m][j] A[m][n]
•
B[0][0] B[0][1] B[0][j] B[0][l]
B[i][0] B[i][1] B[i][j] B[i][l]
B[n][0] B[n][1] B[n][j] B[n][l]

In figure 3 we can observe the following:

- The pointer iteration along a fixed matrix row is a relatively simple task. This task is in fact equivalent to the iteration through the ordinary dimensional table explained in item 2. The simplicity of this task arises from the fact that a single matrix row occupies a coherent block in the operating memory.
- The problem significantly complicates when it comes to the pointer iteration along a matrix column. In this task we have to iterate over a series of the row tables. Each row is usually placed in a different memory location, as we presented in item 0.

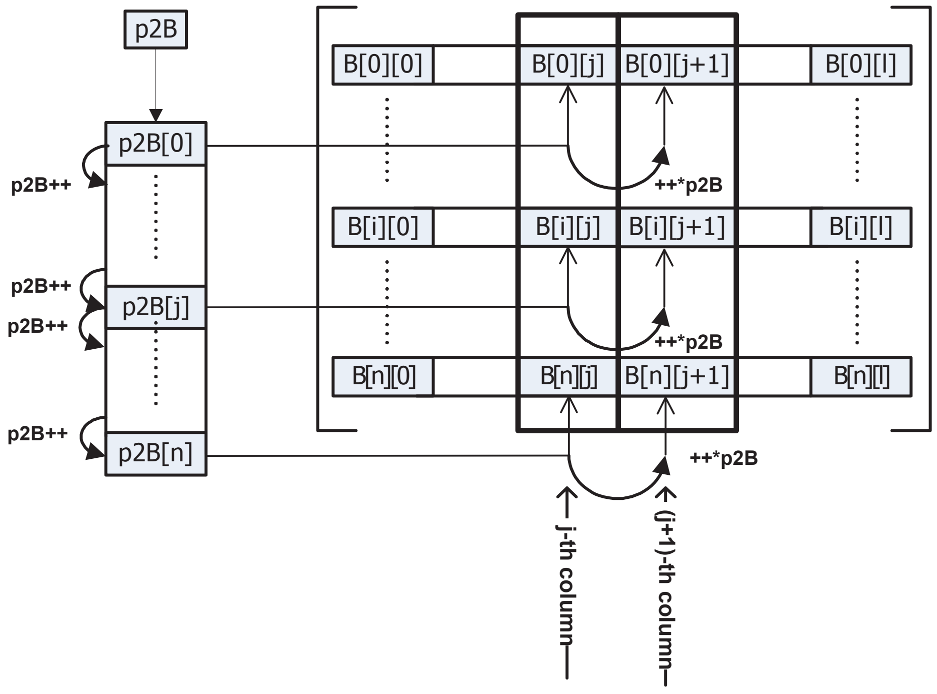
The effective solution of that problem is the crucial part of this article and we presented it in the next chapter.

4.2 The main idea of the proposed C++ matrix multiplication algorithm

To eliminate the problem of the time consuming iteration along the matrix columns we proposed the following solutions:

- Introduction of an auxiliary pointer table `p1B`. In that table we store the addresses of the current column elements, initializing them by the addresses of the first column cells of the B matrix (code line 11 in item 0). Next, we sequentially increment each pointer value in the pointer table `p1B` (26th code line). This way we get a shifting memory window, enveloping a single matrix column. This idea is clearly illustrated in figure 4.

Figure 4. Shifting the memory window



The most nested loop code (line 22) of the proposed algorithm deserves a closer look. It adds the subsequent scalar terms of the form $a_{ik}b_{kj}$, due to the definition 0. It has the following form:

temp += *p2A++ * **p2B++; (2)

The sub-expression: ***p2B++ performs 3 tasks at once:

- gets the value of the current B matrix cell (sub-expression **p2B),
- multiplies it by the value of the proper cell of the A matrix,
- finally post-increments the value of the p2B pointer, shifting it to the next matrix cell.

