МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №7
по курсу «ЭВМ и периферийные устройства»

ВЕКТОРИЗАЦИЯ ВЫЧИСЛЕНИЙ

Выполнил: студент 2-го курса гр. 17208
Гафиятуллин А.Р.

Новосибирск, 2018

1. ЦЕЛИ РАБОТЫ:

1. Изучение SIMD-расширений архитектуры x86/x86-64;
2. Изучение способов использования SIMD-расширений в программах на языке Си;
3. Получение навыков использования SIMD-расширений.

2. ХОД РАБОТЫ:

1. Для достижения поставленных целей были написаны три варианта программы, реализующей алгоритм из задания:
   a. без ручной векторизации;
   b. с использованием встроенных SIMD-функций компилятора;
   c. с использованием оптимизированной библиотеки BLAS.
2. Программы были протестированы на 3-х небольших наборах тестовых данных(матрицы порядка 3 и 4). Тестирование происходило на Linux - машине со следующими характеристиками: ОС Elementary OS(Ubuntu-based, Linux kernel 4.15.0-39-generic), процессор Intel Core i5-7200U CPU @ 3.1GHz, загруженность - около 130-140 процессов.

   Тесты(N - порядок квадратной матрицы, M - количество членов ряда):

   1. N = 3, M = 1 000 000

1	7	3
-4	9	4
0	3	2

      Вывод программы a):

0.2308	-0.1923	0.0385
0.3077	0.0769	-0.6154
-0.4615	-0.1154	1.4230

      Вывод программы b):

0.2308	-0.1923	0.0385
0.3077	0.0769	-0.6154
-0.4615	-0.1154	1.4231

Вывод программы c):

0.2308	-0.1923	0.0385
0.3077	0.0769	-0.6154
-0.4615	-0.1154	1.4231

Правильный ответ:

0.2308	-0.1923	0.038
0.3077	0.0769	-0.6154
-0.4615	-0.1154	1.4230

2. N = 4, M = 1 000 000

6	-5	8	4
9	7	5	2
7	5	3	7
-4	8	-8	-3

Вывод программы a):

5.3882	-0.4829	-0.7033	4.9903
-3.0231	0.5002	0.5383	-2.4396
-5.1737	0.5553	0.5836	-4.9163
-1.1290	0.0400	0.3547	-0.9430

Вывод программы b):

5.3875	-0.4829	-0.7031	4.9897
-3.0192	0.5020	0.5392	-2.4428
-5.1753	0.5584	0.5856	-4.9120
-1.1289	0.0399	0.3546	-0.9428

Вывод программы c):

5.3675	-0.4654	-0.6905	4.9917
-2.9656	0.5336	0.5604	-2.4818
-5.1796	0.5876	0.6062	-4.8720
-1.1264	0.0393	0.3541	-0.9405

Правильный ответ:

5.56	-0.77	-0.93	4.73
-3	0.5	0.5	-2.5
-5.36	0.87	0.83	-4.63
-1.12	0.04	0.36	-0.96

3. Результаты измерения времени работы трех программ при N = 2048 и M = 10 (несколько раз генерировалась произвольная матрица):
    a. без ручной векторизации: 812 секунд;
    b. с использованием встроенных SIMD-функций компилятора: 36 секунд;
    c. с использованием оптимизированной библиотеки BLAS: 38 секунд.

    Таким образом, наибольшей скорости работы алгоритма удалось добиться, используя встроенные SIMD-функции компилятора. Связано это, скорее всего, с тем, что в данных программах реализовывался частный случай с квадратными матрицами, а сами функции бедны функционалом и выполняют только одну операцию(в противоположность аналогичным функциям библиотеки BLAS).

