ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КАФЕДРА ВС

# ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4

«Оптимизация доступа к памяти»

по дисциплине «Архитектура вычислительных систем»

Выполнил: студент гр. ИП-811
Адов Артем Сергеевич
Проверил: ст. преп. Кафедры ВС
Ткачёва Татьяна Алексеевна

Новосибирск 2020

# Постановка задачи

1.На языке C/C++/C# реализовать функцию DGEMMBLAS последовательное умножение двух квадратных матриц с элементами типа double. Обеспечить возможность задавать размерности матриц в качестве аргумента командной строки при запуске программы. Инициализировать начальные значения матриц случайными числами.

2.Провести серию испытаний и построить график зависимости времени выполнения программы от объёма входных данных. Например, для квадратных матриц с числом строк/столбцов 1000, 2000, 3000, ... 10000.

3.Оценить предельные размеры матриц, которые можно перемножить на вашем вычислительном устройстве.

4.Реализовать дополнительную функциюDGEMM_opt_1,в которой выполняется оптимизация доступа к памяти, за счет построчного перебора элементов обеих матриц.

5.* Реализовать дополнительную функцию DGEMM_opt_2, в которой выполняется оптимизация доступа к памяти, за счет блочного перебора элементов матриц. Обеспечить возможность задавать блока, в качестве аргумента функции.

6.** Реализовать дополнительную функцию DGEMM_opt_3, в которой выполняется оптимизация доступа к памяти, за счет векторизации кода.

7.Оценить ускорение умножения для матриц фиксированного размера, например, 1000x1000, 2000x2000, 5000x5000, 10000x10000.* Для блочного умножения матриц определить размер блока, при котором достигается максимальное ускорение.

8.С помощью профилировщика для исходной программы и каждого способа оптимизации доступа к памяти оценить количество промахов при работе к КЭШ памятью (cache-misses).

9.Подготовить отчет отражающий суть, этапы и результаты проделанной работы

## Выполнение работы

Предельные размеры матриц, которые можно перемножить на моей машине, рассчитываются следующим способом:

$$\text{макс}N = \sqrt{\frac{80\%\text{отобъёма}RAM(\text{б})}{\frac{\text{количествоматрицхранимыхвпамяти}}{\text{размеродногоэлемента(б)}}}}$$

$$\text{макс}N = \sqrt{\frac{0.8 * 1000 * 1000 * 1000}{\frac{3}{8}}} = 5774 \text{ элемент}$$

