МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

«Векторизация вычислений»

Студента 2 курса, 21211 группы

Петрова Сергея Евгеньевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Антон Юрьевич Кудинов

Новосибирск 2022

СОДЕРЖАНИЕ

ЦЕЛЬ

- *Изучение SIMD-расширений архитектуры x86/x86-64;*
- *Изучение способов использования SIMD-расширений в программах на языке Си;*
- *Получение навыков использования SIMD-расширений;*

ЗАДАНИЕ

Алгоритм обращения матрицы A размером N×N с помощью разложения в ряд: $A^{-1} = (I + R + R^2 + \ldots)B$, *где* $R = I - BA$,

$B = \frac{A^T}{||A||_1 \cdot ||A||_\infty}$, $||A||_1 = max_j \sum_i \left|A_{ij}\right|$, $||A||_\infty = max_i \sum_j \left|A_{ij}\right|$,

I – единичная матрица. Параметры алгоритма: N – размер матрицы, M – число членов ряда.

1. *Написать три варианта программы, реализующей алгоритм из задания:*
   - *вариант без ручной векторизации;*
   - *вариант с ручной векторизацией (выбрать любой вариант из возможных трех: ассемблерная вставка, встроенные функции компилятора, расширение GCC);*
   - *вариант с матричными операциями, выполненными с использованием оптимизированной библиотеки BLAS.*

   Для элементов матриц использовать тип данных float;
2. *Проверить правильность работы программ на нескольких небольших тестовых наборах входных данных;*
3. *Каждый вариант программы оптимизировать по скорости, насколько это возможно;*
4. *Сравнить время работы трех вариантов программы для N = 2048, M = 10;*
5. *Составить отчет по лабораторной работе. Отчет должен содержать следующее:*
   - *Титульный лист;*
   - *Цель лабораторной работы;*
   - *Результаты измерения времени работы трех программ;*
   - *Полный компилируемый листинг реализованных программ и команды для их компиляции;*
   - *Вывод по результатам лабораторной работы.*

ОПИСАНИЕ РАБОТЫ

Пошаговое описание выполненной работы

1. *Написать 3 варианта программы, реализующей алгоритм из задания:*
    a. *вариант без ручной векторизации;*
    b. *вариант с использованием встроенных функций компилятора;*
    c. *вариант с матричными операциями, выполненными с использованием оптимизированной библиотеки BLAS;*
2. *Проверил правильность работы программ на небольшом тестовом наборе входных данных;*

```
/home/evmpu/21211/s.petrov1/lab7/cmake-build-release-remote-host-evmpu/default
3 0 9 0
3 4 8 1
2 4 6 2
6 2 2 6

-0.142865 0.714249 -0.78568 0.142868
-0.253991 0.380928 -0.119045 -0.0238106
0.158722 -0.238101 0.261879 -0.0476267
0.174613 -0.761862 0.738069 0.0476403
```

```
/home/evmpu/21211/s.petrov1/lab7/cmake-build-release-remote-host-evmpu/manual
3 0 9 0
3 4 8 1
2 4 6 2
6 2 2 6

-0.142865 0.714249 -0.78568 0.142868
-0.253991 0.380928 -0.119045 -0.0238106
0.158722 -0.238101 0.261879 -0.0476267
0.174613 -0.761862 0.738069 0.0476403
```

```
/home/evmpu/21211/s.petrov1/lab7/cmake-build-release-remote-host-evmpu/BLAS
3 0 9 0
3 4 8 1
2 4 6 2
6 2 2 6

-0.142865 0.714249 -0.78568 0.142868
-0.253991 0.380928 -0.119045 -0.0238106
0.158722 -0.238101 0.261879 -0.0476267
0.174613 -0.761862 0.738069 0.0476403
```

3. *Провёл несколько оптимизаций кода:*
    a. *Скомпилировал программу с уровнем оптимизации -O3;*

b. *Изменил индексацию в цикле функции Multplication, чтобы получился последовательный обход памяти;*

Не последовательный обход памяти

```
void Multiplication(const float * multiplier1,
                    const float * multiplier2, float * result)
{
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
            {
                if (k == 0) result[N * i + j] = 0;
                result[N * i + j] += multiplier1[N * i + k] *
                                     multiplier2[N * k + j];
            }
}
```

```
/home/evmpu/21211/s.petrov1/lab7/cmake-build-release-remote-host-evmpu/default
Time without vectorization: 651.89 sec.
```

