Государственное образовательное учреждение высшего
профессионального образования
«Московский Государственный Технический Университет имени Н. Э.
Баумана»

ОТЧЕТ
По лабораторной работе №2
По курсу «Анализ алгоритмов»
Тема: **«Умножение матриц»**

Студент: Кононенко С. Д.
Группа: ИУ7-51

Москва, 2017

Матрица - математический объект, эквивалентный двумерному массиву. Числа располагаются в матрице по строкам и столбцам. Числа располагаются в матрице по строкам и столбцам. Две матрицы одинакового размера можно поэлементно сложить или вычесть друг их друга. Если число столбцов в первой матрице совпадает с числом строк во второй, то эти две матрицы можно перемножить. У произведения будет столько же строк, сколько в первой матрице, и столько же столбцов, сколько во второй. При умножении матрицы размером 3 x 4 на матрицу размером 4 x 7 мы получаем матрицу размером 3 x 7.

Умножение матриц

Пусть даны две матрицы **A** и **B** размерности **a x n** и **n x b** соответственно, тогда результатом их умножения будет матрица **C** размерности **a x b**, в которой

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$$

Умножение матриц по Винограду

Если посмотреть на результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее.

Рассмотрим два вектора **V** = (v_1, v_2, v_3, v_4) и **W** = (w_1, w_2, w_3, w_4). Их скалярное произведение равно:

V • W = v1w1 + v2w2 + v3w3 + v4w4.

Это равенство можно переписать в виде:

V • W = (v1 + w2)(v2 + w1) + (v3 + w4)(v4 + w3) - v1v2 - v3v4 - w1w2 - w3w4.

Из этого следует, что произведение матриц можно выполнять эффективнее, произведя некоторые вычисления заранее

Стандартный алгоритм умножения матриц

```
matrix_t matr_mult(const matrix_t m1, const matrix_t m2, unsigned long long int *t)
{
    matrix_t res;

    res = create_matrix(m1.n, m2.m);
    *t = tick();
    for (int i = 0; i < m1.n; i++)
        for (int j = 0; j < m2.m; j++)
        {
            res.matr[i][j] = 0;
            for (int k = 0; k < m1.m; k ++)
                res.matr[i][j] += m1.matr[i][k] * m2.matr[k][j];
        }
    *t = tick() - *t;
    return res;
}
```

Алгоритм Винограда

```
matrix_t vinograd_mult(const matrix_t m1, const matrix_t m2, unsigned long
long int *t)
{
    matrix_t res;

    res = create_matrix(m1.n, m2.m);

    int *mul1 = malloc(sizeof(int) * m1.n ),
        *mul2 = malloc(sizeof(int) * m2.m);

    *t = tick();
    for (int i = 0; i < m1.n; i++)
    {
        mul1[i] = m1.matr[i][0] * m1.matr[i][1];
        for (int j = 1; j < m1.m / 2; j++)
            mul1[i] += m1.matr[i][2 * j] * m1.matr[i][2 * j + 1];
    }

    for (int j = 0; j < m2.m; j++)
    {
        mul2[j] = m2.matr[0][j] * m2.matr[1][j];
        for (int i = 1; i < m1.n / 2; i++)
            mul2[j] += m2.matr[2 * i][j] * m2.matr[2 * i + 1][j];
    }

    for (int i = 0; i < m1.n; i++)
        for (int j = 0; j < m2.m; j++)
        {
            res.matr[i][j] = -mul1[i] - mul2[j];
            for (int k = 0; k < m1.m / 2; k ++)
                res.matr[i][j] += (m1.matr[i][2*k+1] + m2.matr[2*k][j]) *
                                  (m1.matr[i][2*k] + m2.matr[2*k+1][j]);
        }

    if (m1.m  % 2 == 1)
        for (int i = 0; i < m1.n; i++)
            for (int j = 0; j < m2.m; j++)
                res.matr[i][j] += m1.matr[i][m1.m - 1]* m2.matr[m2.n - 1][j];
    *t = tick() - *t;

    free(mul1);
    free(mul2);
    return res;
}
```

