МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра вычислительной техники

ЛАБОРАТОРНАЯ РАБОТА №2

по дисциплине «Параллельные алгоритмы и системы»

ТЕМА: УМНОЖЕНИЕ МАТРИЦЫ НА ВЕКТОР

Студент гр. 9308 _____ Семенов А. И.

Преподаватель _____ Пазников А. А.

Санкт-Петербург

2023

Цель работы

Реализация и оптимизация программы для умножения матрицы на вектор.

Задание

Реализовать алгоритм для умножения матрицы на вектор и оптимизировать работу программы по следующим метрикам: время выполнения и количество инструкций.

Варианты применённых оптимизаций кода

Ниже представлен список применённыз оптимизаций кода:

1. Отказ от функции
2. Расспараллеливание
3. Превращение 2d массива (матрицы) в 1d
4. Отказ от функции + расспараллеливание
5. OpenMP SIMD
6. Векторизация MAVX2

Количество просчетов: 1000

Кол-во строк матрицы: 700

Кол-во столбцов матрицы, оно же строки вектора: 500

Исходная реализация имеет следующие показатели:

Компилятор / Метрика	g++	clang++
Время выполнения, с	0,664025	0,637477
Кол-во инструкций	12 426 224 058	7 477 459 452

Отказ от функции

В исходном коде программы, представленного в приложении А, был проведен перенос тела функции умножении матрицы на вектор в *main*.

Исходный блок:

```
for(int i = 0; i < MEASURE_NUM; ++i)
  multiply_matrix_by_vector(matrix, M_ROWS, M_COLUMNS_V_ROWS, vector);
```

Измененный блок:

```
for(int i = 0; i < MEASURE_NUM; ++i)
{
  double* m_result = new double[M_ROWS];
  for(int i = 0; i < M_ROWS; ++i)
  {
      for(int j = 0; j < M_COLUMNS_V_ROWS; ++j)
          m_result[i] += matrix[i][j] * vector[j];
  }
}
```

Подобное изменение снизило количество инструкций и немного ускорило работу программу для компилятора g++ и никак не повлияло на clang++.

Компилятор / Метрика	g++	clang++
Исходная программа		
Время выполнения, с	0,664025	0,637477
Кол-во инструкций	12 426 224 058	7 477 459 452
Текущий результат		
Время выполнения, с	0,649109	0,637468
Кол-во инструкций	12 071 488 924	7 122 339 459

Расспараллеливание

Использование директивы OpenMP для расспараллеливания вычислений:

Исходный блок в функции умножения матрицы на вектор:

```
for(int i = 0; i < rows; ++i)
{
  for(int j = 0; j < columns; ++j)
      m_result[i] += matrix[i][j] * vector[j];
}
```

Измененный:

```
#pragma omp parallel for num_threads(THREAD_NUM)
for(int i = 0; i < rows; ++i)
{
  for(int j = 0; j < columns; ++j)
      m_result[i] += matrix[i][j] * vector[j];
}
```

Такой подход, как и ожидалось, привел к значетельному ускорению работы программы, однако повлек за собой увеличение количества инструкций (наиболее выражено у компилятора clang++):

Компилятор / Метрика	g++	clang++
Исходная программа		
Время выполнения, с	0,664025	0,637477
Кол-во инструкций	12 426 224 058	7 477 459 452
Прошлый результат		
Время выполнения, с	0,649109	0,637468
Кол-во инструкций	12 071 488 924	7 122 339 459
Текущий результат		
Время выполнения, с	0,120439	0,120782
Кол-во инструкций	12 806 544 442	9 085 993 769

Представление матрицы как одномерный массив

Было решено проверить, как повлияет на работу представление матрицы не как двумерного массива, а как одномерного.

Исходный блок:

```
#pragma omp parallel for num_threads(THREAD_NUM)
for(int i = 0; i < rows; ++i)
{
  for(int j = 0; j < columns; ++j)
      m_result[i] += matrix[i][j] * vector[j];
}
```

Измененный блок:

```
#pragma omp parallel for num_threads(THREAD_NUM)
for(int i = 0; i < rows; ++i)
{
  for(int j = 0; j < columns; ++j)
      m_result[i] += matrix[i*columns + j] * vector[j];
}
```

Перевод матрицы в одномерный массив привел к уменьшению инструкций для компилятора g++, однако увеличил их число для clang++. Скорость выполнения работы программы для g++ не изменился, а для clang++ слегка увеличился, что можно объяснить возможной помехой.

