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

ЛАБОРАТОРНАЯ РАБОТА №1

по дисциплине «Параллельные алгоритмы и системы»

ТЕМА: РЕШЕНИЕ СЛАУ МЕТОДОМ ГАУССА

Студент гр. 9308 _____ Семенов А. И.

Преподаватель _____ Пазников А. А.

Санкт-Петербург

2023

Цель работы

Реализация и оптимизация программы для решения СЛАУ методом Гаусса.

Задание

Реализовать алгоритм для решения систему линейных алгебраических уравнений (СЛАУ) методом Гаусса и оптимизировать работу программы по следующим метрикам: время выполнения и количество инструкций.

Варианты применённых оптимизаций кода

Ниже представлен список применённыз оптимизаций кода:

1. Использование встроенной функции __builtin_expect, указывающая компилятору наиболее вероятное значение.
2. Раскручивание цикла: изменение сразу нескольких значений в массиве.
3. Расспараллеливание работы.
4. Уход от функции: тело функции встроить в тело выполнения алгоритма.
5. Векторизация

Исходная реализация имеет следующие показатели:

Компилятор / Метрика	g++	clang++
Время выполнения, с	13,3109	13,5008
Кол-во инструкций	326 305 306 192	311 897 300 646

Встроенная функция __builtin_expect

Исходный код программы, представленный в приложении А, претерпевает в проверке наличие нуля на текущей итерации цикла (проход по всем строкам сверху вниз) изменение в виде добавления встроенной функции __builtin_expect со значением 0, т.е. в данном условии ожидается, что оно, скорее всего, не будет выполнено.

Исходная строка: `if(solution[row][column] == 0.0)`

Измененная строка: `if(unlikely(solution[row][column] == 0.0))`

Подобное изменение повлекло за собой увеличение количества инструкций и времени выполнения на менее чем 1% для компиляции через g++, и уменьшение времени выполнения на менее чем 1% для компиляции через clang++.

Компилятор / Метрика	g++	clang++
Исходная программа		
Время выполнения, с	13,3109	13,5008
Кол-во инструкций	326 305 306 192	311 897 300 646
Текущий результат		
Время выполнения, с	13,3613	13,4933
Кол-во инструкций	326 308 389 760	311 901 475 696

Раскручивание цикла

Раскрутка цикла в виде прохода сразу по 8 элементов за итерацию:

Исходный блок:

```
for(int i = 0; i < vectorSize; ++i)
  vector1[i] = vector1[i] - vector2[i] * multiplyBy;
```

Замена:

```
int i;
for(i = 0; i < vectorSize-8; i += 8)
{
  vector1[i] = vector1[i] - vector2[i] * multiplyBy;
  vector1[i+1] = vector1[i+1] - vector2[i+1] * multiplyBy;
  vector1[i+2] = vector1[i+2] - vector2[i+2] * multiplyBy;
  vector1[i+3] = vector1[i+3] - vector2[i+3] * multiplyBy;
  vector1[i+4] = vector1[i+4] - vector2[i+4] * multiplyBy;
  vector1[i+5] = vector1[i+5] - vector2[i+5] * multiplyBy;
  vector1[i+6] = vector1[i+6] - vector2[i+6] * multiplyBy;
  vector1[i+7] = vector1[i+7] - vector2[i+7] * multiplyBy;
}
for(;i < vectorSize; ++i)
  vector1[i] = vector1[i] - vector2[i] * multiplyBy;
```

Данный вариант привел к ускорению работы программы, а также снижению кол-ва инструкций:

Компилятор / Метрика	g++	clang++
Исходная программа		
Время выполнения, с	13,3109	13,5008
Кол-во инструкций	326 305 306 192	311 897 300 646
Прошлый результат		
Время выполнения, с	13,3613	13,4933
Кол-во инструкций	326 308 389 760	311 901 475 696
Текущий результат		
Время выполнения, с	12,4306	12,7807
Кол-во инструкций	316 425 869 689	303 716 120 353

Расспараллеливание работы

Расспараллеливание прошлого варианта программы (раскручивание цикла) произведено с помощью использования директивы стандарта OpenMP:

```
#pragma omp parallel for num_threads(THREAD_NUM)
```

где число потоков равно 12.

Расспараллеливание происходит для цикла, относящегося к вычитании текущей строки из всех, что ниже нее, домножая на необходимое число:

```
#pragma omp parallel for num_threads(THREAD_NUM)
        for(int j = row+1; j < rows; ++j) // проходим по всем строкам ниже
            sub_vector_from_vector2(solution[j], solution[row], columns,
                                    (solution[j][column]/solution[row][column]));
```

Данная реализация повышает кол-во инструкций (даже в сравнении с оригинальной программой), но значительно ускоряет выполнение программы.

