

*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»

РАСЧЁТНО - ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к лабораторной работе на тему:

Перемножение матриц : алгоритм Винограда

Студент	_____	Киселев А.М.
	(Подпись, дата)	
Преподаватель	_____	Волкова Л.Л.
	(Подпись, дата)	

Москва 2018

Содержание

Введение	3
1 Аналитический раздел	4
1.1 Алгоритм Винограда	4
1.2 Расчет трудоемкости алгоритма	5
2 Конструкторский раздел	6
2.1 Разработка алгоритмов	6
2.1.1 Классический алгоритм умножения матриц	6
2.1.2 Классический алгоритм Винограда умножения матриц	6
3 Технологический раздел	7
3.1 Требования к программному ПО	7
3.2 Средства реализации	7
3.3 Листинг	7
4 Экспериментальный раздел	10
Заключение	12

Введение

Целью работы является изучение нахождения трудоемкости на материале алгоритма перемножения матриц Винограда. Для достижения поставленной цели необходимо решить следующие задачи:

- Изучить алгоритм Винограда;
- Получить улучшенную версию Винограда и сравнить ее с обычной реализацией;
- Провести оценку трудоемкости алгоритма;
- Рассмотреть лучший и худший случаи при перемножении матриц данным алгоритмом;
- Привести экспериментальное подтверждение различий во временной эффективности оптимизированного алгоритма и обычного при помощи разработанного ПО на материале замеров процессорного времени выполнения реализации на варьирующихся размерах перемножаемых матриц;
- Описать и обосновать полученные результаты о выполненной работе;

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 перемножению матриц стал алгоритм Винограда. Он представляет собой перемножение матриц с использованием скалярного произведения соответствующих строки и столбца исходных матриц.

Более подробное описание алгоритма:

1.1 Алгоритм Винограда

Пусть даны матрицы $A[N \times M]$ и $B[M \times K]$, тогда их результирующее произведение можно представить как матрицу $C[N \times K]$. При умножении "в лоб" мы получаем много операций умножения, которые являются очень трудоемкими по сравнению со сложением:

$$A * B = v_1 * w_1 + v_2 * w_2 + v_3 * w_3 + \dots + v_m * w_m \quad (1.1)$$

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 $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Произведение $V * W$ можно найти по формуле 1.1, которая представлена в общем виде, но можно записать так:

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

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.

1.2 Расчет трудоемкости алгоритма

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

2 Конструкторский раздел

2.1 Разработка алгоритмов

2.1.1 Классический алгоритм умножения матриц

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

2.1.2 Классический алгоритм Винограда умножения матриц

```
1 b = число строк B // 2
2 пока i < число строк матрицы A:
3     пока j < число строк B // 2:
4         rows[i][j] += A[i][2*j]*A[i][2*j + 1]
5
6 пока i < число столбцов B:
7     пока j < число строк B // 2:
8         cols[i] += B[2*j][i]*B[2*j][i]
9
10 пока i < число строк A:
11     пока j < число столбцов B // 2:
12         если размерность матрицы четная:
13             результат[i][j] = sum(A[i][b-1]*B[b-1][j])
14         иначе:
15             результат[i][j] = sum((A[i][2 * k] + B[2 * k + 1][j]) *
                                     (A[i][2 * k + 1] + B[2 * k][j]) - rows[i] - cols[j])
```

3 Технологический раздел

Данный раздел содержит информацию о реализации ПО и листингах программ.

3.1 Требования к программному 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

```
1
2 void winograd(  int** a, int ra, int ca ,
3                 int** b, int rb, int cb ,
4                 int** c, int rc, int cc ,
5                 int* rows, int* columns)
6 {
7     // f1 = 2 + 2 + ra(2 + 6 + 1 + 1 + 1 + 2 + d(2 + 6 + 1 + 2 + 3))=
8     //      = ra*d*14 + 13*ra + 2
9     int d = ca / 2;
10
11     for(int i = 0; i < ra; i++) {
12         rows[i] = rows[i] + a[i][0] * a[i][1];
13         for(int j = 1; j < d; j++)
14             rows[i] = rows[i] + a[i][2*j] * a[i][2*j+1];
15     }
16
17     // f2 = 2 + cb(2 + 6 + 1 + 1 + 1 + 2 + d(2 + 6 + 1 + 2 + 3))
18     //      = cb * d * 14 + 13*cb + 2
19     for(int i = 0; i < cb; i++) {
20         columns[i] = columns[i] + b[0][i]*b[1][i];
21         for(int j = 1; j < d; j++)
```

```

22         columns[i] = columns[i] + b[2*j][i] * b[2*j+1][i];
23     }
24
25     // f3 = 2 + ra(2 + 2 + cb(2 + 4 + 2 + 1 + 2 + d(2 + 12 + 1 + 5 + 5))) =
26     //      = 25*ra*cb*d + 11*ra*cb + 4*ra + 2
27     for(int i = 0; i < ra; i++)
28         for(int j = 0; j < cb; j++) {
29             c[i][j] = -rows[i] - columns[j];
30             for(int k = 0; k < d; k++)
31                 c[i][j] = c[i][j] + (a[i][2*k] + b[2*k+1][j]) *
32                     (a[i][2*k+1] + b[2*k][j]);
33         }
34
35     // f4 = 1 + ( (0) or ra*cb*14 + 4*ra + 2 )
36
37     if(ca%2)
38         for(int i = 0; i < ra; i++)
39             for(int j = 0; j < cb; j++)
40                 c[i][j] = c[i][j] + a[i][ca-1] * b[ca-1][j];
41
42     // fwinogradaBest = 25*ra*cb*d + 11*ra*cb + 14*ra*d + 14*cb*d + 17*ra +
43     //                  13*cb + 9
44     // fwinogradWorst = 25*ra*cb*d + 25*ra*cb + 14*ra*d + 14*cb*d + 21*ra +
45     //                  13*cb + 11
46 }