• Changing the order of calculations of the matrix product. The classic matrix multiplication algorithm (ch.0) calculates the elements of the $C = AB$ matrix in the natural row-by-row order. In the advanced multiplication algorithm we changed the calculation order of the $C = AB$ matrix to the column-by-column order. It enabled to minimize the number of necessary memory window shifting moves

(26th code line) thanks to moving this operation to the least nested loop.

It is worth to notice that the code (2) requires no integer multiplication to determine the addresses of the a_{ik}, b_{kj} matrix cells. Instead we use the highly C++-oriented fast code involving advanced double pointer iteration. That is just the main advantage of the proposed C++ algorithm and the reason why it is so efficient.

4.3 The final C++ matrix multiplication algorithm code

The innovations proposed in the previous item lead to the following, final C++ code for the fast matrix multiplication:

```
   1 :     T **p1B,**p1C;
   2 :
   3 :     template <class T> // C=AB
   4 :     void           Fast_Matrix_Multiply(Matrix<T>
&A,Matrix<T> &B,Matrix<T> &C)
   5 :     {
   6 :           int
rows1=A.rows,cols1=A.cols,cols2=B.cols;
   7 :           int i,j,k;
   8 :           T temp;
   9 :           T
**p1A,*p2A,**p2B=p1B,**p3B=B.data,**p2C=p1C,**p3C=C.data;
   10:
   11:           for(k=0;k<cols1;k++) *p2B++=*p3B++;
   12:           for(k=0;k<rows1;k++) *p2C++=*p3C++;
   13:
   14:           for(j=0;j<cols2;j++)
   15:           {
   16:                 p1A=A.data,p2C=Cp1;
   17:                 for(i=0;i<rows1;i++)
   18:                 {
   19:                       p2A=*p1A++,p2B=p1B;
   20:                       temp=0.0;
   21:                       for(k=0;k<cols1;k++)
   22:                             temp   +=   *p2A++   *
**p2B++;
   23:                       **p2C++ = temp;
   24:                 }
   25:                 p2B=p1B,p2C=p1C;
   26:                 for(k=0;k<cols1;k++) ++*p2B++;
   27:                 for(k=0;k<rows1;k++) ++*p2C++;
   28:           }
   29:     }
```

The meaning of the variables is as follows:

- **i, j, k** – loop working variables,
- **temp** – the variable storing the temporary value of the scalar product,
- **rows1, cols1, cols2** – number of the respective matrix rows and columns,
- **A, B** – matrices to be multiplied,
- **C** – result of the matrix multiplication, with respect to the formula $C = AB$,
- **pxA ,pxB, pxC** – auxiliary pointers.

5. The performance tests

In this chapter we present the results of the performance tests we performed to verify the robustness of the proposed fast C++ matrix multiplication algorithm 0. The test was performed on the Intel™ Core i7 workstation under the Visual Studio™ Ultimate 2010 suite with the control of the Windows 7 64bit operating system. We stored the following:

- The matrix multiplication time for both the classic matrix multiplication algorithm (item 0) and the fast one (item 0) – the results are presented in fig. 5 for the 32-bit code and in fig. 7 for the 64-bit one.
- The received execution time shortage of the fast algorithm with respect to the classic one – similarly, Figure 6 presents the received results for the 32-bit code and fig. 8 for the 64-bit one.

Then we performed the execution time tests for the square matrices of the dimensions ranging from 10×10 up to 4000×4000.