Компилятор / Метрика	g++	clang++
Исходная программа		
Время выполнения, с	13,3109	13,5008
Кол-во инструкций	326 305 306 192	311 897 300 646
Прошлый результат		
Время выполнения, с	12,4306	12,7807
Кол-во инструкций	316 425 869 689	303 716 120 353
Текущий результат		
Время выполнения, с	2,55635	2,66892
Кол-во инструкций	334 392 019 305	324 272 625 344

Уход от функции

Было внедрено тело функции sub_vector_from_vector2 в тело функции solveGauss непосредственно.

Исходный блок:

```
#pragma omp parallel for num_threads(THREAD_NUM)
        for(int j = row+1; j < rows; ++j) // проходим по всем строкам ниже
            sub_vector_from_vector2(solution[j], solution[row], columns,
(solution[j][column]/solution[row][column]));
        ++row;
```

Замена:

```
#pragma omp parallel for num_threads(THREAD_NUM)
        for(int j = row+1; j < rows; ++j) // проходим по всем строкам ниже
        {
            double *irow = solution[row],
                   *jrow = solution[j];
            double multiplyBy = (jrow[column]/irow[column]);
            int i;
            for(i = 0; i < columns-8; i += 8)
            {
                jrow[i] -= irow[i] * multiplyBy;
                jrow[i+1] -= irow[i+1] * multiplyBy;
                jrow[i+2] -= irow[i+2] * multiplyBy;
                jrow[i+3] -= irow[i+3] * multiplyBy;
                jrow[i+4] -= irow[i+4] * multiplyBy;
                jrow[i+5] -= irow[i+5] * multiplyBy;
                jrow[i+6] -= irow[i+6] * multiplyBy;
                jrow[i+7] -= irow[i+7] * multiplyBy;
            }
            for(;i < columns; ++i)
                jrow[i] -= irow[i] * multiplyBy;
        }

        ++row;
```

Данный подход позволил еще ускорить работу программы, а также снизить количество инструкций (особенно хорошо заметно при использовании компилятора clang++).

Компилятор / Метрика	g++	clang++
Исходная программа		
Время выполнения, с	13,3109	13,5008
Кол-во инструкций	326 305 306 192	311 897 300 646
Прошлый результат		
Время выполнения, с	2,55635	2,66892
Кол-во инструкций	334 392 019 305	324 272 625 344
Текущий результат		
Время выполнения, с	2,27947	2,15132
Кол-во инструкций	318 478 054 980	267 462 921 635

Векторизация

Была использована директива OpenMP для создания векторизации в цикле:

```
#pragma omp simd
```

Использование данной директивы привело, в общем случае, к ухудшению: время выполнения немного увеличилось, кол-во инструкций у clang++ возросло. Однако для g++ кол-во инструкций было уменьшено.

Компилятор / Метрика	g++	clang++
Исходная программа		
Время выполнения, с	13,3109	13,5008
Кол-во инструкций	326 305 306 192	311 897 300 646
Прошлый результат		
Время выполнения, с	2,27947	2,15132
Кол-во инструкций	318 478 054 980	267 462 921 635
Текущий результат		
Время выполнения, с	2,28031	2,17637
Кол-во инструкций	315 278 568 877	275 148 538 411

