Государственное образовательное учреждение высшего профессионального образования
«Московский государственный технический университет имени Н. Э. Баумана»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Р А С Ч Ё Т Н О - П О Я С Н И Т Е Л Ь Н А Я З А П И С К А
к лабораторной работе на тему:

Перемножение матриц : алгоритм Винограда

Студент ______________________ Киселев А.М.
(Подипсь, дата)

Преподаватель ______________________ Волкова Л.Л.
(Подпись, дата)

Москва 2018

Содержание

1 Аналитический раздел

Умножение матриц – важная задача в разных областях, которая помогает рассчитать системы линейных уравнение, которые в свою очередь применяются в системах моделирования реального мира, 3D моделировании, в обработке и хранении информации.

Задача по эффективной обработке матриц является актуальной и востребованной в наши дни. Первым шагом на пути к более быстрому по времени перемножении матриц стал алгоритм Винограда. Он представляет собой перемножение матриц с использованием скалярного произведения соответствующих строки и столбца исходных матриц.

1.1 Стандартный алгоритм перемножения матриц.

Пусть даны 2 квадратные матрицы размерностью $[l * m]$ и $[m * n]$ соответсвенно.

$$A = \begin{Bmatrix} a_1 1 & a_1 2 & ...a_1 m \\ a_2 1 & a_2 2 & ...a_2 m \\ ... & & \\ a_l 1 & a_l 2 & ...a_l m \end{Bmatrix}$$

$$B = \begin{Bmatrix} b_1 1 & b_1 2 & ...b_1 n \\ b_2 1 & b_2 2 & ...b_2 n \\ ... & & \\ b_m 1 & b_m 2 & ...b_m n \end{Bmatrix}$$

Тогда результатом умножения будет матрица размерностью $[l * n]$ вида:

$$C = \begin{Bmatrix} c_1 1 & c_1 2 & ...c_1 n \\ c_2 1 & c_2 2 & ...c_2 n \\ ... & & \\ c_l 1 & c_l 2 & ...c_l n \end{Bmatrix}$$

$c_{ij} = \sum_{r=1}^{m} a_{ir} * b_{rj}$, где $(i = 1, 2, ...l; j = 1, 2, ...n)$.

Операция умножения двух матриц выполнима только в том случае, если число столбцов в первом сомножителе равно числу строк во втором; в этом случае говорят, что матрицы согласованы. В частности, умножение всегда выполнимо, если оба сомножителя — квадратные матрицы одного и того же порядка.

1.2 Алгоритм Винограда

Пусть даны матрицы A[N*M] и B[M*K], тогда их результирующее произведение можно представить как матрицу C[N*K]. При перемножении матриц стандартным алгоритмом, мы получаем много операций умножения, которые являются очень трудоемкими по сравнению со сложением:

$$A * B = v_1 * w_1 + v_2 * w_2 + v_3 * w_3 + ... + v_m * w_m \tag{1.1}$$

Но это выражение можно представить в другом виде, в котором хорошо будут видны элементы, которые возможно рассчитать заранее и использовать несколько раз. Рассмотрим на частном случае: Пусть даны вектора $V = (v1, v2, v3, v4)$ и $W = (w1, w2, w3, w4)$. Произведение $V * W$ можно найти по формуле 1.1, которая представлена в общем виде, но можно записать так:

$$V*W = (v_1+w_2)*(v_2+w_1)+(v_3+w_4)*(v_4+w_3)-v_1*v_2-v_3*v_4-w_1*w_2-w_3*w_4 \tag{1.2}$$

Кажется, что второе выражение задает больше работы, чем первое: вместо четырех умножений мы насчитываем их шесть, а вместо трех сложений - десять. Менее очевидно, что выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй. Часть $-v_1 * v_2 - v_3 * v_4 - w_1 * w_2 - w_3 * w_4$ из выражения 1.2 вычислима заранее, что позволят нам избавиться от лишних трудоемких операций умножения. Таким образом, несмотря на то, что второе выражение требует вычисления большего количества операций, чем первое: вместо четырех умножений - шесть, а вместо трех сложений - десять, выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй, что позволяет выполнять для каждого элемента лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения.

1.3 Рассчет трудоемкости алгоритма

Рассчет трудоемкости алгоритма опирается на модель вычисления, в которой нужно определить свод правил : оценка операций, циклов, стоимость перехода по условиям.

