Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

Московский государственный технический университет имени Н.Э.Баумана
(МГТУ им. Н.Э.Баумана)

ОТЧЕТ
По лабораторной работе № 2
По курсу «Анализ алгоритмов»
на тему «Алгоритмы умножения матриц»

Выполнил	Студент:	Московец Н.С
	Группа:	ИУ7-51

Москва, 2017

Оглавление

Постановка задачи

Изучить и реализовать алгоритмы умножения матриц: стандартный, алгоритм Винограда для умножения матриц и модифицированный алгоритм Винограда. Сравнить реализованные алгоритмы.

Описание модели вычислений

Опишем модель вычислений, которой будем пользоваться в дальнейшем при оценке трудоемкости алгоритмов.

Операции, имеющие трудоемкость “1”:

- арифметические операции { сложение(+), вычитание(-), умножение(*), деление(/), битовый сдвиг(<<, >>), деление нацело, взятие остатка(%) };
- логические операции { и(&&), не(!), или(||) };
- операции сравнения {<, >, =, !=, >=, <=};
- операции присваивания {=, +=, -=, *=, /=, %=};
- операция взятия индекса ([]);
- операции побитового И(&) и ИЛИ(|)

- унарный плюс и минус
- операции инкремента и декремента(постфиксные и префиксные) (++, --).

Операции, имеющие трудоемкость “0”:

- логический переход по ветвлению;
- операции обращения к полю структуры/класса (->, .);
- объявление переменных

Стандартный алгоритм

1. Описание

Стандартный алгоритм умножения матриц заключается в реализации формулы умножения матрицы A, с размерностью N×M на матрицу B с размерностью M×K:

$R[i][j] = \sum_{k=0}^{M} A[i][k] * B[k][j]$, где i, j - пробегают все значения i = 0..N - 1, j = 0..K - 1.

2. Реализация

```
1.  Matrix multStandart(const Matrix &a, const Matrix &b)
2.  {
3.      assert(a.col == b.row); //проверка корректности умножения
4.  
5.      Matrix res(a.row, b.col); //создание и обнуление результирующей матрицы
6.  
7.      for(int i = 0; i < a.row; i++) { // f1 =  2+ a.row * (2 + f2)
8.          for(int j = 0; j < b.col; j++) { // f2 = 2 + b.col * (2 + f3)
9.              for(int k = 0; k < b.row; k++) { //f3 = 2 + b.row * (2 + f4)
10.                     res.arr[i][j] += a.arr[i][k] * b.arr[k][j]; // f4 = 8
11.             }
12.         }
13.     }
14. 
15.     return res;
16. }
```

3. Теоретическая оценка

Трудоемкость алгоритма подсчитаем по строкам 7-13, так как проверка корректности матриц, выделение памяти под результирующую матрицу и ее обнуление не зависит от реализации алгоритма и выполняется для каждого и рассмотренных алгоритмов.
Сложность умножение матрицы N×M на матрицу M×K будет равна:
f = 2 + n(2 + 2 + m(2 + 2 + k(2+ 8))) = 2 + 4n + 4nm + 10nmk;

Алгоритм Винограда

1. Описание

Рассмотрим два вектора V = (v1, v2, v3, v4) и W = (w1, w2, w3, w4). Их скалярное произведение равно:

$$V \bullet W = v1w1 + v2w2 + v3w3 + v4w4.$$

Это равенство можно переписать в виде:

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

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.

Выражение v1w1 + v2w2 + v3w3 + v4w4 имеет большую трудоемкость, чем (v1 + w2)(v2 + w1) + (v3 + w4)(v4 + w3).