Последовательный обход памяти

```
void Multiplication(const float * multiplier1,
                    const float * multiplier2, float * result)
{
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
            {
                if (j == 0) result[N * i + k] = 0;
                result[N * i + k] += multiplier1[N * i + j] *
                                     multiplier2[N * j + k];
            }
}
```

```
/home/evmpu/21211/s.petrov1/lab7/cmake-build-release-remote-host-evmpu/default
Time without vectorization: 31.3678 sec.
```

c. *Избавился от условного оператора в цикле функции Multiplication;*

С условным оператором в цикле

```
void Multiplication(const float * multiplier1,
                    const float * multiplier2, float * result)
{
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
```

```
            {
                if (j == 0) result[N * i + k] = 0;
                result[N * i + k] += multiplier1[N * i + j] *
                                     multiplier2[N * j + k];
            }
    }
```

```
/home/evmpu/21211/s.petrov1/lab7/cmake-build-release-remote-host-evmpu/default
Time without vectorization: 31.2445 sec.
```

Без условного оператора в цикле

```
    void Multiplication(const float * multiplier1,
                        const float * multiplier2, float * result)
    {
        for (int i = 0; i < N * N; ++i)
            result[i] = 0;

        for (int i = 0; i < N; ++i)
            for (int j = 0; j < N; ++j)
                for (int k = 0; k < N; ++k)
                    result[N * i + k] += multiplier1[N * i + j] *
                                         multiplier2[N * j + k];
    }
```

```
/home/evmpu/21211/s.petrov1/lab7/cmake-build-release-remote-host-evmpu/default
Time without vectorization: 29.7316 sec.
```

Команды для компиляции

```
evmpu@comrade:~$ /usr/bin/cmake
-DCMAKE_BUILD_TYPE=Release
-DCMAKE_C_COMPILER=/usr/bin/gcc
-DCMAKE_CXX_COMPILER=/usr/bin/g++
-G "CodeBlocks - Unix Makefiles"
-S /home/evmpu/21211/s.petrov1/lab7
-B /home/evmpu/21211/s.petrov1/lab7/cmake-build-release

evmpu@comrade:~$ /usr/bin/cmake
--build /home/evmpu/21211/s.petrov1/lab7/cmake-build-release
--target default

evmpu@comrade:~$ /usr/bin/cmake
--build /home/evmpu/21211/s.petrov1/lab7/cmake-build-release
--target manual

evmpu@comrade:~$ /usr/bin/cmake
--build /home/evmpu/21211/s.petrov1/lab7/cmake-build-release
--target BLAS
```

Результаты измерения времени

```
/home/evmpu/21211/s.petrov1/lab7/cmake-build-release/default
Time without vectorization: 29.9737 sec.
```

```
/home/evmpu/21211/s.petrov1/lab7/cmake-build-release/manual
Time with manual vectorization: 29.6524 sec.
```

```
/home/evmpu/21211/s.petrov1/lab7/cmake-build-release/BLAS
Time with BLAS: 1.07232 sec.
```

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы:

- *Изучил SIMD-расширений архитектуры x86/x86-64;*
- *Изучил способов использования SIMD-расширений в программах на языке Си;*
- *Получил навыков использования SIMD-расширений;*
- *Изучил работу оптимизированной библиотеки линейной алгебры BLAS;*

По результатам проведённых исследований можно сделать следующие выводы:

- *Прирост производительности с SIMD расширениями был незначительным, т.к. компилятор уже в традиционном методе вычисления в некоторых местах использовал автовекторизацию простых циклов;*
- *BLAS лучше всего использовать в вычислениях с матрицами;*