# Результат работы

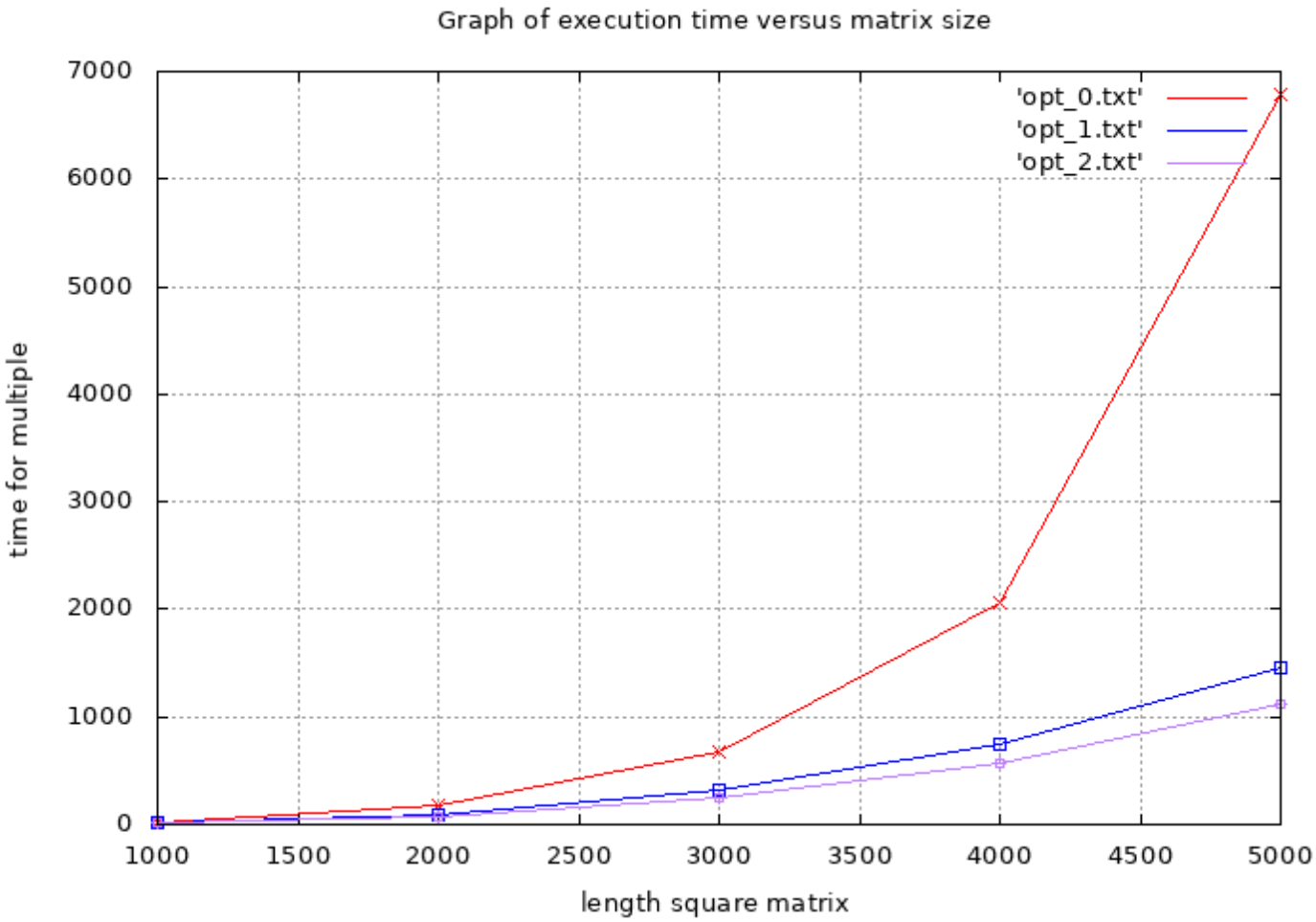



Рисунок №1. «График зависимости времени выполнения различных методов перемножения матриц от объема данных»

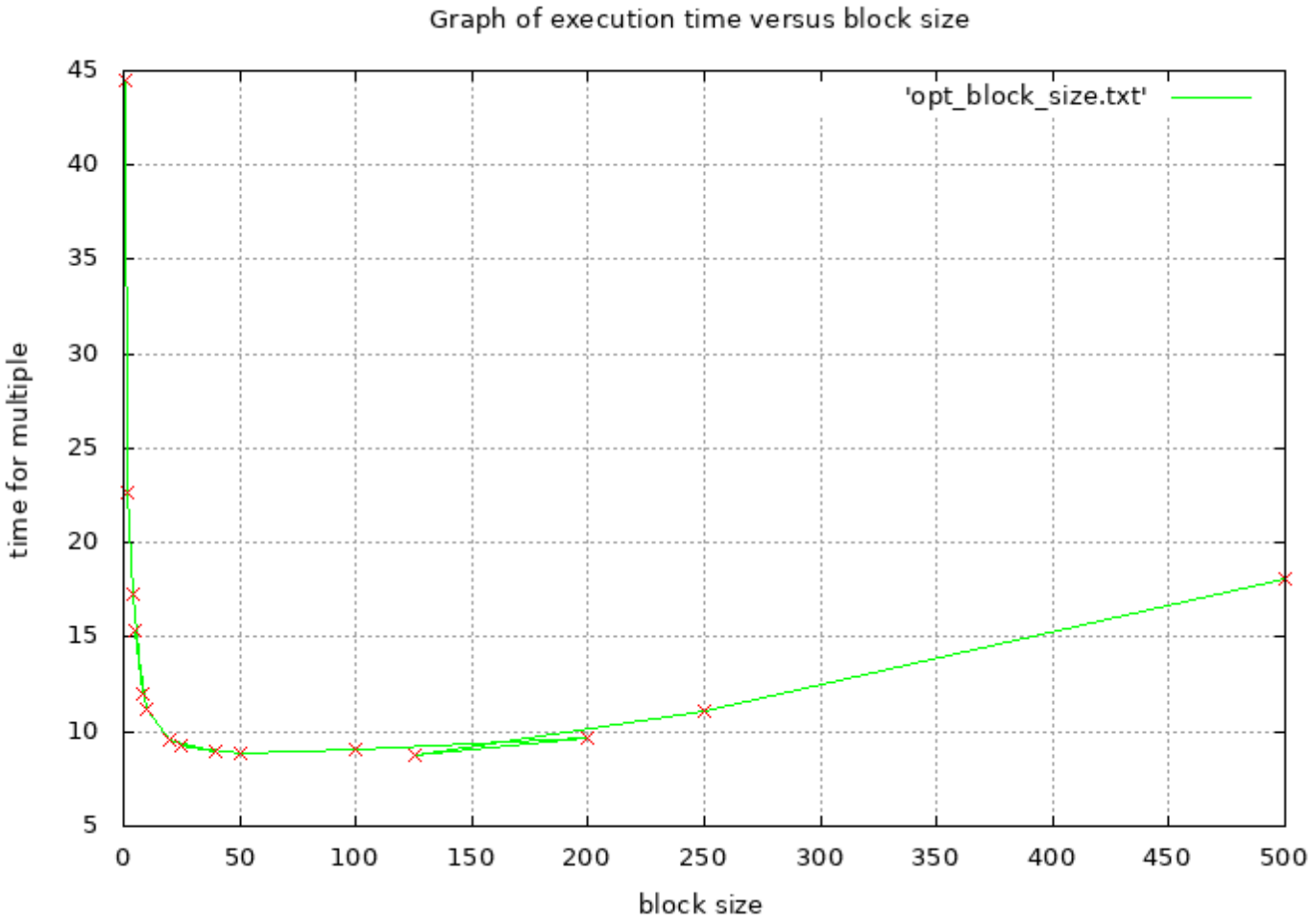



Рисунок №2. «График зависимости времени выполнения перемножения матрицы с помощью блочного перебора от размера блока.»