4. Полный компилируемый листинг реализованных программ и команды для их компиляции:
    a. без ручной векторизации:

matrix.h:

```c
#include <stdio.h>  //fscanf, fprintf
#include <stdlib.h> //calloc, malloc, free
#include <string.h> //memcpy
#include <time.h>   //time
#include <unistd.h> //sysconf
#include <sys/times.h>//times
#include <math.h>   //fabsf

//---------------------------Matrix creation--------------------------------
struct matrix{
    float* matrix_;
    int order_;
};

//create an uninitialized matrix
struct matrix* create_matrix(int order);

//get a matrix from input stream
void get_matrix(struct matrix* m, FILE* in);

//generate a random matrix
struct matrix* gen_matrix(int order, float range);

//get a get identity matrix
struct matrix* get_identity_matrix(int order);

//print the matrix in the output stream
void print_matrix(struct matrix* m, FILE* out);

//free the memory allocated for the matrix
void free_matrix(struct matrix* m);

//-------------------------Matrix operations--------------------------------
//copy matrix from src to dest
```

```c
void copy_matrix(struct matrix* dst, struct matrix* src);

//transpose the matrix
void transpose_matrix(struct matrix* m);

//summarize two matrices and assign the result to the first operand
void sum_matrices(struct matrix* a, struct matrix* b);

//subtract the second operand from the first
//and assign the result to the first operand
void sub_matrices(struct matrix* a, struct matrix* b);

//matrix multiplication
struct matrix* mul_matrices(struct matrix* a, struct matrix* b);

//matrix multiplication by a scalar
void mul_matrix_on_scalar(struct matrix* m, float scalar);

//get l1 and maximal matrix norms
void get_matrix_norms(struct matrix* m, float* l1_norm, float* max_norm);

//invert the matrix
struct matrix* invert_matrix(struct matrix* A, int iter_number);
```