1.3.1 Модель вычеслений

Стоимость каждой операции из следующего множества будем считать 1: {+,-,*, /, %, <, <=, !=, >=, >, =, :=, &, ||, []}

Стоимость условного перехода if-else – 0 и будет учитывать лишь стоимость выражения условия.

Стоимость циклал будет рассчитываться следующим образом:

Пусть цикл задан так:

$$for(i = 0, i < N, i++)\{body\} \tag{1.3}$$

, где N – количество итераций

Тогда i=0, i<N – операции, выполняемые перед началом выполнения тела цикла и их общая стоимость равна 2, i++, i<N – операции, которые выполняются каждую итерацию(их стоимость в общем тоже равна 2)

$$f_{for} = 2 + N * (2 + f_{body}) \tag{1.4}$$

2 Констукторский раздел

2.1 Разработка алгоритмов

2.1.1 Классический алгоритм умножения матриц

В листенге 2.1 представлен стандартный алгоритм умножения матриц.

Листинг 2.1 — Pseudocode of classic matrix multiplication

```
пока i < число строк матрицы A:
    пока j < число элементов в строке матрицы B:
        пока k < число строк матрицы B:
            Результирующая матрица C[i][j] += A[i][k]*B[k][j]
```

2.1.2 Классический алгоритм Винограда умножения матриц

В листинге 2.2 представлен классический алгоритм умножения Винограда

Листинг 2.2 — Pseudocode of classic Winograd algorithm

```
d = число строк B // 2
пока i < число строк матрицы A:
    пока j < d:
        rows[i] += A[i][2*j]*A[i][2*j + 1]

пока i < число столбцов B:
    пока j < d:
        colms[i] += B[2*j][i]*B[2*j+1][i]

пока i < число строк A:
    пока j < число столбцов B:
        результат[i][j] = -rows[i] - colms[i]
        пока k < d:
            результат[i][j] += (A[i][2*k] + B[2*k+1][j]) * (A[i][2*k+1] +
                B[2*k][j])

если количество элементов в строке матрицы A нечетная:
    пока i < число строк в A:
        пока j < число элементов в строке B:
            результат[i][j] += A[i][индекс последнего элемента строки A] *
                B[индекс последнего элемента строки A][j];
```

3 Технологический раздел

Данный раздел содержит информацию о реализации ПО и листингах програм.

3.1 Требования к програмному ПО

Программы реализованы в двух вариациях. Первая – каждая программа реализована поотдельности и обладает своим собственным функционалом – возможность определять размеры матриц и получать вывод в поток вывода. Вторая – программа, содержащая в себе функции с реализованными алгоритмами, в которой производится тестирование времени алгоритмов и вывод отправляется в поток вывода информации.

3.2 Средства реализации

Данные реализации разрабатывались на языке C, использовался компилятор gcc-8.2.1. Замер времени производится с помощью библиотеки *time.h*. Данная библиотека позволяет производить замер тиков, откуда можно получать время в секундах. Программы компилировались с выключенной оптимизацией с флагом -O0