Компилятор / Метрика	g++	clang++
Исходная программа		
Время выполнения, с	0,664025	0,637477
Кол-во инструкций	12 426 224 058	7 477 459 452
Прошлый результат		
Время выполнения, с	0,120439	0,120782
Кол-во инструкций	12 806 544 442	9 085 993 769
Текущий результат		
Время выполнения, с	0,120736	0,125856
Кол-во инструкций	12 451 757 627	9 896 290 372

Отказ от функции и расспараллеливание

Вариант, используемый при простом расспараллеливании, дополнился переносом тела функции в *main*.

Исходный блок:

```
for(int i = 0; i < MEASURE_NUM; ++i)
      multiply_matrix_by_vector(matrix, M_ROWS, M_COLUMNS_V_ROWS, vector);
```

Измененный блок:

```
for(int i = 0; i < MEASURE_NUM; ++i)
{
  double* m_result = new double[M_ROWS];
  #pragma omp parallel for num_threads(THREAD_NUM)
  for(int i = 0; i < M_ROWS; ++i)
  {
      for(int j = 0; j < M_COLUMNS_V_ROWS; ++j)
          m_result[i] += matrix[i][j] * vector[j];
  }
}
```

Данная реализация положительно сказать как на скорости работы, так и на количестве инструкций.

Компилятор / Метрика	g++	clang++
Исходная программа		
Время выполнения, с	0,664025	0,637477
Кол-во инструкций	12 426 224 058	7 477 459 452
Прошлый результат		
Время выполнения, с	0,120736	0,125856
Кол-во инструкций	12 451 757 627	9 896 290 372
Текущий результат		
Время выполнения, с	0,11789	0,115037
Кол-во инструкций	12 448 157 725	8 405 152 802

OpenMP SIMD

Была использована директива OpenMP для создания векторизации в цикле:

```
#pragma omp simd
```

Использование данной директивы привело к улучшению скорости выполнения для компилятора g++, но отрицательно сказалась на программе, скомпилируемом clang++, где обе метрики ухудшились.

Компилятор / Метрика	g++	clang++
Исходная программа		
Время выполнения, с	0,664025	0,637477
Кол-во инструкций	12 426 224 058	7 477 459 452
Прошлый результат		
Время выполнения, с	0,11789	0,115037
Кол-во инструкций	12 448 157 725	8 405 152 802
Текущий результат		
Время выполнения, с	0,109875	0,123346
Кол-во инструкций	12 451 782 020	11 256 350 614

Векторизация MAVX2

Была использована опция -mavx2 для компилятора, позволяя векторизировать цикл:

Вариант OpenMP SIMD:

```
#pragma omp parallel for num_threads(THREAD_NUM)
for(int i = 0; i < rows; ++i)
{
  #pragma omp simd
  for(int j = 0; j < columns; ++j)
      m_result[i] += matrix[i][j] * vector[j];
}
```

Использование MAVX2:

```
#pragma omp parallel for num_threads(THREAD_NUM)
for(int i = 0; i < rows; ++i)
{
  __m256d t1, t2;
  for(int j = 0; j < columns-5; j += 4)
  {
      t1 = _mm256_loadu_pd(&matrix[i][j]);
      t2 = _mm256_loadu_pd(&vector[j]);
      t2 = _mm256_mul_pd(t1, t2);
      m_result[i] += t2[0] + t2[1] + t2[2] + t2[3];
  }
  for(int j = std::max(columns-5, 0); j < columns; ++j)
      m_result[i] += matrix[i][j] * vector[j];
}
```

Данный подход наилучшим образом отразился на обоих метриках: время сократилось на треть, а кол-во инструкции более чем в 2 раза.

Компилятор / Метрика	g++	clang++
Исходная программа		
Время выполнения, с	0,664025	0,637477
Кол-во инструкций	12 426 224 058	7 477 459 452
Прошлый результат		
Время выполнения, с	0,109875	0,123346
Кол-во инструкций	12 451 782 020	11 256 350 614
Текущий результат		
Время выполнения, с	0,060152	0,082163
Кол-во инструкций	5 158 741 244	5 376 077 094