Модифицированный алгоритм Винограда

```
matrix_t vinograd_mult_o2(const matrix_t m1, const matrix_t m2, unsigned long long int *t)
{
    matrix_t res;

    res = create_matrix(m1.n, m2.m);

    int *mul1 = malloc(sizeof(int) * m1.n ),
        *mul2 = malloc(sizeof(int) * m2.m);

    *t = tick();
    int tmp1 = m1.m - 1, tmp2 = m1.n - 1;
    for (int i = 0; i < m1.n; i++)
    {
        mul1[i] = m1.matr[i][0] * m1.matr[i][1];
        for (register int j = 2; j < tmp1; j += 1)

            mul1[i] += m1.matr[i][j] * m1.matr[i][j + 1];
    }

    for (int j = 0; j < m2.m; j++)
    {
        mul2[j] = m2.matr[0][j] * m2.matr[1][j];
        for (register int i = 2; i < tmp2; i += 2)
            mul2[j] += m2.matr[i][j] * m2.matr[i + 1][j];
    }
    int flag = m1.m % 2;
    for (register int i = 0; i < m1.n; i++)
        for (register int j = 0; j < m2.m; j++)
        {
            res.matr[i][j] = -mul1[i] - mul2[j] +
                    (flag ? m1.matr[i][tmp1]* m2.matr[tmp2][j] : 0);
            for (register int k = 0; k < tmp1; k += 2)
                res.matr[i][j] += (m1.matr[i][k+1] + m2.matr[k][j]) *
                                  (m1.matr[i][k] + m2.matr[k+1][j]);
        }
    *t = tick() - *t;

    free(mul1);
    free(mul2);
    return res;
}
```

Теоретическая оценка:

<u>Сложность 1: </u>+, -, * ,/, <, >, <=, >=, =, ==, +=, -=, /=, *=, []

Размер исходный матриц **N x N** , учитывается только основной цикл т.к. в остальные ассимптотически имеют меньшую сложность

Стандартный алгоритм:

$2 + N(2_{цикл} + 2 + N(2_{цикл} + 3 + 2 + N(2_{цикл} + 8))) = 2 + 4N + 7NN + 10NNN \sim \mathbf{10N^3}$

Алгоритм Винограда:

$2 + N(2_{цикл} + N(2_{цикл} + 7 + 3 + (N/2)(3 + 20))) = 2 + 2N + 12NN + 11.5NNN \sim \mathbf{11.5N^3}$

Учитывая реальную сложность умножения относительно сложения на вычислительной машине, стоит ожидать от алгоритма Винограда большей скорости работы.

Результаты работы

*По оси ординат используется время работы в тиках *1e-6*
По оси абсцисс – размерность матрицы