without_vectorization.c

```c
#include "matrix.h"

struct matrix* create_matrix(int order){
    struct matrix* m = malloc(sizeof(struct matrix));
    m->order_ = order;
    m->matrix_ = calloc(order * order, sizeof(float));
    return m;
}
```

```c
struct matrix* gen_matrix(int order, float range){
    struct matrix* m = malloc(sizeof(struct matrix));
    m->order_ = order;
    m->matrix_ = calloc(order * order, sizeof(float));
    order *= order;
    srand(time(NULL));
    for(int i = 0; i < order; i++)
        m->matrix_[i] = ((float)rand()/(float)(RAND_MAX)) * range;
    return m;
}

void get_matrix(struct matrix* m, FILE* in){
    register int order = m->order_ * m->order_;
    for(int i = 0; i < order; i++)
        fscanf(in, "%f", &m->matrix_[i]);
}

struct matrix* get_identity_matrix(int order){
    struct matrix* m = malloc(sizeof(struct matrix));
    m->order_ = order;
    m->matrix_ = calloc(order * order, sizeof(float));
    for(int i = 0; i < order; i++)
        m->matrix_[i * order + i] = 1;
    return m;
}

void print_matrix(struct matrix* m, FILE* out){
    register int order = m->order_;
    fprintf(out, "Order: %d\n", m->order_);
    for(int i = 0; i < order; i++){
        for(int j = 0; j < order; j++)
            fprintf(out, "%.4f ", m->matrix_[i * order + j]);
        fprintf(out, "\n");
    }
```

```
    fprintf(out, "\n");
}

void free_matrix(struct matrix* m){
    free(m->matrix_);
    free(m);
}

void copy_matrix(struct matrix* dst, struct matrix* src){
    memcpy(dst->matrix_, src->matrix_, sizeof(float) * src->order_ *
src->order_);
    dst->order_ = src->order_;
}

void transpose_matrix(struct matrix* m){
    register int order = m->order_;
    for(int i = 0; i < order; i++)
        for(int j = i; j < order; j++){
            float tmp = m->matrix_[i * order + j];
            m->matrix_[i * order + j] = m->matrix_[j * order + i];
            m->matrix_[j * order + i] = tmp;
    }
}

void sum_matrices(struct matrix* a, struct matrix* b){
    register int order = a->order_ * a->order_;
    for(int i = 0; i < order; i++)
        a->matrix_[i] += b->matrix_[i];
}

void sub_matrices(struct matrix* a, struct matrix* b){
    register int order = a->order_ * a->order_;
    for(int i = 0; i < order; i++)
        a->matrix_[i] -= b->matrix_[i];
```

```c
}

struct matrix* mul_matrices(struct matrix* a, struct matrix* b){
    register int order = a->order_;
    struct matrix* tmp = create_matrix(order);
    for(int i = 0; i < order; i++)
        for(int j = 0; j < order; j++)
            for(int k = 0; k < order; k++)
                tmp->matrix_[i * order + j] += a->matrix_[i * order + k]
                    * b->matrix_[k * order + j];
    return tmp;
}

void mul_matrix_on_scalar(struct matrix* m, float scalar){
    register int order = m->order_ * m->order_;
    for(int i = 0; i < order; i++)
        m->matrix_[i] *= scalar;
}

void get_matrix_norms(struct matrix* m, float* l1_norm, float* max_norm){
    register int order = m->order_;
    *l1_norm = 0;
    *max_norm = 0;
    for(int i = 0; i < order; i++){
        float row_sum = 0;
        float col_sum = 0;
        for(int j = 0; j < order; j++){
            row_sum += fabsf(m->matrix_[i * order + j]);
            col_sum += fabsf(m->matrix_[j * order + i]);
        }
        if(*l1_norm < col_sum)
            *l1_norm = col_sum;
        if(*max_norm < row_sum)
            *max_norm = row_sum;
```

```c
    }
}

struct matrix* invert_matrix(struct matrix* A, int iter_number){
    struct matrix* B = create_matrix(A->order_);
    copy_matrix(B, A);
    transpose_matrix(B);
    float l1_norm, max_norm;
    get_matrix_norms(A, &l1_norm, &max_norm);
    mul_matrix_on_scalar(B, 1.0 / (l1_norm * max_norm));//A^(T) / (l1 * max)
= B
    struct matrix* R = get_identity_matrix(A->order_);
    struct matrix* BA = mul_matrices(B, A);
    sub_matrices(R, BA);    //R = I - BA
    free_matrix(BA);
    struct matrix* R_1_deg = create_matrix(A->order_);
    copy_matrix(R_1_deg, R);
    struct matrix* inv_A = get_identity_matrix(A->order_);
    struct matrix* tmp;
    for(int i = 0; i < iter_number; i++){
        sum_matrices(inv_A, R); //I + R^2 + ... + R^(i)
        tmp = mul_matrices(R, R_1_deg); //R^(i)*R = R^(i+1)
        free_matrix(R);
        R = tmp;
    }
    tmp = mul_matrices(inv_A, B);   //(I + R^2 + R^3 + ...)*B
    free_matrix(inv_A);
    inv_A = tmp;
    free_matrix(B);
    free_matrix(R);
    free_matrix(R_1_deg);
    return inv_A;
}
```

test.c

```
#include "matrix.h"

int main(){
    struct tms start, finish;
    long long int clocks_per_sec = sysconf(_SC_CLK_TCK);
    FILE* input = fopen("input.txt", "r");
    FILE* output = fopen("output.txt", "w");
    int N = 0, M = 0;
    fscanf(input, "%d%d", &N, &M);
    struct matrix* A = gen_matrix(N, 5.0);
    time_t start_real = time(NULL);
    times(&start);
    struct matrix* inv_A = invert_matrix(A, M);
    times(&finish);
    time_t finish_real = time(NULL);
    double total_process_time = finish.tms_utime - start.tms_utime;
    struct matrix* rez = mul_matrices(inv_A, A);    //test an answer by
multiplication it on the source matrix
    print_matrix(rez, output);
    printf("Total process time: %lf sec.\n", total_process_time /
clocks_per_sec);
    printf("Total real time: %ld sec.\n", finish_real - start_real);
    free_matrix(A);
    free_matrix(inv_A);
    free_matrix(rez);
    fclose(input);
    fclose(output);
    return 0;
}
```