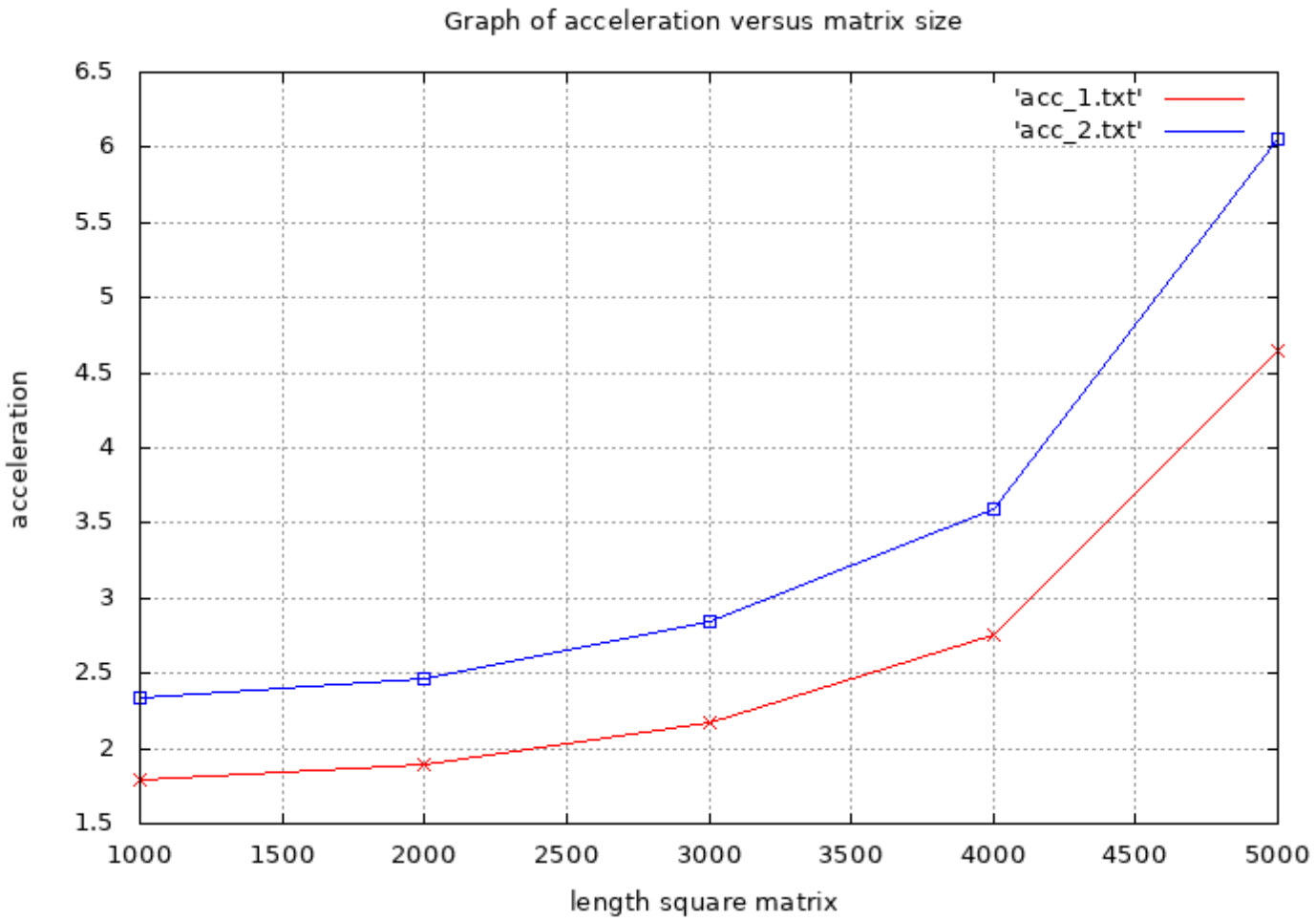
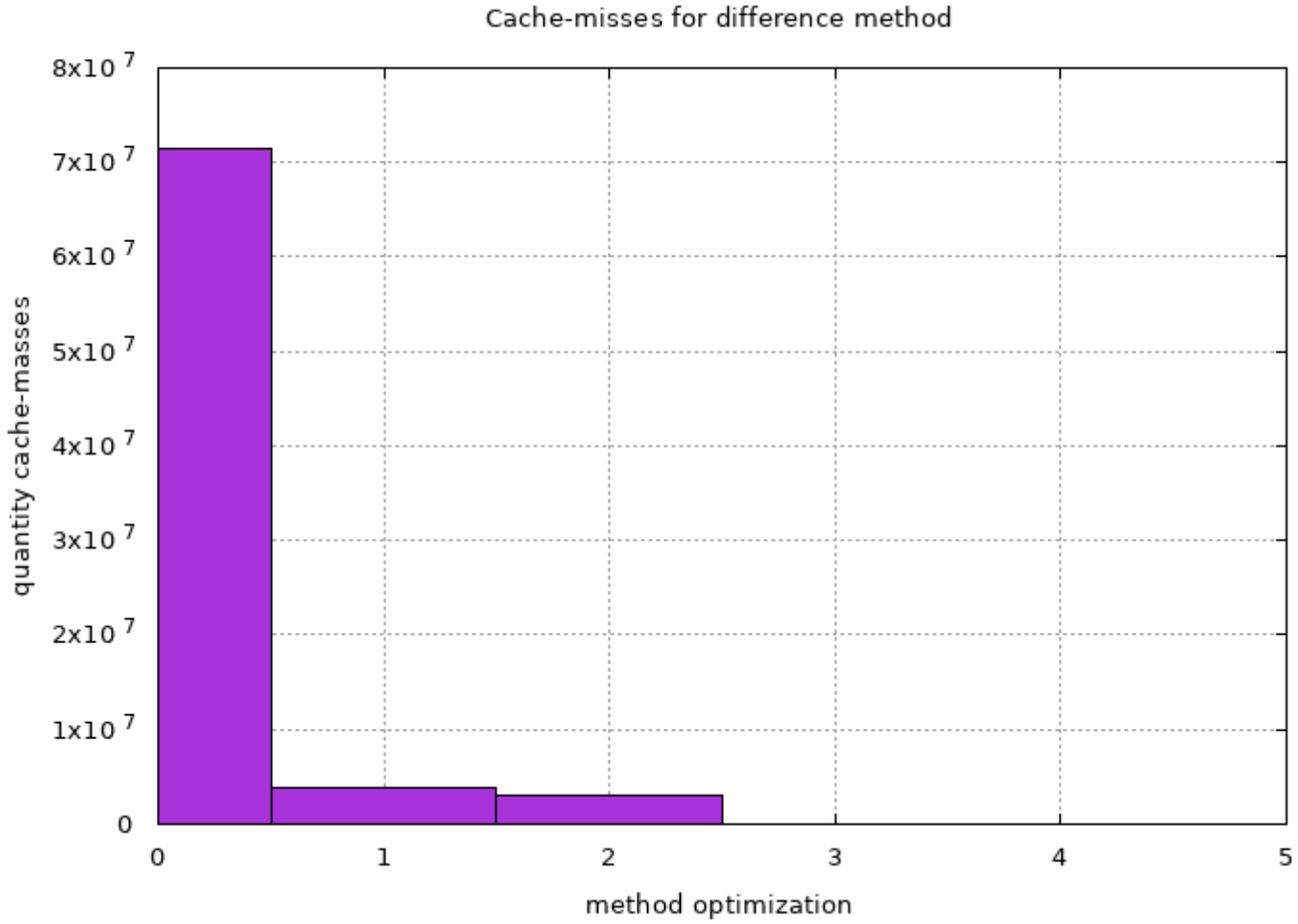



