*Государственное образовательное учреждение высшего профессионального образования*

*«Московский Государственный Технический Университет имени Н. Э. Баумана»*

**ОТЧЕТ**

По лабораторной работе №1

По курсу «Анализ алгоритмов»

Тема: «Умножение матриц»

Студент: Иванов И. В.

Группа: ИУ7-53

Преподаватель: Волкова Л. Л.

Москва, 2014

**Стандартный алгоритм умножения матриц**

Код функции:

static void MultClassic (const cMatrix<T> &A, const cMatrix<T> &B, cMatrix<T> &C)

{

//cMatrix<T> \*result = new cMatrix<T>(A.m\_rows, B.m\_columns);

//cMatrix<T> &C = \*result;

for ( int i = 0; i < C.m\_rows; i++ ) //F: (2 + m\*(2 + F{})

{

for ( int j = 0; j < C.m\_columns; j++ ) //F: (2 + n\*(2 + F{})

{

C.m\_matrix[i][j] = 0; // F: 3

for ( int k = 0; k < B.m\_rows; k++ ) //F: (2 + q\*(2 + F{})

C.m\_matrix[i][j] += A.m\_matrix[i][k] \* B.m\_matrix[k][j]; // F: 8

}

}

}

Пусть матрица A имеет размерность m x n, а матрица B – n x q. В этом случае оценка трудоемкости алгоритма:  
.

**Алгоритм Винограда**

Код функции:

static void MultGrape(const cMatrix<T> &A, const cMatrix<T> &B, cMatrix<T> &C)

{

int m = A.m\_rows;

int n = A.m\_columns;

int q = B.m\_columns;

T \*mulh = new T [m];

T \*mulv = new T [q];

for (int i = 0; i < m; i++) // F: 2 + m(2 + F{})

{

mulh[i] = 0; // F: 1

for (int j = 0; j < n/2; j++) // F: 3 + n/2(3 + F{})

mulh[i] = mulh[i] + A.m\_matrix[i][2\*j] \* A.m\_matrix[i][2\*j+1]; // F: 12

// 1 1 1 1 1 2 1 1 3

}

for (int i = 0; i < q; i++) // F: 2 + q(2 + F{})

{

mulv[i] = 0; // F: 1

for (int j = 0; j < n/2; j++) // F: 3 + n/2(3 + F{})

mulv[i] = mulv[i] + B.m\_matrix[2\*j][i] \* B.m\_matrix[2\*j+1][i]; // F: 12

// 1 1 1 1 2 1 1 3 1

}

for (int i = 0; i < m; i++) // F: 2 + m(2 + F{})

{

for (int j = 0; j < q; j++) // F: 2 + q(2 + F{})

{

C.m\_matrix[i][j] = -mulh[i]-mulv[j]; // F: 7

for (int k = 0; k < n/2; k++) // F: 3 + n/2(3 + F{})

C.m\_matrix[i][j] = C.m\_matrix[i][j] + (A.m\_matrix[i][k\*2] + B.m\_matrix[k\*2+1][j]) \*

// 1 1 1 1 1 1 1 2 1 3 1 1  
(A.m\_matrix[i][2\*k+1] + B.m\_matrix[2\*k][j]); // F: 21  
 1 3 1 2 1

}

}

if ( n % 2 ) // F: 1

{

for (int i = 0; i < m; i++) // F: 2 + m(2 + F{})

for (int j = 0; j < q; j++) // F: 2 + q(2 + F{})

C.m\_matrix[i][j] = C.m\_matrix[i][j] + A.m\_matrix[i][n-1] \* B.m\_matrix[n-1][j]; // F: 13

// 1 1 1 1 1 1 1 2 1 2 1

}

delete []mulv;

delete []mulh;

}

«Слабым местом» алгоритма Винограда является необходимость выделения и освобождения дополнительной памяти под массивы строковых и столбцовых коэффициентов. Кроме того, в случае нечетного количества столбцов в первой (строк во второй) матрице, алгоритм должен выполнять дополнительные действия во вложенных циклах.

Оценка трудоемкости каждого блока алгоритмов:

Таким образом, результирующая трудоемкость алгоритма равна  
в лучшем случае: ;  
в худшем случае: .

**Оптимизированный алгоритм Винограда**

Алгоритм Винограда можно улучшить с помощью следующих добавлений:

1. Заменить a = a + x на a += x везде где это возможно;
2. В циклах for ( i=0; i<n/2; i++) заменить на for ( i=0; i<n; i+=2) (избавимся от долгой операции деления);
3. При заполнении массивов коэффициентов h,v записывать в них н +=, а -=, чтобы в основном цикле можно было записать c[i][j] += h[i]+v[j] вместо += -h[i] – h[j];
4. Внести проверку n%2 внутрь основного цикла, чтобы не нужно было проходить через два дополнительных for;
5. После этого вычислить n%2 только один раз, в зависимости от этого установить флаг, и проводить в цикле проверку if (flag) вместо if (n%2).