Команда компиляции: gcc -O2 without_vectorization.c test.c

b. с использованием встроенных SIMD-функций компилятора:

matrix.h

```
#include <stdio.h>  //fscanf, fprintf
#include <stdlib.h> //calloc, malloc
#include <string.h> //memcpy
#include <time.h>   //time
#include <unistd.h> //sysconf
#include <sys/times.h>//times
#include <xmmintrin.h>  //operations with vectors
#include <math.h>   // fabsf
#define  ALIGN 16
//----------------------------Matrix creation--------------------------------
struct matrix{
    float* matrix_;
    int order_;
    int align_; //align matrix_'s ending to 16 bytes
};

//create an uninitialized matrix
struct matrix* create_matrix(int order);

//get a matrix from input stream
void get_matrix(struct matrix* m, FILE* in);

//generate a random matrix
struct matrix* gen_matrix(int order, float range);

//get a get identity matrix
struct matrix* get_identity_matrix(int order);

//print the matrix in the output stream
void print_matrix(struct matrix* m, FILE* out);
```

```c
//free the memory allocated for the matrix
void free_matrix(struct matrix* m);

//-------------------------Matrix operations------------------------------
//copy matrix from src to dest
void copy_matrix(struct matrix* dst, struct matrix* src);

//transpose the matrix
void transpose_matrix(struct matrix* m);

//summarize two matrices and assign the result to the first operand
void sum_matrices(struct matrix* a, struct matrix* b);

//subtract the second operand from the first
//and assign the result to the first operand
void sub_matrices(struct matrix* a, struct matrix* b);

//matrix multiplication
struct matrix* mul_matrices(struct matrix* a, struct matrix* b);

//matrix multiplication by a scalar
void mul_matrix_on_scalar(struct matrix* m, float scalar);

//get l1 and maximal matrix norms
void get_matrix_norms(struct matrix* m, float* l1_norm, float* max_norm);

//invert the matrix
struct matrix* invert_matrix(struct matrix* A, int iter_number);
```