Рисунок №3. «График ускорения оптимизированных вариантов перемножения матриц относительно стандартного метода.»




Рисунок№ 4. «График количества промахов при работе с кешем при различных способах доступа к памяти.»

## Приложение

## *acs_4.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// #define SEVERAL_ITERATION 10

#define ERROR_ALLOCATION_MEMORY 0b00000001
#define ERROR_PARAMETR_FOR_CONVERT_TO_INTEGER_FROM_ARGV 0b00000010
#define ERROR_QUANTITY_ARGUMENT_IN_MAIN 0b00000100
#define ERROR_OPEN_FILE_TABLE_TIME_TXT 0b00001000
```

```
#define ERROR_SIZE_MATRIX_TOO_LARGE 0b00010000
#define ERROR_OPTIMIZE_ARGUMENT_FROM_MAIN 0b00100000

#define INDEX_SIZE_SQUARE_MATRIX_FROM_ARGV 1
#define INDEX_OPTIMIZE_TYPE_FROM_ARGV 2
#define INDEX_BLOCK_SIZE_FROM_ARGV 3

#define MINIMAL_QUANTITY_ARGUMENT_IN_MAIN 3
#define DIGIT_CAPACITY_IN_DECIMAL 10
#define STEP_SIZE_MATRIX 1000
#define MAX_SIZE_SQUARE_MATRIX 5000

#define NUMBER_FOR_SAVE_FROM_DIVISION_BY_ZERO 0.5

#define FIRST_CODE_SYMBOL_BY_DIGIT 0x30
#define LAST_CODE_SYMBOL_BY_DIGIT 0x39
#define CONVERT_CODE_SYMBOL_BY_DIGIT_IN_NUMBER 0x0F