В результате получим оптимизированный алгоритм:

static void MultGrapeOpt(const cMatrix<T> &A, const cMatrix<T> &B, cMatrix<T> &C)

{

int m = A.m\_rows;

int n = A.m\_columns;

int q = B.m\_columns;

T \*mulh = new T [m];

T \*mulv = new T [q];

for (int i = 0; i < m; i++) // F: 2 + m(2 + F{})

{

mulh[i] = 0; // F: 1

for (int j = 1; j < n; j += 2) // F: 2 + n/2(2 + F{})

mulh[i] -= A.m\_matrix[i][j-1] \* A.m\_matrix[i][j]; // F: 8

// 1 1 1 2 1 1 1

}

for (int i = 0; i < q; i++) // F: 2 + m(2 + F{})

{

mulv[i] = 0; // F: 1

for (int j = 1; j < n; j += 2) // F: 2 + n/2(2 + F{})

mulv[i] -= B.m\_matrix[j-1][i] \* B.m\_matrix[j][i]; // F: 8

// 1 1 2 1 1 1 1

}

bool isOdd = (n % 2 == 1); // F: 3

for (int i = 0; i < m; i++) // F: 2 + m(2 + F{})

{

for (int j = 0; j < q; j++) // F: 2 + q(2 + F{})

{

C.m\_matrix[i][j] += mulh[i] + mulv[j]; // F: 6

for (int k = 1; k < n; k += 2) // n/2 // F: 2 + n/2(2 + F{})

C.m\_matrix[i][j] += (A.m\_matrix[i][k-1] + B.m\_matrix[k][j]) \* (A.m\_matrix[i][k] +

// 1 1 1 1 2 1 1 1 1 1 1 1  
B.m\_matrix[k-1][j]); // F: 16  
 2 1

if ( isOdd ) // F: 0

C.m\_matrix[i][j] += A.m\_matrix[i][n-1] \* B.m\_matrix[n-1][j]; // F\_if: 10

// 1 1 1 1 2 1 2 1

}

}

delete []mulv;

delete []mulh;

}

Оценка трудоемкости:

Таким образом, результирующая трудоемкость оптимизированного алгоритма равна  
в лучшем случае: ;  
в худшем случае: .

В неоптимизированном алгоритме  
в лучшем случае: ;  
в худшем случае: .

Разница в эффективности между оптимизированным и обычным алгоритмом составляет  
в лучшем случае: ;  
в худшем случае: .

Трудоемкость стандартного алгоритма равна .

Оптимизированный АВ (в худшем случае), также как и неоптимизированный АВ, оказываются более трудоёмкими, чем стандартный алгоритм, при сравнении по мажоранте (mnq). Тем не менее, оба АВ должны работать быстрее за счет того, что в них производится гораздо меньшее число умножений.

**Тестирование алгоритмов**

Были проведены исследования зависимости времени работы алгоритмов от размеров перемножаемых матриц. Замеры времени проводились с использованием функций QueryPerformanceFrequency и QueryPerformanceCounter, входящих в состав Windows API; использовались квадратные матрицы, заполняемые случайными числами.

|  |  |  |  |
| --- | --- | --- | --- |
| Размерность | Стандартный | Виноград | Опт. Виноград |
| 50 | 0,001 | 0,001 | 0,001 |
| 100 | 0,007 | 0,006 | 0,006 |
| 150 | 0,027 | 0,017 | 0,017 |
| 200 | 0,066 | 0,039 | 0,033 |
| 250 | 0,113 | 0,08 | 0,086 |
| 300 | 0,214 | 0,151 | 0,151 |
| 350 | 0,298 | 0,192 | 0,189 |
| 400 | 0,444 | 0,297 | 0,292 |
| 450 | 0,628 | 0,417 | 0,408 |
| 500 | 0,904 | 0,641 | 0,792 |
| 550 | 1,243 | 0,807 | 0,778 |
| 600 | 1,612 | 1,149 | 1,296 |
| 650 | 1,907 | 1,302 | 1,264 |
| 700 | 2,459 | 1,703 | 1,66 |
| 750 | 3,075 | 2,137 | 2,038 |
| 800 | 3,847 | 2,624 | 2,573 |
| 850 | 4,491 | 3,124 | 3,007 |
| 900 | 5,945 | 4,086 | 4,015 |
| 950 | 7,244 | 5,426 | 5,382 |
| 1000 | 9,343 | 7,201 | 6,91 |