with_vectorization.c

```c
#include "matrix.h"

struct matrix* create_matrix(int order){
    struct matrix* m = malloc(sizeof(struct matrix));
```

```
    m->order_ = order;
    m->align_ = (order % 4 == 0 ? 0 : 4 - (order % 4)); //calculate an
alignment
    m->matrix_ = _mm_malloc(order * (order + m->align_) * sizeof(float),
ALIGN); //allocate memory with alignment
    return m;
}

void get_matrix(struct matrix* m, FILE* in){
    register int order = m->order_;
    register int real_order = order + m->align_;
    for(int i = 0; i < order; i++)
        for(int j = 0; j < order; j++)
            fscanf(in, "%f", &m->matrix_[i * real_order + j]);
}

struct matrix* gen_matrix(int order, float range){
    struct matrix* m = malloc(sizeof(struct matrix));
    m->order_ = order;
    m->align_ = (order % 4 == 0 ? 0 : 4 - (order % 4));
    register int real_order = order + m->align_;
    m->matrix_ = _mm_malloc(order * real_order * sizeof(float), ALIGN);
    srand(time(NULL));
    for(int i = 0; i < order; i++)
        for (int j = 0; j < order; j++)
            m->matrix_[i * real_order + j] = ((float)rand() /
(float)(RAND_MAX)) * range;
    return m;
}

struct matrix* get_identity_matrix(int order){
    struct matrix* m = malloc(sizeof(struct matrix));
    m->order_ = order;
    m->align_ = (order % 4 == 0 ? 0 : 4 - (order % 4));
```

```
    register int real_order = order + m->align_;
    m->matrix_ = _mm_malloc(order *real_order * sizeof(float), ALIGN);
    memset(m->matrix_, 0, order * real_order * sizeof(float));
    for(int i = 0; i < order; i++)
        m->matrix_[i * real_order + i] = 1;
    return m;
}

void print_matrix(struct matrix* m, FILE* out){
    register int order = m->order_;
    register int real_order = order + m->align_;
    fprintf(out, "Order: %d\n", m->order_);
    for(int i = 0; i < order; i++){
        for(int j = 0; j < order; j++)
            fprintf(out, "%.4f ", m->matrix_[i * real_order + j]);
        fprintf(out, "\n");
    }
    fprintf(out, "\n");
}

void free_matrix(struct matrix* m){
    _mm_free(m->matrix_);
    free(m);
}

void copy_matrix(struct matrix* dst, struct matrix* src){
    memcpy(dst->matrix_, src->matrix_, src->order_ * (src->order_ +
src->align_) * sizeof(float));
    dst->order_ = src->order_;
    dst->align_ = src->align_;
}

void transpose_matrix(struct matrix* m){
    register int order = m->order_;
```

```
    register int real_order = order + m->align_;
    for(int i = 0; i < order; i++)
        for(int j = i; j < order; j++){
            float tmp = m->matrix_[i * real_order + j];
            m->matrix_[i * real_order + j] = m->matrix_[j * real_order + i];
            m->matrix_[j * real_order + i] = tmp;
        }
}

void sum_matrices(struct matrix* a, struct matrix* b){
    register int order = a->order_ * (a->order_ + a->align_) / 4;
    __m128* xx = (__m128*)(a->matrix_);
    __m128* yy = (__m128*)(b->matrix_);
    for(int i = 0; i < order; i++)
            xx[i] = _mm_add_ps(xx[i], yy[i]);
}

void sub_matrices(struct matrix* a, struct matrix* b){
    register int order = a->order_ * (a->order_ + a->align_) / 4;
    __m128* xx = (__m128*)(a->matrix_);