#define OPTIMIZE_NONE 'N'
#define OPTIMIZE_LINE_BY_LINE 'L'
#define OPTIMIZE_BLOCK_ENUMERATION 'B'

#define ADDITIONAL_RECORDING "a"
#define FILE_NAME_FOR_TABLE_TIME_OPT_0 "opt_0.txt"
#define FILE_NAME_FOR_TABLE_TIME_OPT_1 "opt_1.txt"
#define FILE_NAME_FOR_TABLE_TIME_OPT_2 "opt_2.txt"

int Check_Argc(int argc)
{
   if(argc < MINIMAL_QUANTITY_ARGUMENT_IN_MAIN)
   {
      return 0;
   }
   return ERROR_QUANTITY_ARGUMENT_IN_MAIN;
}

int Convert_Char_Argument_To_Integer(int* integer_value_argv, char argv[])
{
   for(short int i = 0; argv[i] != '\0'; i++)
   {
      if((argv[i] >= FIRST_CODE_SYMBOL_BY_DIGIT) && (argv[i] <=
LAST_CODE_SYMBOL_BY_DIGIT))
      {
```

```
        (*integer_value_argv) = ((*integer_value_argv) + (int)(argv[i] &
CONVERT_CODE_SYMBOL_BY_DIGIT_IN_NUMBER));
        (*integer_value_argv) = ((*integer_value_argv) * DIGIT_CAPACITY_IN_DECIMAL);
      }
   }
   (*integer_value_argv) = ((*integer_value_argv) / DIGIT_CAPACITY_IN_DECIMAL);
   if((*integer_value_argv) > 0)
   {
      return 0;
   }
   return ERROR_PARAMETR_FOR_CONVERT_TO_INTEGER_FROM_ARGV;
}

void Print_Matrix_Square(double* matrix[], int size_matrix)
{
        for(short int i = 0; i < size_matrix; i++)
        {
                for(short int j = 0; j < size_matrix; j++)
                {
                        printf("%.4lf ",matrix[i][j]);
                }
                printf("\n");
        }
        printf("\n");
}

int Fill_Rand_Matrix_Square(double** matrix[], int size_matrix)
{
        (*matrix) = (double**)calloc(size_matrix,sizeof(double*));
        if(*matrix == NULL)
        {
                return ERROR_ALLOCATION_MEMORY;
        }
        for(short int i = 0; i < size_matrix; i++)
        {
                (*matrix)[i] = (double*)calloc(size_matrix,sizeof(double));
                if((*matrix)[i] == NULL)
                {
                        return ERROR_ALLOCATION_MEMORY;
                }
                for(short int j = 0; j < size_matrix; j++)
                {
```

```
                (*matrix)[i][j] = ((double)(rand()%DIGIT_CAPACITY_IN_DECIMAL) /
((double)(rand()%DIGIT_CAPACITY_IN_DECIMAL) +
NUMBER_FOR_SAVE_FROM_DIVISION_BY_ZERO));
            }
        }
        return 0;
}

int Multiple_Matrix_Square_Standart(double** matrix_result[], double* matrix_first[],
double* matrix_second[], int size_matrix)
{
        (*matrix_result) = (double**)calloc(size_matrix, sizeof(double*));
        if(*matrix_result == NULL)
        {
            return ERROR_ALLOCATION_MEMORY;
        }
        for(short int i = 0; i < size_matrix; i++)
        {
            (*matrix_result)[i] = (double*)calloc(size_matrix, sizeof(double));
            if((*matrix_result)[i] == NULL)
            {
                return ERROR_ALLOCATION_MEMORY;
            }
            for(short int j = 0; j < size_matrix; j++)
            {
                for(short int k = 0; k < size_matrix; k++)
                {
                    (*matrix_result)[i][j] = ((*matrix_result)[i][j] + (matrix_first[i][k] *
matrix_second[k][j]));
                }
            }
        }
        return 0;
}

int Multiple_Matrix_Square_Line_By_Line(double** matrix_result[], double* matrix_first[],
double* matrix_second[], int size_matrix)
{
        (*matrix_result) = (double**)calloc(size_matrix, sizeof(double*));
        if(*matrix_result == NULL)
        {
            return ERROR_ALLOCATION_MEMORY;
        }
```