3.3 Листинг

Алгоритм Винограда классический представлен в листинге 3.1

```

void winograd(  int** a, int ra, int ca  ,
                int** b, int rb, int cb,
                int** c, int rc, int cc,
                int* rows, int* columns)
{
    // f1 = 2 + 2 + ra(2 + 6 + 1 + 1 + 1 + 2 + d(2 + 6 + 1 + 2 + 3))=
    //    = ra*d*14 + 13*ra + 2
    int d = ca / 2;

    for(int i = 0; i < ra; i++) {
        rows[i] = rows[i] + a[i][0] * a[i][1];
        for(int j = 1; j < d; j++)
            rows[i] = rows[i] + a[i][2*j] * a[i][2*j+1];
    }

    // f2 = 2 + cb(2 + 6 + 1 + 1 + 1 + 2 + d(2 + 6 + 1 + 2 + 3))
    //    = cb * d * 14 + 13*cb + 2
    for(int i = 0; i < cb; i++) {
        columns[i] = columns[i] + b[0][i]*b[1][i];
        for(int j = 1; j < d; j++)
```

```
            columns[i] = columns[i] + b[2*j][i] * b[2*j+1][i];
    }

    // f3 = 2 + ra(2 + 2 + cb(2 + 4 + 2 + 1 + 2 + d(2 + 12 + 1 + 5 + 5))) =
    //    = 25*ra*cb*d + 11*ra*cb + 4*ra + 2
    for(int i = 0; i < ra; i++)
        for(int j = 0; j < cb; j++) {
            c[i][j] = -rows[i] - columns[j];
            for(int k = 0; k < d; k++)
                c[i][j] = c[i][j] + (a[i][2*k] + b[2*k+1][j]) *
                                (a[i][2*k+1] + b[2*k][j]);
        }

    // f4 = 1 + ( (0) or (ra*cb*14 + 4*ra + 2) )

    if(ca%2)
        for(int i = 0; i < ra; i++)
            for(int j = 0; j < cb; j++)
                c[i][j] = c[i][j] + a[i][ca-1] * b[ca-1][j];

    // fwinogradaBest = 25*ra*cb*d + 11*ra*cb + 14*ra*d + 14*cb*d + 17*ra +
    13*cb + 9
    // fwinogradWorst = 25*ra*cb*d + 25*ra*cb + 14*ra*d + 14*cb*d + 21*ra +
    13*cb + 11
}
```

Листинг 3.1 — Winograd algorithm

Оптимизированный алгоритм Винограда представлен в листинге 3.2

```
void winogradEnhanced(  int** a, int ra, int ca ,
                        int** b, int rb, int cb,
                        int** c, int rc, int cc,
                        int* rows, int* columns)
{
    // f1 = 2 + 2 + ra(2 + 2 + d(2 + 5 + 1 + 1 + 1))=
    //    = 10*ra*d + 4*ra + 4
    int d = ca - 1;
    for(int i = 0; i < ra; i++) {
        for(int j = 0; j < d; j+=2)
            rows[i] += a[i][j] * a[i][j+1];
    }

    // f2 = 2 + cb(2 + 2 + d(2 + 5 + 1 + 1 + 1))=
    //    = 10*cb*d + 4*cb + 4
    for(int i = 0; i < cb; i++) {
        for(int j = 0; j < d; j+=2)
```

```
            columns[i] += b[j][i] * b[j+1][i];
    }

    // f3 = 2 + ra(2 + 2 + cb(2 + 4 + 2 + 1 + 2 + d(2 + 10 + 1 + 4 + 1))) =
    //    = 18*ra*cb*d + 11*ra*cb + 4*ra + 2
    for(int i = 0; i < ra; i++)
        for(int j = 0; j < cb; j++) {
            c[i][j] = -rows[i] - columns[j];
            for(int k = 0; k < d; k+=2)
                c[i][j] += (a[i][k] + b[k+1][j]) *
                            (a[i][k+1] + b[k][j]);
        }

    // f4 = 1 + ( (0) or (ra*cb*8 + 4*ra + 2) )

    if(ca%2)
        for(int i = 0; i < ra; i++)
            for(int j = 0; j < cb; j++)
                c[i][j] += a[i][d] * b[d][j];

    // fwinogradEnhancedBest = 18*ra*cb*d + 10*cb*d + 10*ra*d + 11*ra*cb +
    8*ra + 4*cb + 10
    // fwinogradEnhancedWorst = 18*ra*cb*d + 10*cb*d + 10*ra*d + 19*ra*cb +
     12*ra + 4*cb + 12
}
```

Листинг 3.2 — Winograd enhanced algorithm

Листинг стандартного перемножения матриц представлен в 3.3

```
void classic(            int** a, int ra, int ca ,
                         int** b, int rb, int cb,
                         int** c, int rc, int cc)
{
    for(int i = 0; i < ra; i++)
        for(int j = 0; j < cb; j++)
            for(int k = 0; k < rb; k++)
                c[i][j] += a[i][k] * b[k][j];
    // fstd = 2 + ra(2 + 2 + cb(2 + 2 + rb(1 + 1 + 8 + 1 + 2))) = 13*ra*cb*
    rb + 4*ra*cb + 4*ra + 2
}
```

Листинг 3.3 — Classic matrix multiplication

4 Экспериментальный раздел

Результаты работы алгоритмов представлены в таблицах 4.1 и 4.2(для четных и нечетных размерностей матриц). Полученные данные представлены в виде диаграмм изображенных на рисунках 4.1 и 4.2

1 – Классический алгоритм Винограда(на графиках представлен черным цветом)

2 – Оптимизированный алгоритм Винограда(на графиках представлен красным цветом)

3 – Последовательное перемножение матриц(на графиках представлен синим цветом)

Таблица 4.1 — Таблица для сравнения результатов работы алгоритма для четной размерности в микросекундах.