Figure 5. Execution time for the 32bit code

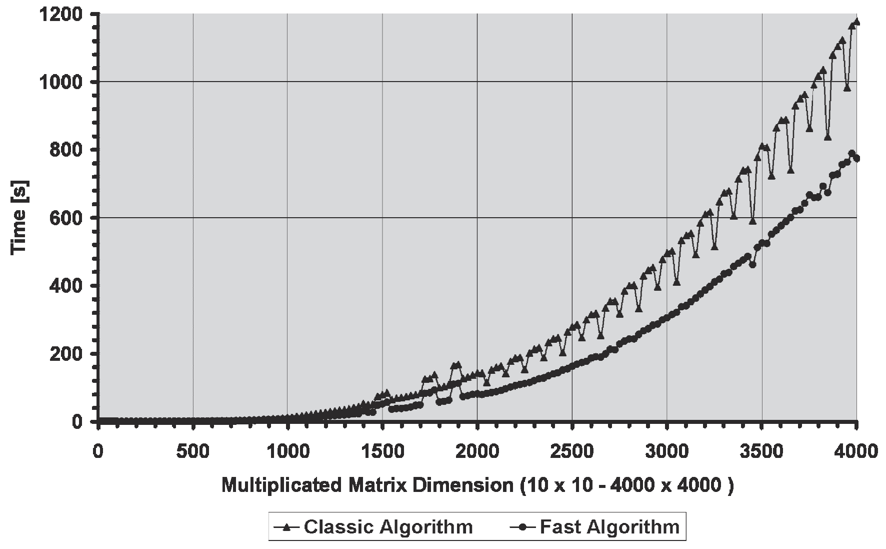


Figure 6. Execution time boost for the 32bit code

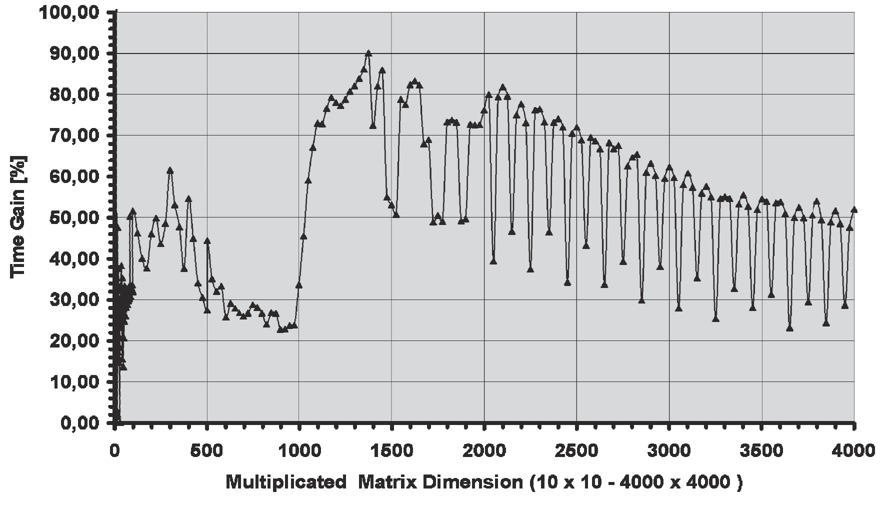


Figure 7. Execution time for the 64bit code

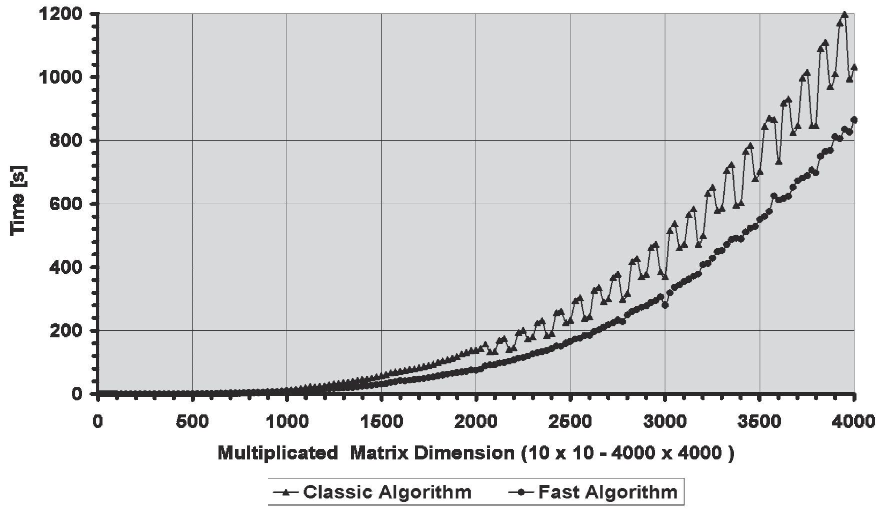
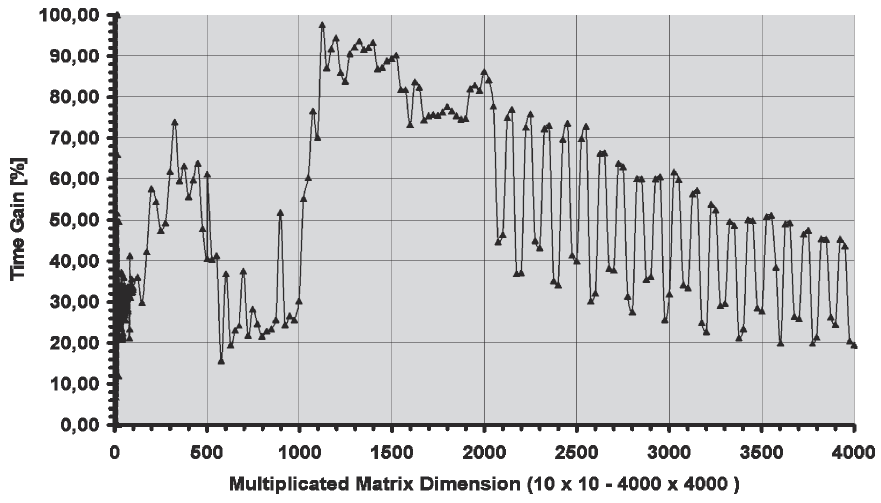


Figure 8. Execution time boost for the 64bit code



We can observe that for both small-size, medium-size and large matrices the proposed fast C++ matrix multiplication algorithm 0 allows to shorten significantly the necessary calculation time. Thus we consider the alg. 0 worth to be published.

Summary

The main novelty of this article is that we proposed the fast C++ algorithm for the matrix multiplication. The C++ programming language appeared to be a powerful tool in the matrix multiplication task and generally in the numerical recipes field.

To sum up, we hope that many researchers and engineers will find the proposed ready-to-use algorithms useful in their work.

References

[1] Bellman R., Introduction to Matrix Analysis, Society for Industrial Mathematics, 2nd ed., New York, 1987.