```
	for(short int i = 0; i < size_matrix; i++)
	{
		(*matrix_result)[i] = (double*)calloc(size_matrix, sizeof(double));
		if((*matrix_result)[i] == NULL)
		{
			return ERROR_ALLOCATION_MEMORY;
		}
		for(short int j = 0; j < size_matrix; j++)
		{
			for(short int k = 0; k < size_matrix; k++)
			{
				(*matrix_result)[i][k] = ((*matrix_result)[i][k] + (matrix_first[i][j] *
matrix_second[j][k]));
			}
		}
	}
	return 0;
}

int Multiple_Matrix_Square_Block_Enumeration(double** matrix_result[], double*
matrix_first[], double* matrix_second[], int size_matrix, int block_size)
{
	(*matrix_result) = (double**)calloc(size_matrix, sizeof(double*));
	if(*matrix_result == NULL)
	{
		return ERROR_ALLOCATION_MEMORY;
	}
	for(short int i = 0; i < size_matrix; i++)
	{
		(*matrix_result)[i] = (double*)calloc(size_matrix, sizeof(double));
		if((*matrix_result)[i] == NULL)
		{
			return ERROR_ALLOCATION_MEMORY;
		}
	}
	double temp = 0;
	for(short int jj = 0; jj < size_matrix; jj += block_size)
	{
		for(short int kk = 0; kk < size_matrix; kk += block_size)
		{
			for(short int i=0; i < size_matrix; i++)
			{
```

```
                        for(short int j = jj; j < ((jj + block_size) > size_matrix ? size_matrix:(jj
+ block_size)); j++)
                        {
                                temp = 0;
                                for(short int k = kk; k < ((kk + block_size) > size_matrix ?
size_matrix:(kk + block_size)); k++)
                                {
                                        temp += (matrix_first[i][k] * matrix_second[k][j]);
                                }
                                (*matrix_result)[i][j] += temp;
                        }
                }
        }
        }
        return 0;
}

int main(int argc, char* argv[])
{
        double** matrix_first = NULL;
        double** matrix_second = NULL;
        double** matrix_result = NULL;
        FILE* file = NULL;
        double value_time = 0.0;
        #ifdef SEVERAL_ITERATION
        double average_time = 0.0;
        #endif
        int error_flag = 0b0;
        int size_square_matrix = 0;
        int block_size = 0;
        srand(time(NULL));
        error_flag = (error_flag | Check_Argc(argc));
        error_flag = (error_flag | Convert_Char_Argument_To_Integer(&size_square_matrix,
argv[INDEX_SIZE_SQUARE_MATRIX_FROM_ARGV]));
        if(MAX_SIZE_SQUARE_MATRIX < size_square_matrix)
        {
                return (error_flag | ERROR_SIZE_MATRIX_TOO_LARGE);
        }
        if(!((argv[INDEX_OPTIMIZE_TYPE_FROM_ARGV] != NULL) &&
(argv[INDEX_OPTIMIZE_TYPE_FROM_ARGV][1] == '\0')))
        {
                return (error_flag | ERROR_OPTIMIZE_ARGUMENT_FROM_MAIN);
        }
```

```
	error_flag = (error_flag | Fill_Rand_Matrix_Square(&matrix_first,
size_square_matrix));
	error_flag = (error_flag | Fill_Rand_Matrix_Square(&matrix_second,
size_square_matrix));
	if(argc == MINIMAL_QUANTITY_ARGUMENT_IN_MAIN)
	{
		if(argv[INDEX_OPTIMIZE_TYPE_FROM_ARGV][0] == OPTIMIZE_NONE)
		{
			file =
fopen(FILE_NAME_FOR_TABLE_TIME_OPT_0,ADDITIONAL_RECORDING);
			if(file == NULL)
			{
				return (error_flag | ERROR_OPEN_FILE_TABLE_TIME_TXT);
			}
			#ifdef SEVERAL_ITERATION
			for(short int i = 0; i < SEVERAL_ITERATION; i++)
			{
				#endif
				value_time = clock();
				error_flag = (error_flag |
Multiple_Matrix_Square_Standart(&matrix_result, matrix_first, matrix_second,
size_square_matrix));
				value_time = (clock() - value_time);
				#ifdef SEVERAL_ITERATION
				for(short int i = 0; i < size_square_matrix; i++)
				{
					free(matrix_result[i]);
					matrix_result[i] = NULL;
				}
				free(matrix_result);
				average_time = (average_time + value_time);
			}
			value_time = (average_time / SEVERAL_ITERATION);
			#endif
		}
		if(argv[INDEX_OPTIMIZE_TYPE_FROM_ARGV][0] == OPTIMIZE_LINE_BY_LINE)
		{
			file =
fopen(FILE_NAME_FOR_TABLE_TIME_OPT_1,ADDITIONAL_RECORDING);
			if(file == NULL)
			{
				return (error_flag | ERROR_OPEN_FILE_TABLE_TIME_TXT);
			}
```