Размер	**1**	**2**	**3**
100 X 100	20.077	20.859	25.865
200 X 200	258.613	154.072	201.215
300 X 300	536.736	516.737	687.302
400 X 400	1277.363	1229.492	1657.234
500 X 500	2508.538	2408.629	3294.885
600 X 600	4878.377	4685.910	6107.689
700 X 700	7778.614	7495.057	9871.546
800 X 800	11597.817	11170.562	14822.127
900 X 900	16712.889	16070.812	21545.400
1000 X 1000	23102.137	22225.149	29986.602

По полученным данным можно сделать вывод, что стандартный алгоритм перемножения матриц работает медленнее, чем классический алгоритм Винограда и улучшенный алгоритм Винограда. Улучшенный алгоритм работает быстрее по времени, чем классический. Если обратиться к рассчитанным трудоемкостям алгоритмов, можно заметить, что стандартный алгоритм менее трудоемкий, чем два других, но время, затрачиваемое на рассчет матрицы стандартным методом все равно очень большое. Так как это зависит от модели вычислений, где умножение и обращения к индексам и сложение были взяты за 1, можно сказать, что следует учитывать некоторые ньюансы при выборе весов операций, чтобы вычесленная трудоемкость была более точной.

Таблица 4.2 — Таблица для сравнения результатов работы алгоритма для нечетной размерности в микросекундах.

Размер	**1**	**2**	**3**
101 X 101	20.809	21.282	26.734
201 X 201	161.322	156.745	204.425
301 X 301	541.239	525.954	692.841
401 X 401	1284.754	1236.969	1685.261
501 X 501	2528.653	2422.614	3316.100
601 X 601	4918.674	4727.957	6148.991
701 X 701	7833.712	7528.168	9911.066
801 X 801	11725.979	11282.570	15007.543
901 X 901	16744.922	16120.746	21658.227
1001 X 1001	23189.963	22264.232	30098.645