Полученные результаты

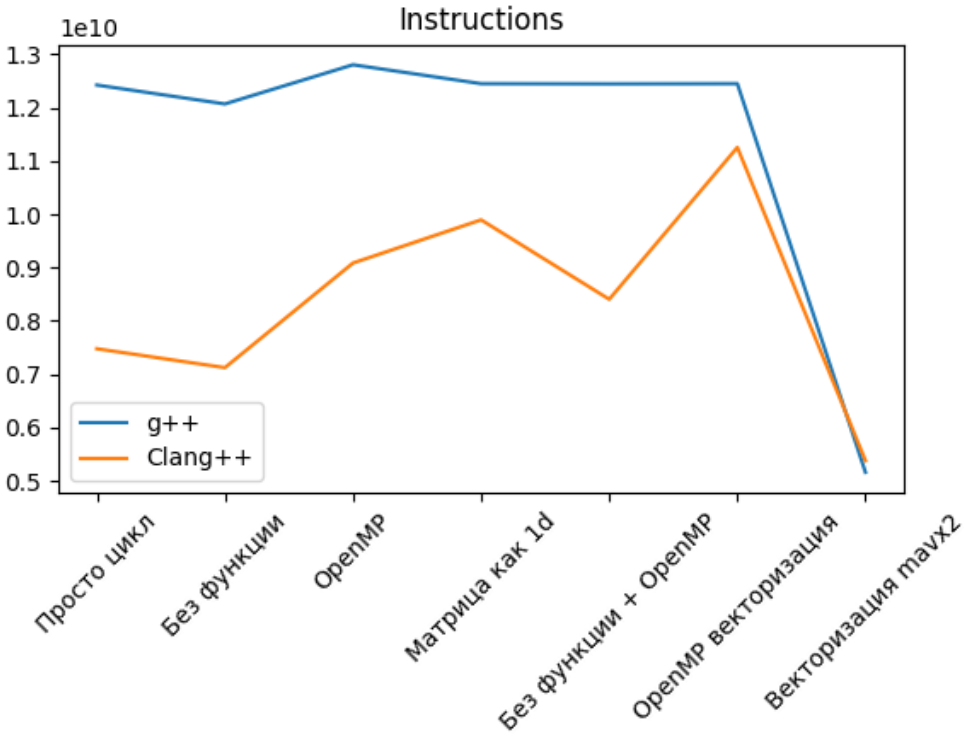

Рисунок 1. Кол-во инстукций для реализаций

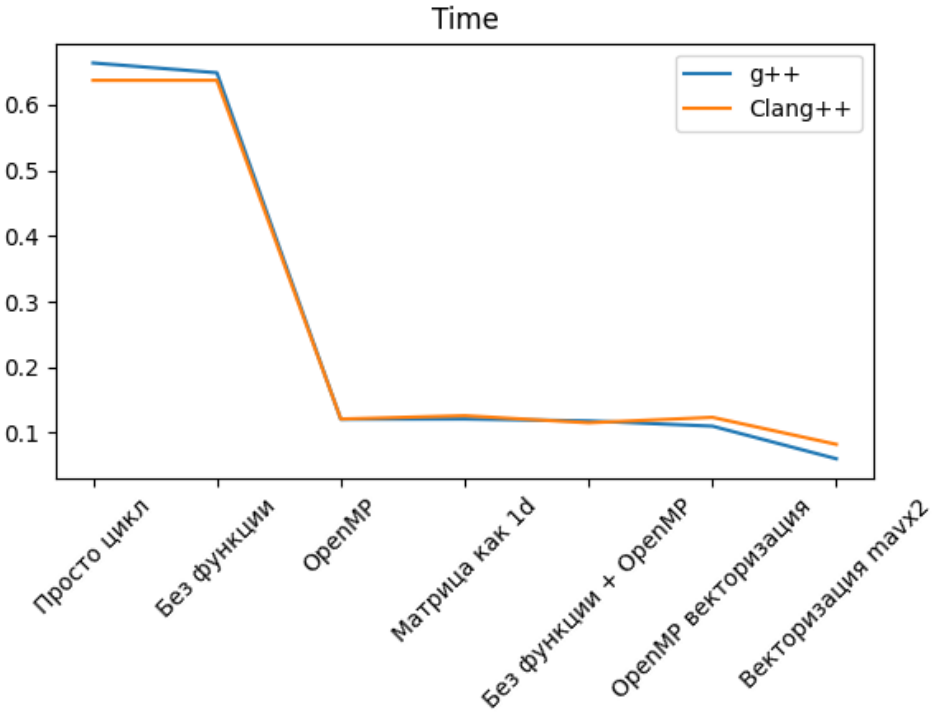

Рисунок 2. Время выполнения в секундах для реализаций

Итоговая таблица:

Компилятор / Метрика	g++	clang++
Исходная программа		
Время выполнения, с	0,664025	0,637477
Кол-во инструкций	12 426 224 058	7 477 459 452
Отказ от функции		
Время выполнения, с	0,649109	0,637468
Кол-во инструкций	12 071 488 924	7 122 339 459
Расспараллеливание работы OpenMP		
Время выполнения, с	0,120439	0,120782
Кол-во инструкций	12 806 544 442	9 085 993 769
Представление матрицы как одномерный массив		
Время выполнения, с	0,120736	0,125856
Кол-во инструкций	12 451 757 627	9 896 290 372
Отказ от функции + OpenMP		
Время выполнения, с	0,11789	0,115037
Кол-во инструкций	12 448 157 725	8 405 152 802
OpenMP SIMD		
Время выполнения, с	0,109875	0,123346
Кол-во инструкций	12 451 782 020	11 256 350 614
Векторизация MAVX2		
Время выполнения, с	0,060152	0,082163
Кол-во инструкций	5 158 741 244	5 376 077 094

Вывод

Был реализован алгоритм для умножения матрицы на вектор с последующими этапами оптимизации:

– перенос тела функции с целью ухода от ее вызова

Такой подход привел к незначительному по сравнению с исходной программы ускорении времени работы программы, но позволил сократить кол-во инструкций, выполняемых процессором.