```
                        #ifdef SEVERAL_ITERATION
                        for(short int i = 0; i < SEVERAL_ITERATION; i++)
                        {
                                #endif
                                value_time = clock();
                                error_flag = (error_flag |
Multiple_Matrix_Square_Line_By_Line(&matrix_result, matrix_first, matrix_second,
size_square_matrix));
                                value_time = (clock() - value_time);
                                #ifdef SEVERAL_ITERATION
                                for(short int i = 0; i < size_square_matrix; i++)
                                {
                                        free(matrix_result[i]);
                                        matrix_result[i] = NULL;
                                }
                                free(matrix_result);
                                average_time = (average_time + value_time);
                        }
                        value_time = (average_time / SEVERAL_ITERATION);
                        #endif
                }
        }
        else
        {
                if(argv[INDEX_OPTIMIZE_TYPE_FROM_ARGV][0] ==
OPTIMIZE_BLOCK_ENUMERATION)
                {
                        error_flag = (error_flag |
Convert_Char_Argument_To_Integer(&block_size,
argv[INDEX_BLOCK_SIZE_FROM_ARGV]));
                        file =
fopen(FILE_NAME_FOR_TABLE_TIME_OPT_2,ADDITIONAL_RECORDING);
                        if(file == NULL)
                        {
                                return (error_flag | ERROR_OPEN_FILE_TABLE_TIME_TXT);
                        }
                        #ifdef SEVERAL_ITERATION
                        for(short int i = 0; i < SEVERAL_ITERATION; i++)
                        {
                                #endif
                                value_time = clock();
```

```
                        error_flag = (error_flag |
Multiple_Matrix_Square_Block_Enumeration(&matrix_result, matrix_first, matrix_second,
size_square_matrix, block_size));
                        value_time = (clock() - value_time);
                        #ifdef SEVERAL_ITERATION
                        for(short int i = 0; i < size_square_matrix; i++)
                        {
                                free(matrix_result[i]);
                                matrix_result[i] = NULL;
                        }
                        free(matrix_result);
                        average_time = (average_time + value_time);
                }
                value_time = (average_time / SEVERAL_ITERATION);
                #endif
        }
        else
        {
                return (error_flag | ERROR_OPTIMIZE_ARGUMENT_FROM_MAIN);
        }
}
for(short int i = 0; i < size_square_matrix; i++)
{
        free(matrix_first[i]);
        matrix_first[i] = NULL;
}
for(short int i = 0; i < size_square_matrix; i++)
{
        free(matrix_second[i]);
        matrix_second[i] = NULL;
}
#ifndef SEVERAL_ITERATION
for(short int i = 0; i < size_square_matrix; i++)
{
        free(matrix_result[i]);
        matrix_result[i] = NULL;
}
free(matrix_result);
#endif
free(matrix_first);
free(matrix_second);
matrix_first = NULL;
matrix_second = NULL;
```

```
        matrix_result = NULL;
        value_time = (value_time / CLOCKS_PER_SEC);
        fprintf(file,"%-14d %lf\n", size_square_matrix, value_time);
        fclose(file);
        file = NULL;
        return error_flag;
}
```

## *acceleration.c*

```
#include <stdio.h>

int main()
{
   FILE* file_r0 = NULL;
   FILE* file_r1 = NULL;
   FILE* file_r2 = NULL;
   FILE* file_w1 = NULL;
   FILE* file_w2 = NULL;
   double time_0 = 0.0;
   double time_1 = 0.0;
   double time_2 = 0.0;
   int size = 0;
   file_r0 = fopen("opt_0.txt","r");
   if(file_r0 == NULL)
   {
      return -1;
   }
   file_r1 = fopen("opt_1.txt","r");
   if(file_r1 == NULL)
   {
      return -1;
   }
   file_r2 = fopen("opt_2.txt","r");
   if(file_r2 == NULL)
   {
      return -1;
   }
   file_w1 = fopen("acc_1.txt","w");
```