ПРИЛОЖЕНИЕ (ЛИСТИНГ ПРОГРАММЫ)

src/default.cpp

```
#include <cfloat>       // FLT_MIN
#include <cmath>        // fabs()
#include <ctime>
#include <iostream>

#define N 2048
#define M 10

void Inverse(const float * matrix, float * result);
float GetMaxSum(const float * matrix); // |A|_1 * |A|_infinity
void FillB(const float * matrix, float * B);
void FillI(float * I);
void Multiplication(const float * multiplier1,
                    const float * multiplier2, float * result);
void Addition(const float * addend1, const float * addend2,
              float * result);
void Subtraction(const float * minuend, const float * subtrahend,
                 float * result);
void Copy(float * dest, const float * src);
//void Print(const float * matrix);

int main()
{
    srandom(time(nullptr));
    auto * matrix = new float [N * N];
    auto * result = new float [N * N];
    timespec start = { 0, 0 };
    timespec end = { 0, 0 };

    for (int i = 0; i < N * N; ++i)
    {
        matrix[i] = float(random());
        matrix[i] *= (random() % 2) ? 1 : -1;
        result[i] = 0;
    }

//    float matrix[N * N] = {
//        3, 0, 9, 0,
//        3, 4, 8, 1,
//        2, 4, 6, 2,
//        6, 2, 2, 6
//    };
//    float result[N * N] = { 0 };

    clock_gettime(CLOCK_MONOTONIC_RAW, &start);
    Inverse(matrix, result);
    clock_gettime(CLOCK_MONOTONIC_RAW, &end);

//    Print(matrix);
//    cout << endl;
//    Print(result);
//    cout << endl;
```

```
    std::cout << "Time without vectorization: "
         << (double)end.tv_sec - (double)start.tv_sec + 1e-9 *
            ((double)end.tv_nsec - (double)start.tv_nsec)
         << " sec." << std::endl;

    return EXIT_SUCCESS;
}

void Inverse(const float * matrix, float * result)
{
    auto * B = new float[N * N];
    auto * I = new float[N * N];
    auto * tmp = new float[N * N];
    auto * R = new float[N * N];
    bool flag = true;

    FillB(matrix, B);
    FillI(I);
    Multiplication(B, matrix, tmp);
    Subtraction(I, tmp, R);
    Addition(I, R, tmp);
    Copy(result, R);

    for (int i = 2; i < M; ++i)
    {
        Multiplication(flag ? result : I, R, flag ? I : result);
        Addition(tmp, flag ? I : result, tmp);
        flag = !flag;
    }

    Multiplication(tmp, B, result);

    delete[] I;
    delete[] B;
    delete[] tmp;
    delete[] R;
}

float GetMaxSum(const float * matrix)
{
    float max_sum_row = FLT_MIN;
    float max_sum_column = FLT_MIN;
    float sum_row = 0;
    float sum_column = 0;

    for (int i = 0; i < N; i++) // rows
    {
        sum_row = 0;
        sum_column = 0;

        for (int j = 0; j < N; j++) // columns
        {
            sum_row += std::fabs(matrix[N * i + j]);
            sum_column += std::fabs(matrix[j * N + i]);
        }
```

```
        if (sum_row > max_sum_row) max_sum_row = sum_row;
        if (sum_column > max_sum_column) max_sum_column = sum_column;
    }

    return max_sum_row * max_sum_column;
}

void FillB(const float * matrix, float * B)
{
    float max = GetMaxSum(matrix);

    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            B[N * i + j] = matrix[j * N + i] / max;
}

void FillI(float * I)
{
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            I[N * i + j] = (float)(i == j);
}

void Multiplication(const float * multiplier1,
                    const float * multiplier2, float * result)
{
    for (int i = 0; i < N * N; ++i)
        result[i] = 0;

    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
                result[N * i + k] += multiplier1[N * i + j] *
                                     multiplier2[N * j + k];
}

void Addition(const float * addend1, const float * addend2,
              float * result)
{
    for (int i = 0; i < N * N; ++i)
        result[i] = addend1[i] + addend2[i];
}

void Subtraction(const float * minuend, const float * subtrahend,
                 float * result)
{
    for (int i = 0; i < N * N; i++)
        result[i] = minuend[i] - subtrahend[i];
}

void Copy(float * dest, const float * src)
{
    for (int i = 0; i < N * N; i++)
        dest[i] = src[i];
}
```

```
//void Print(const float * matrix)
//{
//    for (int i = 0; i < N; i++)
//    {
//        for (int j = 0; j < N; j++)
//            std::cout << matrix[N * i + j] << " ";
//
//        std::cout << std::endl;
//    }
//}
```