    __m128* yy = (__m128*)(b->matrix_);
    for(int i = 0; i < order; i++)
        xx[i] = _mm_sub_ps(xx[i], yy[i]);
}

struct matrix* mul_matrices(struct matrix* a, struct matrix* b){
    register int order = a->order_;
    register int real_order = (order + a->align_) / 4;
    struct  matrix* tr_b = create_matrix(b->order_);
    copy_matrix(tr_b, b);
    transpose_matrix(tr_b); //transpose right matrix for better performance 
of multiplication
    struct matrix* tmp = create_matrix(order);
    __m128 p, sum;
```

```
    __m128* row = (__m128*)(a->matrix_);
    __m128* col = (__m128*)(tr_b->matrix_);
    for(int i = 0; i < order; i++)
        for(int j = 0; j < order; j++){
            sum = _mm_setzero_ps();
            for (int k = 0; k < real_order; k++){
                p = _mm_mul_ps(row[i * real_order + k], col[j * real_order +
k]);
                sum = _mm_add_ps(sum, p);
            }
            p = _mm_movehl_ps(p, sum);
            sum = _mm_add_ps(sum, p);
            p = _mm_shuffle_ps(sum, sum, 1);
            sum = _mm_add_ss(sum, p);
            _mm_store_ss(&tmp->matrix_[i * real_order * 4 + j], sum);
        }
    free_matrix(tr_b);
    return tmp;
}

void mul_matrix_on_scalar(struct matrix* m, float scalar){
    float* div_row = _mm_malloc(4 * sizeof(float), ALIGN);
    for(int i = 0; i < 4; i++)
        div_row[i] = scalar;
    __m128* row = (__m128*)(m->matrix_);
    __m128* div = (__m128*)(div_row);
    register int order = m->order_ * (m->order_ + m->align_) / 4;
    for(int i = 0; i < order; i++)
            row[i] = _mm_mul_ps(row[i], *div);
    _mm_free(div_row);
}

void get_matrix_norms(struct matrix* m, float* l1_norm, float* max_norm){
    register int order = m->order_;
```

```
    register int real_order = order + m->align_;
    *l1_norm = 0;
    *max_norm = 0;
    for(int i = 0; i < order; i++){
        float row_sum = 0;
        float col_sum = 0;
        for(int j = 0; j < order; j++){
            row_sum += fabs(m->matrix_[i * real_order + j]);
            col_sum += fabs(m->matrix_[j * real_order + i]);
        }
        if(*l1_norm < col_sum)
            *l1_norm = col_sum;
        if(*max_norm < row_sum)
            *max_norm = row_sum;
    }
}

struct matrix* invert_matrix(struct matrix* A, int iter_number){
    struct matrix* B = create_matrix(A->order_);
    copy_matrix(B, A);
    transpose_matrix(B);
    float l1_norm, max_norm;
    get_matrix_norms(A, &l1_norm, &max_norm);
    mul_matrix_on_scalar(B, 1.0 / (l1_norm * max_norm));
    struct matrix* R = get_identity_matrix(A->order_);
    struct matrix* BA = mul_matrices(B, A);
    sub_matrices(R, BA);
    free_matrix(BA);
    struct matrix* R_1_deg = create_matrix(A->order_);
    copy_matrix(R_1_deg, R);
    struct matrix* inv_A = get_identity_matrix(A->order_);
    struct matrix* tmp;
    for(int i = 0; i < iter_number; i++){
        sum_matrices(inv_A, R);
```

```
        tmp = mul_matrices(R, R_1_deg);
        free_matrix(R);
        R = tmp;
    }
    tmp = mul_matrices(inv_A, B);
    free_matrix(inv_A);
    inv_A = tmp;
    free_matrix(B);
    free_matrix(R);
    free_matrix(R_1_deg);
    return inv_A;
}
```