```
    if(file_w1 == NULL)
    {
        return -1;
    }
    file_w2 = fopen("acc_2.txt","w");
    if(file_w2 == NULL)
    {
        return -1;
    }
    while(getc(file_r0) != '\n');
    while(getc(file_r1) != '\n');
    while(getc(file_r2) != '\n');
    do
    {
        fscanf(file_r0,"%d",&size);
        fscanf(file_r1,"%d",&size);
        fscanf(file_r2,"%d",&size);
        getc(file_r0);
        getc(file_r1);
        getc(file_r2);
        fscanf(file_r0,"%lf",&time_0);
        fscanf(file_r1,"%lf",&time_1);
        fscanf(file_r2,"%lf",&time_2);
        getc(file_r0);
        getc(file_r1);
        getc(file_r2);
        fprintf(file_w1,"%d %lf\n", size, (time_0/time_1));
        fprintf(file_w2,"%d %lf\n", size, (time_0/time_2));
    }while(!feof(file_r0));
    fclose(file_w1);
    fclose(file_w2);
    fclose(file_r0);
    fclose(file_r1);
    fclose(file_r2);
}
```

# *opt_block_size.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define SEVERAL_ITERATION 10

#define ERROR_ALLOCATION_MEMORY 0b00000001

#define NUMBER_FOR_SAVE_FROM_DIVISION_BY_ZERO 0.5
#define DIGIT_CAPACITY_IN_DECIMAL 10

int Fill_Rand_Matrix_Square(double** matrix[], int size_matrix)
{
	(*matrix) = (double**)calloc(size_matrix,sizeof(double*));
	if(*matrix == NULL)
	{
		return ERROR_ALLOCATION_MEMORY;
	}
	for(short int i = 0; i < size_matrix; i++)
	{
		(*matrix)[i] = (double*)calloc(size_matrix,sizeof(double));
		if((*matrix)[i] == NULL)
		{
			return ERROR_ALLOCATION_MEMORY;
		}
		for(short int j = 0; j < size_matrix; j++)
		{
			(*matrix)[i][j] = ((double)(rand()%DIGIT_CAPACITY_IN_DECIMAL) /
((double)(rand()%DIGIT_CAPACITY_IN_DECIMAL) +
NUMBER_FOR_SAVE_FROM_DIVISION_BY_ZERO));
		}
	}
	return 0;
}

int Multiple_Matrix_Square_Block_Enumeration(double** matrix_result[], double*
matrix_first[], double* matrix_second[], int size_matrix, int block_size)
{
```

```
	(*matrix_result) = (double**)calloc(size_matrix, sizeof(double*));
	if(*matrix_result == NULL)
	{
		return ERROR_ALLOCATION_MEMORY;
	}
	for(short int i = 0; i < size_matrix; i++)
	{
		(*matrix_result)[i] = (double*)calloc(size_matrix, sizeof(double));
		if((*matrix_result)[i] == NULL)
		{
			return ERROR_ALLOCATION_MEMORY;
		}
	}
	double temp = 0;
	for(short int jj = 0; jj < size_matrix; jj += block_size)
	{
		for(short int kk = 0; kk < size_matrix; kk += block_size)
		{
			for(short int i=0; i < size_matrix; i++)
			{
				for(short int j = jj; j < ((jj + block_size) > size_matrix ? size_matrix:(jj
+ block_size)); j++)
				{
					temp = 0;
					for(short int k = kk; k < ((kk + block_size) > size_matrix ?
size_matrix:(kk + block_size)); k++)
					{
						temp += (matrix_first[i][k] * matrix_second[k][j]);
					}
					(*matrix_result)[i][j] += temp;
				}
			}
		}
	}
	return 0;
}

int main()
{
   double** matrix_first = NULL;
	double** matrix_second = NULL;
	double** matrix_result = NULL;
   double value_time = 0.0;
```

```
        double average_time = 0.0;
    FILE* file = NULL;
    int size_square_matrix = 1000;
        int block_size = 0;
    int error_flag = 0b0;
        srand(time(NULL));
    file = fopen("opt_block_size.txt","a");
    if(file == NULL)
    {
      return -1;
    }
    for(short int rate_first = 0, rate_second = 0; block_size < 500;rate_second++)
    {
      if(rate_second > 3)
      {
        rate_second = 0;
        rate_first++;
      }
      block_size = (pow(5,rate_first) * (pow(2,rate_second)));
      printf("block: %d\n",block_size);
      error_flag = (error_flag | Fill_Rand_Matrix_Square(&matrix_first, size_square_matrix));
      error_flag = (error_flag | Fill_Rand_Matrix_Square(&matrix_second,
size_square_matrix));
      printf("start\n");
      for(short int i = 0; i < SEVERAL_ITERATION; i++)
      {
        printf("%d)iterstart\n",i);
        value_time = clock();
        error_flag = (error_flag |
Multiple_Matrix_Square_Block_Enumeration(&matrix_result, matrix_first, matrix_second,
size_square_matrix, block_size));
        value_time = (clock() - value_time);
        for(short int i = 0; i < size_square_matrix; i++)
        {
          free(matrix_result[i]);
          matrix_result[i] = NULL;
        }
        free(matrix_result);
        average_time = (average_time + value_time);
      }
      value_time = (average_time / SEVERAL_ITERATION);
      average_time = 0.0;
      for(short int i = 0; i < size_square_matrix; i++)
```

```
        {
            free(matrix_first[i]);
            matrix_first[i] = NULL;
        }
        for(short int i = 0; i < size_square_matrix; i++)
        {
            free(matrix_second[i]);
            matrix_second[i] = NULL;
        }
        free(matrix_first);
        free(matrix_second);
        matrix_first = NULL;
        matrix_second = NULL;
        matrix_result = NULL;
        value_time = (value_time / CLOCKS_PER_SEC);
        fprintf(file,"%-14d %lf\n", block_size, value_time);
    }
    fclose(file);
          file = NULL;
          return error_flag;
}
```

# *acs_4.sh*

```
#!/bin/bash
rm opt_0.txt;
rm opt_1.txt;
rm opt_2.txt;
rm key;
touch opt_0.txt;
touch opt_1.txt;
touch opt_2.txt;
echo '#Size_Matrix | Time_Execute' >> opt_0.txt;
echo '#Size_Matrix | Time_Execute' >> opt_1.txt
echo '#Size_Matrix | Time_Execute' >> opt_2.txt;
gcc -Wall -Werror -o key acs_4.c;
for((i=1000;i<6000;i=i+1000))
do
./key $i N;
./key $i L;
./key $i B 125;
echo 'iteration end';
done
gnuplot draw_time.plt
xdg-open time_for_multiple.png
```