src/manual.cpp

```
#include <cfloat>      // FLOAT_MIN
#include <ctime>
#include <iostream>
#include <xmmintrin.h>

#define N 2048
#define M 10

void Inverse(const float * matrix, float * result);
float GetMaxSum(const float * matrix); // |A|_1 * |A|_infinity
void FillB(const float * matrix, float * B);
void FillI(float * I);
void Multiplication(const float * multiplier1,
                    const float * multiplier2, float * result);
void Addition(const float * addend1, const float * addend2,
              float * result);
void Subtraction(const float * minuend, const float * subtrahend,
                 float * result);
void Copy(float * dest, const float * src);
//void Print(const float * matrix);

int main()
{
    srandom(time(nullptr));
    auto * matrix = new float [N * N];
    auto * result = new float [N * N];
    timespec start = { 0, 0 };
    timespec end = { 0, 0 };

    for (int i = 0; i < N * N; ++i)
    {
        matrix[i] = float(random());
        matrix[i] *= (random() % 2) ? 1 : -1;
        result[i] = 0;
    }

//    float matrix[N * N] = {
//        3, 0, 9, 0,
//        3, 4, 8, 1,
//        2, 4, 6, 2,
//        6, 2, 2, 6
//    };
```

```
//    float result[N * N] = { 0 };

    clock_gettime(CLOCK_MONOTONIC_RAW, &start);
    Inverse(matrix, result);
    clock_gettime(CLOCK_MONOTONIC_RAW, &end);

//    Print(matrix);
//    cout << endl;
//    Print(result);
//    cout << endl;

    std::cout << "Time with manual vectorization: "
         << (double)end.tv_sec - (double)start.tv_sec + 1e-9 *
            ((double)end.tv_nsec - (double)start.tv_nsec)
         << " sec." << std::endl;

    return EXIT_SUCCESS;
}

void Inverse(const float * matrix, float * result)
{
    auto * B = new float[N * N];
    auto * I = new float[N * N];
    auto * tmp = new float[N * N];
    auto * R = new float[N * N];
    bool flag = true;

    FillB(matrix, B);
    FillI(I);
    Multiplication(B, matrix, tmp);
    Subtraction(I, tmp, R);
    Addition(I, R, tmp);
    Copy(result, R);

    for (int i = 2; i < M; ++i)
    {
        Multiplication(flag ? result : I, R, flag ? I : result);
        Addition(tmp, flag ? I : result, tmp);
        flag = !flag;
    }

    Multiplication(tmp, B, result);

    delete[] I;
    delete[] B;
    delete[] tmp;
    delete[] R;
}

inline __m128 mm_abs_ps(__m128 A)
{
    const __m128 SIGNMASK =
    _mm_castsi128_ps(_mm_set1_epi32(int(0x80000000)));
    return _mm_andnot_ps(SIGNMASK, A);
}
```

```
float GetMaxSum(const float * matrix)
{
    float max_sum_row = FLT_MIN;
    float max_sum_column = FLT_MIN;
    __m128 m128_row;
    __m128 m128_column;
    __m128 m128_sum_row;
    __m128 m128_sum_column;
    float m32_sum[4];
    float sum_row;
    float sum_column;

    for (int i = 0; i < N; ++i)
    {
        m128_sum_row = _mm_setzero_ps();
        m128_sum_column = _mm_setzero_ps();

        for (int j = 0; j < N / 4; ++j)
        {
            m128_row = _mm_load_ps(matrix + N * i + 4 * j);
            m128_column = _mm_setr_ps(matrix[N * j + i],
                                      matrix[N * (j + 1) + i],
                                      matrix[N * (j + 2) + i],
                                      matrix[N * (j + 3) + i]);
            m128_row = mm_abs_ps(m128_row);
            m128_column = mm_abs_ps(m128_column);

            m128_sum_row = _mm_add_ps(m128_sum_row, m128_row);
            m128_sum_column = _mm_add_ps(m128_sum_column,
                                         m128_column);
        }

        _mm_store_ps(m32_sum, m128_sum_row);