– расспараллеливание работы

Используя все доступные потоки для процессора, удалось достичь ускорения в 5 раз. Однако использование стандарта OpenMP в общей сложности увеличило количество инструкций процессора. Наибольший прирост оказался у программы, скомпилированной с помощью clang++.

– представление матрицы в виде одномерного массива

Такое представление избавляет от вычисления адреса для случая двумерного массива, но в целом на скорость работы программы не повлияло. Однако у компилятора clang++ наблюдается ухудшение показателей метрик, когда у версии программы g++ скорость осталась на том же уровне, а количество инструкций было уменьшено.

– перенос тела функции с целью ухода от ее вызова при использовании OpenMP.

В случае расспараллеленой программы эффект отказ от функции виден лучше: уменьшение скорости работы программы, а так же сокращение количества инструкций (для компилятора clang++ на 1/9 от всего числа).

– векторизация OpenMP: simd

Использование векторизации в стандарте OpenMP позволило ускорить работу программы для случая компилятора g++. На работу программы после компиляции через clang++ такой подход не дал никаких улучшений, метрики лишь ухудшились.

– Векторизация MAVX2

Использование данного расширения систем команд позволяет более явно работать с векторами, а также наилучшим образом влияет на скорость работы программы и кол-во инструкций процессора: ускорение на треть и сокращение инструкций в 2 раза по сравнению с использованием OpenMP simd.

По итогу всех оптимизаций удалось сократить время работы в 11 раз. Количество инструкций было сокращено в 2.4 раза для компилятора g++ и 1.4 раза для компилятора clang++.

Приложение А. Исходный код

```
#include <iostream>
#include <chrono>

#define WARMUP_NUM 10
#define MEASURE_NUM 1000

#define M_ROWS 700
#define M_COLUMNS_V_ROWS 500

#define RAND_LEFT -1000
#define RAND_RIGHT 1000

double* multiply_matrix_by_vector(double** matrix, int rows, int columns,
double* vector);

int randomInt(int left, int right)
{
    return rand() % (right - left + 1) + left;
}

void output_matrix(double** matrix, int rows, int columns)
{
    for(int i = 0; i < rows; ++i)
    {
        for(int j = 0; j < columns; ++j)
        {
            std::cout << matrix[i][j] << " ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

void output_vector(double* vector, int columns)
{
    for(int j = 0; j < columns; ++j)
    {
        std::cout << vector[j] << " ";
    }
    std::cout << std::endl;
    std::cout << std::endl;
}

int main()
{
    std::srand(61771);
    double** matrix = new double*[M_ROWS];
    double* vector = new double[M_COLUMNS_V_ROWS];

    for(int i = 0; i < M_ROWS; ++i)
    {
        matrix[i] = new double[M_COLUMNS_V_ROWS];
        for(int j = 0; j < M_COLUMNS_V_ROWS; ++j)
        {
            matrix[i][j] = (double)randomInt(RAND_LEFT, RAND_RIGHT);
        }
    }

    for(int i = 0; i < M_COLUMNS_V_ROWS; ++i)
    {
        vector[i] = (double)randomInt(RAND_LEFT, RAND_RIGHT);
```

```
    }
    for(int i = 0; i < WARMUP_NUM; ++i)
        multiply_matrix_by_vector(matrix, M_ROWS, M_COLUMNS_V_ROWS, vector);

    auto start = std::chrono::high_resolution_clock::now();
    for(int i = 0; i < MEASURE_NUM; ++i)
        multiply_matrix_by_vector(matrix, M_ROWS, M_COLUMNS_V_ROWS, vector);
    auto stop = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop -
start);

    std::cout << "Time taken by function: "
         << duration.count()/1e6 << " seconds" << std::endl;

    for(int i = 0; i < M_ROWS; ++i)
        delete matrix[i];

    delete [] matrix;
    delete [] vector;
    return 0;
}

double* multiply_matrix_by_vector(double** matrix, int rows, int columns,
double* vector)
{
    double* m_result = new double[M_ROWS];
    for(int i = 0; i < rows; ++i)
    {
        for(int j = 0; j < columns; ++j)
            m_result[i] += matrix[i][j] * vector[j];
    }

    return m_result;
}
```