```

Листинг 3.1 — Winograd algorithm

Оптимизированный алгоритм Винограда представлен в листинге 3.2

```

1 void winogradEnhanced( int** a, int ra, int ca ,
2                       int** b, int rb, int cb,
3                       int** c, int rc, int cc,
4                       int* rows, int* columns)
5 {
6     // f1 = 2 + 2 + ra(2 + 2 + d(2 + 5 + 1 + 1 + 1))=
7     //      = 10*ra*d + 4*ra + 4
8     int d = ca - 1;
9     for(int i = 0; i < ra; i++) {
10         for(int j = 0; j < d; j+=2)
11             rows[i] += a[i][j] * a[i][j+1];
12     }
13
14     // f2 = 2 + cb(2 + 2 + d(2 + 5 + 1 + 1 + 1))=
15     //      = 10*cb*d + 4*cb + 4
16     for(int i = 0; i < cb; i++) {
17         for(int j = 0; j < d; j+=2)

```



```

18         columns[i] += b[j][i] * b[j+1][i];
19     }
20
21     // f3 = 2 + ra(2 + 2 + cb(2 + 4 + 2 + 1 + 2 + d(2 + 10 + 1 + 4 + 1))) =
22     //      = 18*ra*cb*d + 11*ra*cb + 4*ra + 2
23     for(int i = 0; i < ra; i++)
24         for(int j = 0; j < cb; j++) {
25             c[i][j] = -rows[i] - columns[j];
26             for(int k = 0; k < d; k+=2)
27                 c[i][j] += (a[i][k] + b[k+1][j]) *
28                     (a[i][k+1] + b[k][j]);
29         }
30
31     // f4 = 1 + ( (0) or ra*cb*8 + 4*ra + 2 )
32
33     if(ca%2)
34         for(int i = 0; i < ra; i++)
35             for(int j = 0; j < cb; j++)
36                 c[i][j] += a[i][d] * b[d][j];
37
38     // fwinogradEnhancedBest = 18*ra*cb*d + 10*cb*d + 10*ra*d + 11*ra*cb +
39     // 8*ra + 4*cb + 10
40     // fwinogradEnhancedWorst = 18*ra*cb*d + 10*cb*d + 10*ra*d + 19*ra*cb +
41     // 12*ra + 4*cb + 12
42 }

```

Листинг 3.2 — Winograd enhanced algorithm

4 Экспериментальный раздел

Результат работы алгоритмов

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

Размер	1	2
100 X 100	20.077	20.859
200 X 200	258.613	154.072
300 X 300	536.736	516.737
400 X 400	1277.363	1229.492
500 X 500	2508.538	2408.629
600 X 600	4878.377	4685.910
700 X 700	7778.614	7495.057
800 X 800	11597.817	11170.562
900 X 900	16712.889	16070.812
1000 X 1000	23102.137	22225.149

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

Размер	1	2
101 X 101	20.809	21.282
201 X 201	161.322	156.745
301 X 301	541.239	525.954
401 X 401	1284.754	1236.969
501 X 501	2528.653	2422.614
601 X 601	4918.674	4727.957
701 X 701	7833.712	7528.168
801 X 801	11725.979	11282.570
901 X 901	16744.922	16120.746
1001 X 1001	23189.963	22264.232

Заключение

В результате выполнения задания были получены следующие основные результаты:

- изучены теоретические понятия редуccionного расстояния
- изучены основные алгоритмы нахождения редуccionного расстояния между строками: алгоритм Левенштейна, Дамерау – Левенштейна
- проведен аналитический вывод формул для заполнения матриц расстояний
- проведено сравнение трех реализаций заданных алгоритмов, были выявлены их слабые места
- в рамках данной работы было выяснено, что рекурсивный алгоритм работает намного медленнее двух других из-за большого количества рекурсивных вызовов, что делает его бесполезным при работе с большими объемами информации, а алгоритм Левенштейна работает медленнее Дамерау – Левенштейна на маленьких словах, но на словах больше 6 символов, Дамерау – Левенштейн начинает работать медленнее