        sum_row = m32_sum[0] + m32_sum[1] + m32_sum[2] + m32_sum[3];

        _mm_store_ps(m32_sum, m128_sum_column);
        sum_column = m32_sum[0] + m32_sum[1] + m32_sum[2] +
                     m32_sum[3];

        if (sum_row > max_sum_row) max_sum_row = sum_row;
        if (sum_column > max_sum_column) max_sum_column = sum_column;
    }

    return max_sum_row * max_sum_column;
}

void FillB(const float * matrix, float * B)
{
    float max = GetMaxSum(matrix);
    __m128 m128_max = _mm_set1_ps(max);
    __m128 * m128_B;
    m128_B = (__m128 *)B;
    __m128 m128_matrix_column;

    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N / 4; ++j)
```

```
        {
            m128_matrix_column = _mm_setr_ps(matrix[N * j + i],
                                             matrix[N * (j + 1) + i],
                                             matrix[N * (j + 2) + i],
                                             matrix[N * (j + 3) + i]);

            m128_B[N * i / 4 + j] = _mm_div_ps(m128_matrix_column,
                                               m128_max);
        }
}

void FillI(float * I)
{
    __m128 * m128_I;
    m128_I = (__m128 *)I;

    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N / 4; ++j)
            m128_I[N * i / 4 + j] = _mm_setr_ps(i == j,
                                                i == (j + 1),
                                                i == (j + 2),
                                                i == (j + 3));
}

void Multiplication(const float * multiplier1,
                    const float * multiplier2, float * result)
{
    __m128 * m128_result;
    const __m128 * m128_multiplier2;
    m128_result = (__m128 *)result;
    m128_multiplier2 = (const __m128 *)multiplier2;
    __m128 m128_multiplier1;
    __m128 tmp;

    for (int i = 0; i < N * N / 4; ++i)
        m128_result[i] = _mm_setzero_ps();

    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
        {
            m128_multiplier1 = _mm_set1_ps(multiplier1[N * i + j]);
            for (int k = 0; k < N / 4; ++k)
            {
                tmp = _mm_mul_ps(m128_multiplier1,
                                 m128_multiplier2[N * j / 4 + k]);
                m128_result[N * i / 4 + k] =
                _mm_add_ps(m128_result[N * i / 4 + k], tmp);
            }
        }
}

void Addition(const float * addend1, const float * addend2,
              float * result)
{
    const __m128 * m128_addend1;
    const __m128 * m128_addend2;
```

```
    __m128 * m128_result;
    m128_addend1 = (const __m128 *)addend1;
    m128_addend2 = (const __m128 *)addend2;
    m128_result = (__m128 *)result;

    for (int i = 0; i < N * N / 4; i++)
        m128_result[i] = _mm_add_ps(m128_addend1[i], m128_addend2[i]);
}

void Subtraction(const float * minuend, const float * subtrahend,
                 float * result)
{
    const __m128 * m128_minuend;
    const __m128 * m128_subtrahend;
    __m128 * m128_result;
    m128_minuend = (const __m128 *)minuend;
    m128_subtrahend = (const __m128 *)subtrahend;
    m128_result = (__m128 *)result;

    for (int i = 0; i < N * N / 4; i++)
        m128_result[i] = _mm_sub_ps(m128_minuend[i],
                                    m128_subtrahend[i]);
}

void Copy(float * dest, const float * src)
{
    for (int i = 0; i < N * N; i++)
        dest[i] = src[i];
}

//void Print(const float * matrix)
//{
//    for (int i = 0; i < N; i++)
//    {
//        for (int j = 0; j < N; j++)
//            std::cout << matrix[N * i + j] << " ";
//
//        std::cout << std::endl;
//    }
//}
```