Полученные результаты

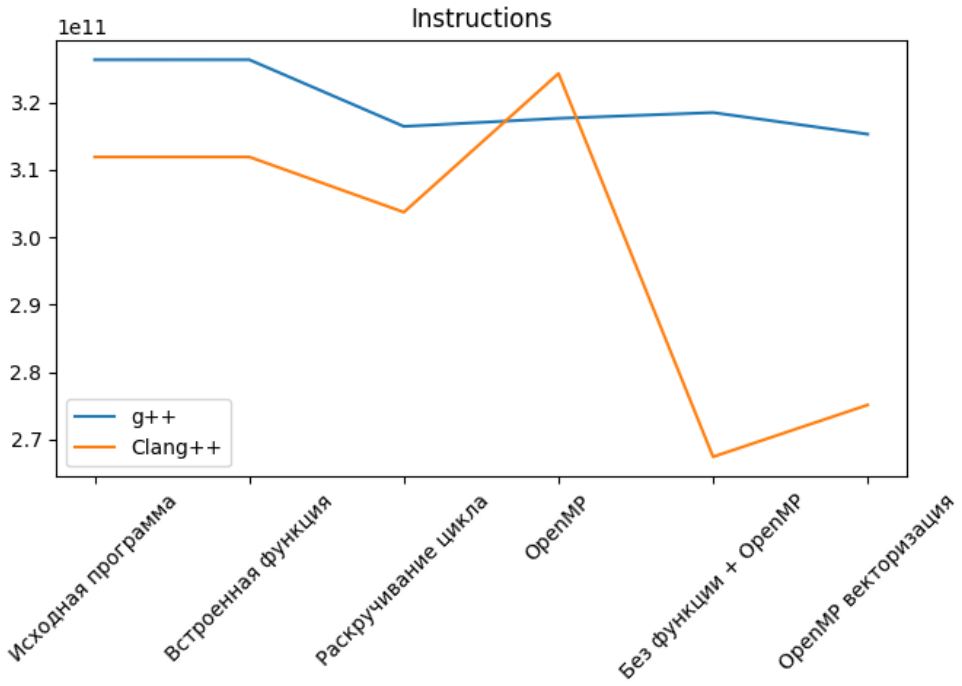

Рисунок 1. Количество инструкций

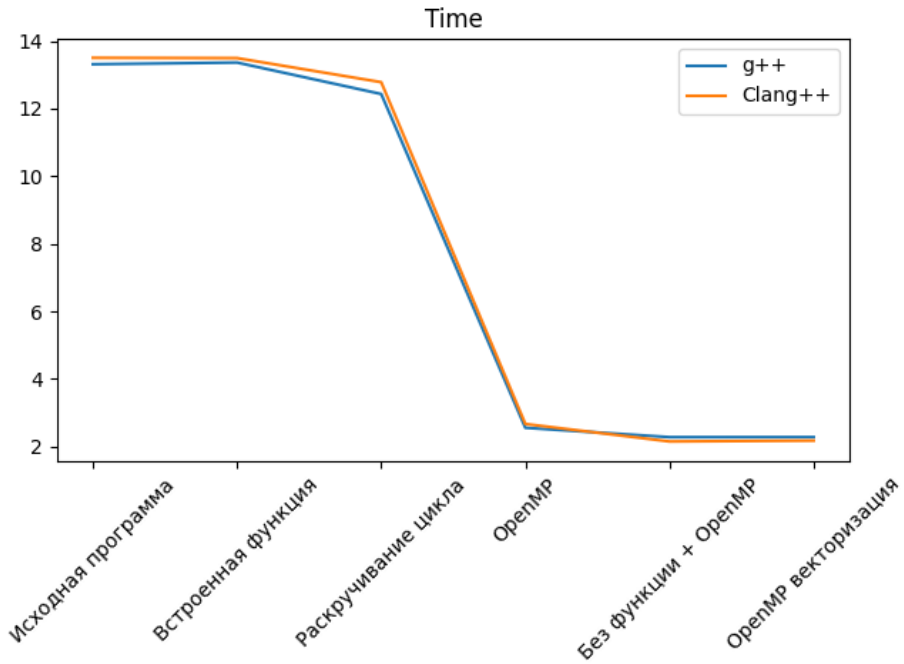

Рисунок 2. Время выполнения

Итоговая таблица:

Компилятор / Метрика	g++	clang++
Исходная программа		
Время выполнения, с	13,3109	13,5008
Кол-во инструкций	326 305 306 192	311 897 300 646
__builtin_expect		
Время выполнения, с	13,3613	13,4933
Кол-во инструкций	326 308 389 760	311 901 475 696
Раскручивание цикла		
Время выполнения, с	12,4306	12,7807
Кол-во инструкций	316 425 869 689	303 716 120 353
Расспараллеливание работы		
Время выполнения, с	2,55635	2,66892
Кол-во инструкций	334 392 019 305	324 272 625 344
Уход от функции		
Время выполнения, с	2,27947	2,15132
Кол-во инструкций	318 478 054 980	267 462 921 635
Векторизация		
Время выполнения, с	2,28031	2,17637
Кол-во инструкций	315 278 568 877	275 148 538 411

Вывод

Был реализован алгоритм для решения СЛАУ методом Гаусса, с последующими этапами оптимизации:

– использование встроенной функции __builtin_expect

Данный вариант на выбранных метриках (время выполнения и кол-во инструкций) и на примере для теста показал в целом те же результаты. Использование было основано на предположении, что в текущей для итерации строке рассматриваемый элемент, скорее всего, не будет нулем.

– раскручивание цикла

Раскрутка цикла (обработка сразу по 8 элементов) позволяет уменьшить кол-во условной составляющей цикла for, что и привело к улучшению результатов работы программы.