test.c

```
#include "matrix.h"

int main(){
    struct tms start, finish;
    long long int clocks_per_sec = sysconf(_SC_CLK_TCK);
    FILE* input = fopen("input.txt", "r");
    FILE* output = fopen("output.txt", "w");
    int N = 0, M = 0;
    fscanf(input, "%d%d", &N, &M);
    struct matrix* A = gen_matrix(N, 5.0);
    time_t start_real = time(NULL);
    times(&start);
    struct matrix* inv_A = invert_matrix(A, M);
    times(&finish);
    time_t finish_real = time(NULL);
    double total_process_time = finish.tms_utime - start.tms_utime;
    struct matrix* rez = mul_matrices(inv_A, A);
    print_matrix(rez, output);
    printf("Total process time: %lf sec.\n", total_process_time /
clocks_per_sec);
```

```
    printf("Total real time: %ld sec.\n", finish_real - start_real);
    free_matrix(A);
    free_matrix(inv_A);
    free_matrix(rez);
    fclose(input);
    fclose(output);
    return 0;
}
```

Команда компиляции: gcc -O2 without_vectorization.c test.c

c. с использованием оптимизированной библиотеки BLAS:

```
#include <stdio.h>  //printf
#include <stdlib.h> //malloc, calloc, free
#include <string.h> //memset
#include <time.h>   //time
#include <cblas.h>  //cblas_scopy, cblas_sscal, cblas_sgemm
#include <sys/times.h>  //times
#include <unistd.h> //sysconf
#include <math.h>   //fabsf

float* gen_matrix(int order, float range){
    float* m = malloc(order * order * sizeof(float));
    srand(time(NULL));
    for(int i = 0; i < order * order; i++)
        m[i] = ((float)rand() / (float)(RAND_MAX)) * range;
    return m;
}

void transpose_matrix(float* m, int order){
    for(int i = 0; i < order; i++)
        for(int j = i; j < order; j++){
            float tmp = m[i * order + j];
            m[i * order + j] = m[j * order + i];
            m[j * order + i] = tmp;
```

```c
        }
}

void print_matrix(float* m, int order, FILE* out){
    fprintf(out, "Order: %d\n", order);
    for(int i = 0; i < order; i++){
        for(int j = 0; j < order; j++)
            fprintf(out, "%.4f ", m[i * order + j]);
        fprintf(out, "\n");
    }
    fprintf(out, "\n");
}

void get_matrix_norms(float* m, int order, float* l1_norm, float* max_norm){
    *l1_norm = 0;
    *max_norm = 0;
    for(int i = 0; i < order; i++){
        float row_sum = 0;
        float col_sum = 0;
        for(int j = 0; j < order; j++){
            row_sum += fabs(m[i * order + j]);
            col_sum += fabs(m[j * order + i]);
        }
        if(*l1_norm < col_sum)
            *l1_norm = col_sum;
        if(*max_norm < row_sum)
            *max_norm = row_sum;
    }
}

float* get_identity_matrix(int order){
    float* m = calloc(order * order, sizeof(float));
    for(int i = 0; i < order; i++)
        m[i * order + i] = 1;
```

```
    return m;
}

float* invert_matrix(float* A, int order, int iter_number){
    float* B = malloc(order * order * sizeof(float));
    cblas_scopy(order * order, A, 1, B, 1);
    transpose_matrix(B, order);
    float l1_norm, max_norm;
    get_matrix_norms(A, order, &l1_norm, &max_norm);
    cblas_sscal(order * order, 1.0 / (l1_norm * max_norm), B, 1);
    float* R = get_identity_matrix(order);
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, order, order,
order, -1, B, order, A, order, 1, R, order);
    float* R_1_deg = malloc(order * order * sizeof(float));
    cblas_scopy(order * order, R, 1, R_1_deg, 1);
    float* tmp = NULL;
    float* inv_A = get_identity_matrix(order);
    for(int i = 0; i < iter_number; i++){
        tmp = malloc(order * order * sizeof(float));
        cblas_saxpy(order * order, 1, R, 1, inv_A, 1);
        cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, order, order,
order, 1, R, order, R_1_deg, order, 0, tmp,
                    order);
        free(R);
        R = tmp;
    }
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, order, order,
order, 1, inv_A, order, B, order, 0.0, tmp,
            order);
    free(inv_A);
    inv_A = tmp;
    free(B);
    free(R_1_deg);
    return inv_A;
```

```
}

int main(){
    struct tms start, finish;
    long long int clocks_per_sec = sysconf(_SC_CLK_TCK);
    FILE* input = fopen("input.txt", "r");
    FILE* output = fopen("output.txt", "w");
    int N = 0, M = 0;
    fscanf(input, "%d%d", &N, &M);
    float* A = gen_matrix(N, 5.0);
    time_t start_real = time(NULL);
    times(&start);
    float* inv_A = invert_matrix(A, N, M);
    times(&finish);
    time_t finish_real = time(NULL);
    double total_process_time = finish.tms_utime - start.tms_utime;
    float* rez = malloc(N * N * sizeof(float));
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1, inv_A,
N, A, N, 0, rez, N);
    print_matrix(rez, N, output);
    printf("Total process time: %lf sec.\n", total_process_time /
clocks_per_sec);
    printf("Total real time: %ld sec.\n", finish_real - start_real);
    free(A);
    free(inv_A);
    free(rez);
    fclose(input);
    fclose(output);
    return 0;
}
```

Команда компиляции: gcc -O2 cblas.c -lcblas

3. ВЫВОДЫ:

1. Изучили SIMD-расширения архитектуры x86/x86-64;

2. Изучили способы использования SIMD-расширений в программах на языке Си;
3. Получили навыки использования SIMD-расширений;
4. SIMD-расширения и BLAS дают огромный прирост скорости при работе с векторными или матричными операциями(в нашем случае - примерно в 22 раза);
5. При решении задач, не требующих общности, может оказаться, что выгоднее использовать SIMD-расширения компилятора, а не оптимизированную библиотеку BLAS.