src/BLAS.cpp

```
#include <cfloat>       // FLT_MIN
#include <cmath>        // fabs()
#include <ctime>
#include <iostream>
#include <mkl_cblas.h>  // cblas_sgemm

#define N 2048
#define M 10

void Inverse(const float * matrix, float * result);
float GetMaxSum(const float * matrix); // |A|_1 * |A|_infinity
void FillB(const float * matrix, float * B);
```

```
void FillI(float * I);
void Multiplication(const float * multiplier1,
                    const float * multiplier2, float * result);
void Addition(const float * addend1, const float * addend2,
              float * result);
void Subtraction(const float * minuend, const float * subtrahend,
                 float * result);
void Copy(float * dest, const float * src);
//void Print(const float * matrix);

int main()
{
    srandom(time(nullptr));
    auto * matrix = new float [N * N];
    auto * result = new float [N * N];
    timespec start = { 0, 0 };
    timespec end = { 0, 0 };

    for (int i = 0; i < N * N; ++i)
    {
        matrix[i] = float(random());
        matrix[i] *= (random() % 2) ? 1 : -1;
        result[i] = 0;
    }

//    float matrix[N * N] = {
//        3, 0, 9, 0,
//        3, 4, 8, 1,
//        2, 4, 6, 2,
//        6, 2, 2, 6
//    };
//    float result[N * N] = { 0 };


    clock_gettime(CLOCK_MONOTONIC_RAW, &start);
    Inverse(matrix, result);
    clock_gettime(CLOCK_MONOTONIC_RAW, &end);

//    Print(matrix);
//    cout << endl;
//    Print(result);
//    cout << endl;

    std::cout << "Time with BLAS: "
         << (double)end.tv_sec - (double)start.tv_sec + 1e-9 *
            ((double)end.tv_nsec - (double)start.tv_nsec)
         << " sec." << std::endl;

    return EXIT_SUCCESS;
}

void Inverse(const float * matrix, float * result)
{


    auto * B = new float[N * N];
```

```
    auto * I = new float[N * N];
    auto * tmp = new float[N * N];
    auto * R = new float[N * N];
    bool flag = true;

    FillB(matrix, B);
    FillI(I);
    Multiplication(B, matrix, tmp);
    Subtraction(I, tmp, R);

    Addition(I, R, tmp);
    Copy(result, R);

    for (int i = 2; i < M; ++i)
    {
        Multiplication(flag ? result : I, R, flag ? I : result);
        Addition(tmp, flag ? I : result, tmp);
        flag = !flag;
    }

    Multiplication(tmp, B, result);

    delete[] I;
    delete[] B;
    delete[] tmp;
    delete[] R;
}

float GetMaxSum(const float * matrix)
{
    float max_sum_row = FLT_MIN;
    float max_sum_column = FLT_MIN;
    float sum_row = 0;
    float sum_column = 0;

    for (int i = 0; i < N; i++) // rows
    {
        sum_row = 0;
        sum_column = 0;

        for (int j = 0; j < N; j++) // columns
        {
            sum_row += std::fabs(matrix[N * i + j]);
            sum_column += std::fabs(matrix[j * N + i]);
        }

        if (sum_row > max_sum_row) max_sum_row = sum_row;
        if (sum_column > max_sum_column) max_sum_column = sum_column;
    }

    return max_sum_row * max_sum_column;
}

void FillB(const float * matrix, float * B)
{
    float max = GetMaxSum(matrix);
```

```
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            B[N * i + j] = matrix[j * N + i] / max;
}

void FillI(float * I)
{
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            I[N * i + j] = (float)(i == j);
}

void Multiplication(const float * multiplier1,
                    const float * multiplier2, float * result)
{
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N,
                N, 1.0, multiplier1, N, multiplier2, N, 0.0,
                result, N);
}

void Addition(const float * addend1, const float * addend2,
              float * result)
{
    for (int i = 0; i < N * N; ++i)
        result[i] = addend1[i] + addend2[i];
}

void Subtraction(const float * minuend, const float * subtrahend,
                 float * result)
{
    for (int i = 0; i < N * N; i++)
        result[i] = minuend[i] - subtrahend[i];
}

void Copy(float * dest, const float * src)
{
    for (int i = 0; i < N * N; i++)
        dest[i] = src[i];
}

//void Print(const float * matrix)
//{
//    for (int i = 0; i < N; i++)
//    {
//        for (int j = 0; j < N; j++)
//            std::cout << matrix[N * i + j] << " ";
//
//        std::cout << std::endl;
//    }
//}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.16.3)
project(lab7)
```

```
add_executable(default
        src/default.cpp)

add_executable(manual
        src/manual.cpp)
target_compile_options(manual PUBLIC
        -msse3)

add_executable(BLAS
        src/BLAS.cpp)
find_package(MKL CONFIG REQUIRED)
target_compile_options(BLAS PUBLIC
        $<TARGET_PROPERTY:MKL::MKL,INTERFACE_COMPILE_OPTIONS>)
target_include_directories(BLAS PUBLIC
        $<TARGET_PROPERTY:MKL::MKL,INTERFACE_INCLUDE_DIRECTORIES>)
target_link_libraries(BLAS PUBLIC
        $<LINK_ONLY:MKL::MKL>)
```