– расспараллеливание работы

Использование нескольких потоков заведомо ускоряет работу подобной программы. Что и вышло: ускорение в почти 5 раз, однако это влечет за собой увеличение кол-ва инструкций, выполняемых во время работы (стоит заметить, что они распределены уже не на одно ядро, а на все сразу).

– вынос тела функции с целью ухода от ее вызова

Убрав функцию, были сведены к нулю затраты на ее вызов, что позволило сократить время работы программы, а также кол-во инстукций для выполнения.

– векторизация OpenMP

Векторизация позволяет ускорять действия над векторами (в случае double это сразу по 4 значения при использовании типа данных __mm256d). В данной реализации была использована директива #pragma openmp simd, что указывает на то, что цикл нужно векторизировать.

По итогу всех оптимизаций удалось сократить время работы в 5,8 раз. Количество инструкций в большей степени сократилось, используя компилятор clang++.

Приложение А. Исходный код

```
#include <ctime>
#include <chrono>

#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)

#define WARMUP_NUM 10
#define MEASURE_NUM 100

#define ROWS_NUM 700
#define COLUMNS_NUM 500

#define RAND_LEFT -1000
#define RAND_RIGHT 1000

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

double** solveGauss(double** matrix, int rows, int columns);
void sub_vector_from_vector2(double* vector1, double* vector2, int vectorSize,
double multiplyBy);
void swap_matrix_rows(double** matrix, int row1, int row2);

int main()
{
    std::srand(61771);
    double** fMatrix = new double*[ROWS_NUM];
    if(fMatrix == NULL)
    {
        std::cout << "Failed to allocate memory for matrix!" << std::endl;
        return 0;
    }

    for(int i = 0; i < ROWS_NUM; ++i)
    {
        fMatrix[i] = new double[COLUMNS_NUM];
        if(fMatrix[i] == NULL)
        {
            for(int j = 0; j < i; ++j)
                delete fMatrix[j];

            delete [] fMatrix;

            std::cout << "Failed to allocate memory for row " << i << " in
matrix!" << std::endl;
            return 0;
        }
    }
    //============================ GEN MATRIX ============================//
    for(int i = 0; i < ROWS_NUM; ++i)
        for(int j = 0; j < COLUMNS_NUM; ++j)
            fMatrix[i][j] = (double)randomInt(RAND_LEFT, RAND_RIGHT);
    //============================ SOLVE SYSTEM ============================//
    for(int i = 0; i < WARMUP_NUM; ++i)
        solveGauss(fMatrix, ROWS_NUM, COLUMNS_NUM);

    auto start = std::chrono::high_resolution_clock::now();
    for(int i = 0; i < MEASURE_NUM; ++i)
        solveGauss(fMatrix, ROWS_NUM, COLUMNS_NUM);
```

```
    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;
    //============================ FREE MEMORY ============================//
    for(int i = 0; i < ROWS_NUM; ++i)
        delete fMatrix[i];

    delete [] fMatrix;
    return 0;
}

/**
 * @brief решение СЛАУ методом Гаусса
 *
 * @param matrix        изначальная матрица
 * @param rows          кол-во строк
 * @param columns       кол-во столбцов
 * @return double**      матрица решений
 */
double** solveGauss(double** matrix, int rows, int columns)
{
    double** solution = new double*[rows];
    for(int i = 0; i < rows; ++i)           // copy origin matrix
    {
        solution[i] = new double[columns];
        for(int j = 0; j < columns; ++j)
            solution[i][j] = matrix[i][j];
    }

    bool firstZero;
    int row = 0, column = 0;
    while(row < rows-1 && column < columns-1)
    {
        firstZero = false;
        if(solution[row][column] == 0.0) // если ноль - надо менять
        {
            firstZero = true;
            for(int j = row+1; firstZero && j < rows; ++j)
                if(solution[j][column] != 0.0)
                {
                    swap_matrix_rows(solution, row, j);
                    firstZero = false;
                }

            if(firstZero)
            {
                ++column;
                continue;
            }
        }

        for(int j = row+1; j < rows; ++j) // проходим по всем строкам ниже
            sub_vector_from_vector2(solution[j], solution[row], columns,
(solution[j][column]/solution[row][column]));
        ++row;
    }
    return solution;
}
```

```
void sub_vector_from_vector2(double* vector1, double* vector2, int vectorSize,
double multiplyBy)
{
    for(int i = 0; i < vectorSize; ++i)
        vector1[i] = vector1[i] - vector2[i] * multiplyBy;
}

void swap_matrix_rows(double** matrix, int row1, int row2)
{
    double* temp = matrix[row2];
    matrix[row2] = matrix[row1];
    matrix[row1] = temp;
}
```