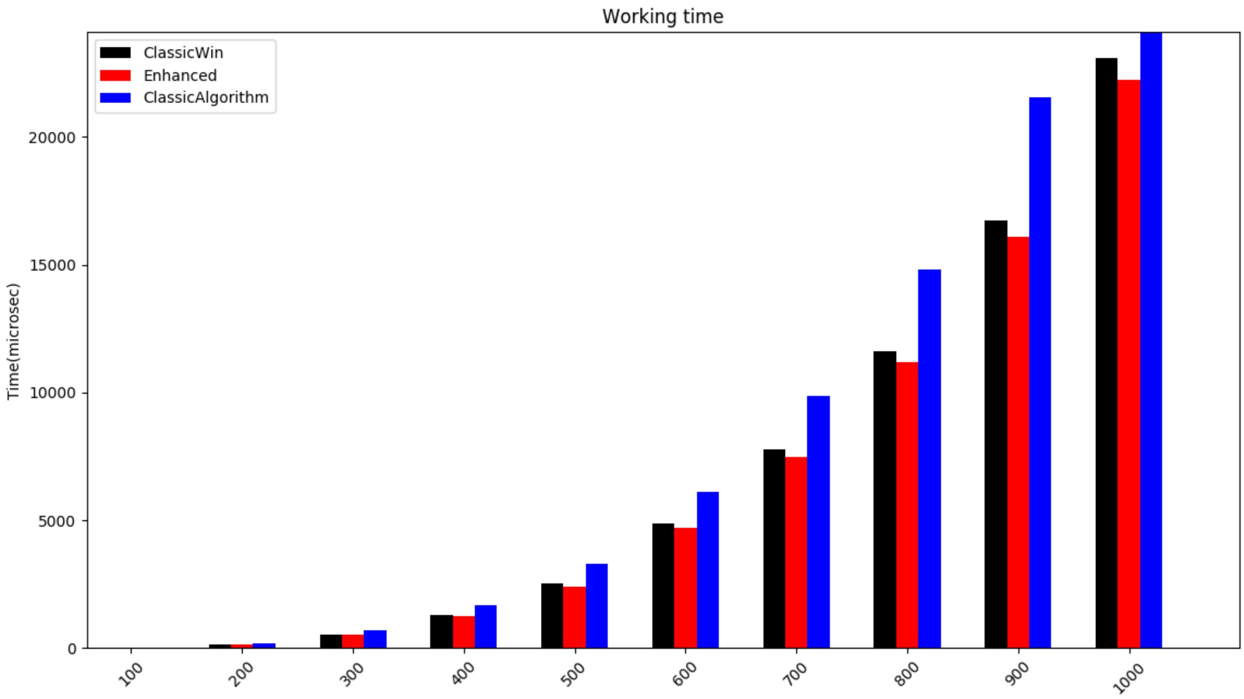

Рисунок 4.1 — График для квадратных матриц четной размерности

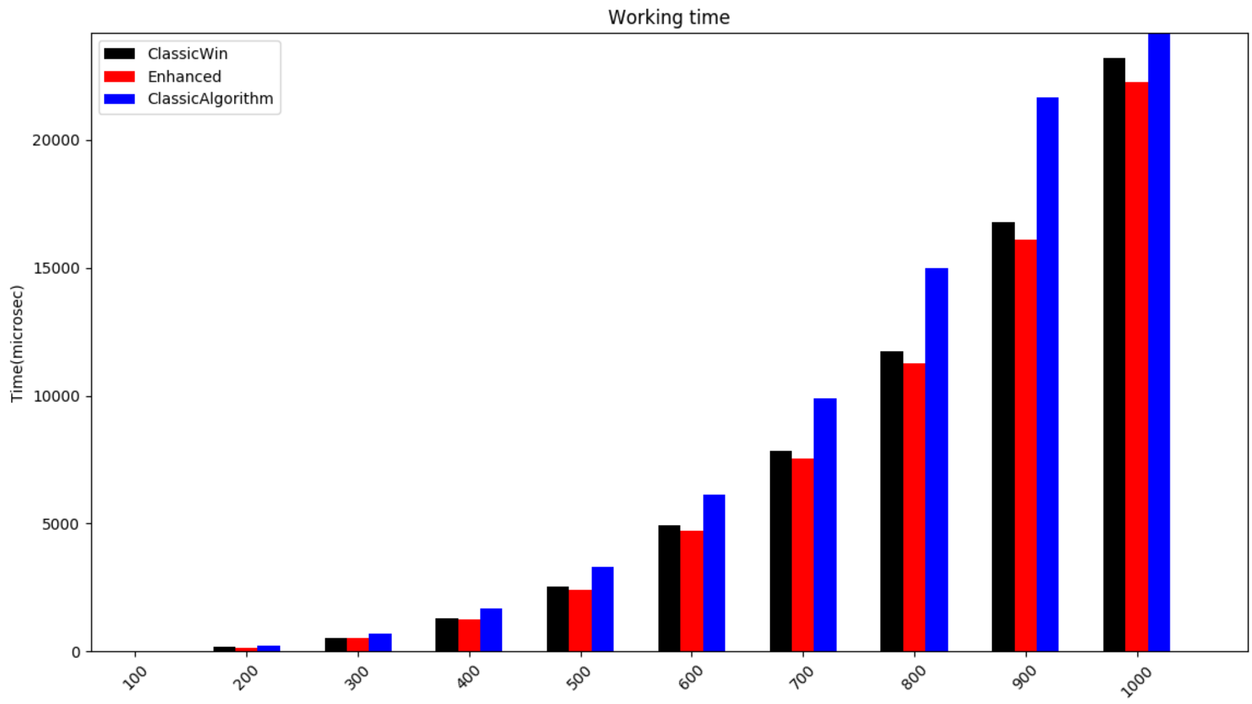

Рисунок 4.2 — График для квадратных матриц нечетных размерностей

Заключение

В результате выполнения лабораторной работы были получены следующие основные навыки:

а) изучены теоритические понятия в алгоритмах перемножения матриц;

б) рассмотрены различные способы оптимизации алгоритмов;

в) проведено сравнение четырех реализаций заданного алгоритма Винограда, выявлены их слабые места;