[2] Kernighan B. W., Ritchie D. M., The C Programming Language, 2nd edtion Prentice-Hall, New Jersey, 1988.

[3] Stroustrup B., The C++ Programming Language, 4th ed., AT&T Labs, New Jersey, 2013.

[4] Stroustrup B., The Design and Evolution of C++, Addison-Wesley, 9th ed., Massachusetts, 1994.

[5] Waite W. M., Goos G., Compiler Construction, Monographs in Computer Science, Springer Verlag, 2nd edition, New York, 1983.

IMPLEMENTACJA W C++ ALGORYTMU MNOŻENIA MACIERZY W POSTACI PRZESUWNEGO OKNA PODWÓJNYCH WSKAŹNIKÓW

Streszczenie

Celem artykułu jest zastosowanie wskaźników do celu efektywnego mnożenia macierzy. Głównym problemem jest iteracja poprzez kolumnę macierzy, która grupuje elementy nieprzylegające. Do celu rozwiązania tego problemu zaproponowano przesuwne okno podwójnych wskaźników. Przeprowadzone testy efektywności potwierdziły wysoką efektywność metody.

Słowa kluczowe: *metody numeryczne, C++, algebra numeryczna, algebra liniowa, mnożenie macierzy, wskaźniki, inteligentne wskaźniki, iteratory*