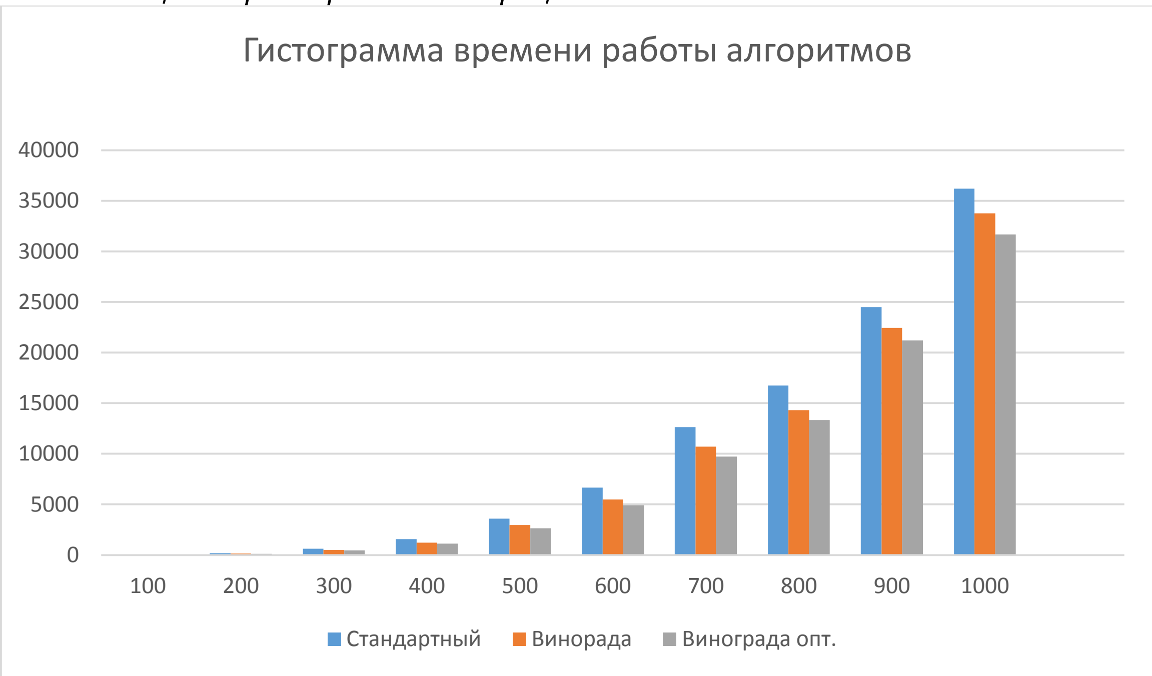

Рисунок 1 Гистограмма времени работы алгоритмов умножения матриц

*По оси ординат используется время работы в тиках *1e-6*
По оси абсцисс – размерность матрицы

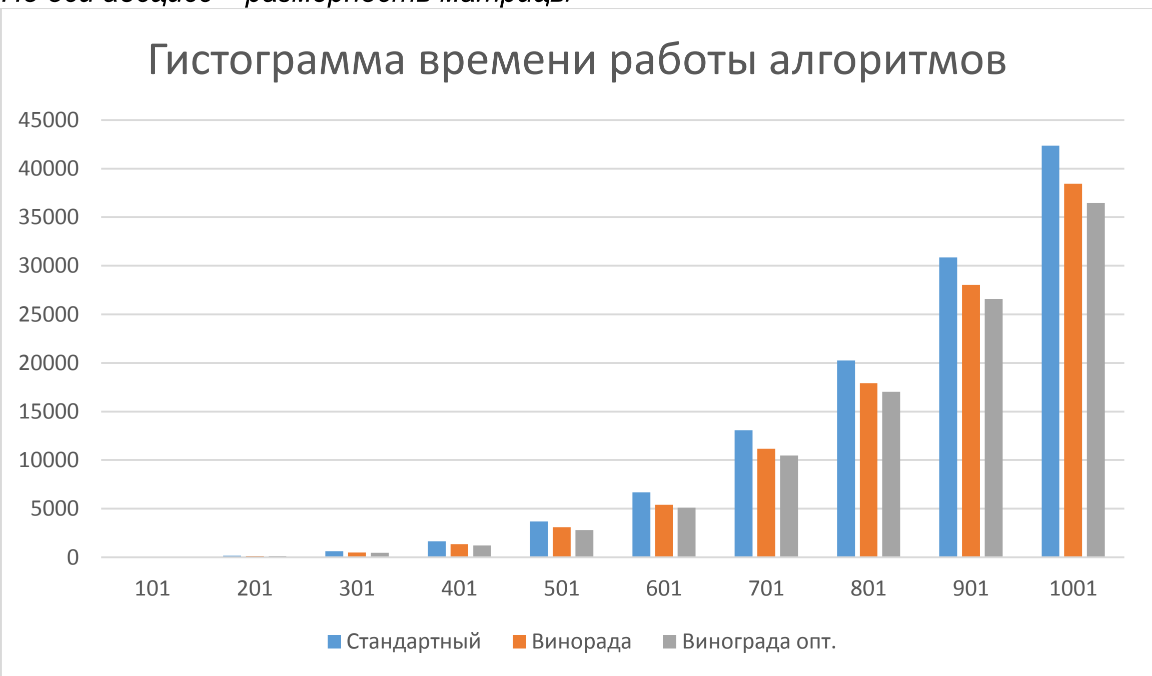

Рисунок 2 Гистограмма времени работы алгоритмов умножения матриц

Заключение

В результате выполнения лабораторной работы были экспериментально получены временные характеристики работы алгоритмов которые подтвердили теоретическую оценку.