2. Реализация

```
Matrix multVinograd(const Matrix &a, const Matrix &b)
{
   assert(a.col == b.row);

   Matrix res(a.row, b.col);

   elemType rowFactor[a.row];
   elemType colFactor[b.col];
 // f1 =  2 + 6*n + 15*n*m / 2
       for(int i = 0; i < a.row; i++) { // 2 + a.row(2 + 4 + 15*a.col / 2)
           rowFactor[i] = 0; //2
           for(int j = 0; j < a.col / 2; j++) { // 2 + (a.col/2)(3 + 12) = 2 + 15*a.col / 2
               rowFactor[i] = rowFactor[i] + a.arr[i][2 * j + 1] * a.arr[i][2 * j]; // 12
           }
       }
     // f2 =  2 + 6*k + 15*m*k / 2
       for(int i = 0; i < b.col; i++) {  //2 + 6*b.col + 15*a.col*b.col / 2
           colFactor[i] = 0;
           for(int j = 0; j < a.col / 2; j++) {
               colFactor[i] = colFactor[i] + b.arr[2 * j + 1][i] * b.arr[2 * j][i];
           }
       }
    // f3 = 2 + 2n + 11nk + 26nmk/2
       for(int i = 0; i < a.row; i++) { //2 + a.row(2 + 11*b.col + 26*a.col*b.col/2)
           for(int j = 0; j < b.col; j++) { //2 + b.col*(11 + 26*a.col/2)
               res.arr[i][j] = -rowFactor[i] - colFactor[j]; //7
               for(int k = 0; k < a.col / 2; k++) { // 2 + (a.col/2)(3 + 23)
                   res.arr[i][j] = res.arr[i][j] + (a.arr[i][2*k+1] + b.arr[2*k][j]) *
                                                  (a.arr[i][2*k] + b.arr[2*k+1][j]); //23
               }
           }
       }
       //f4 = 2 - если m четное
```

```
        //f4 = 4 + 4n + 15nk - если m нечетное
        if(a.col % 2 == 1) { //4 + a.row(4 + 15*b.col)
            for(int i = 0; i < a.row; i++) //2 + a.row(4 + 15*b.col)
                for(int j = 0; j < b.col; j++) //2 + b.col(2 + 13)
                    res.arr[i][j] = res.arr[i][j] + a.arr[i][a.col-1] * b.arr[a.col-1][j]; //13
        }

        return res;
    }
```

3. Теоретическая оценка

Если m четное (лучший случай):

$f_л$ = 2 + 6*n + 15*n*m / 2 + 2 + 6*k + 15*m*k / 2 + 2 + 2n + 11nk + 13nmk + 2 = 8 + 8n + 6k + 7.5m(n+k) + 11nk + 13nmk

Если m нечетное (худший случай) (int)m / 2 = (m - 1) / 2:

$f_х$ = 2 + 6*n + 15*n*(m-1) / 2 + 2 + 6*k + 15*(m-1)*k / 2 + 2 + 2n + 11nk + 26nk(m-1)/2 + 2 + 4n+15nk = 8 + 12n + 6k + (n+k)15(m-1)/2 + 26nk + 13nmk - 13nk = 8 + 4.5n - 1.5k + 7.5m(n+k) + 13nk + 13nmk

Трудоемкость в среднем:

f = ($f_л$ + $f_х$) / 2 = 8 + 6.25n - 2.25k + 7.5m(n+k) + 12nk + 13nmk

Модифицированный алгоритм Винограда

1. Описание

Модифицируем алгоритм Винограда:

- используем составное присваивание
- добавим переменную-буфер
- заранее вычисляем индексы и выражения, которые часто используются
- перенос цикла последнего цикла внутрь основного цикла для нечетных размерностей

2. Реализация

```
Matrix multModVinograd(const Matrix &a, const Matrix &b)
{
    assert(a.col == b.row);

    int ind;
    elemType tmp;
    Matrix res(a.row, b.col);

    elemType rowFactor[a.row];
        elemType colFactor[b.col];

        //int mid = a.col / 2;

        int last = a.col - 1; //2
```

```

    //f1 = 4 + 9n + 9k
    for(int i = 0; i < a.row; i++) { //2 + a.row*(2+7)
        rowFactor[i] = a.arr[i][0] * a.arr[i][1]; //7
    }
    for(int i = 0; i < b.col; i++) {
        colFactor[i] = b.arr[0][i] * b.arr[1][i];
    }

    //f2 = 2 + mid(8 + 9n + 9k) - (8 + 9n + 9k)
    //f2 = mid(8 + 9n + 9k) - 9n - 9k - 6
    for(int j = 2; j < last; j += 2) { //2 + (mid-1)(8 + 9n + 9k)
        ind = j + 1; //2
        for(int i = 0; i < a.row; i++) { //2 + n(7 + 2)
            rowFactor[i] += a.arr[i][j] * a.arr[i][ind]; // 7
        }
        for(int i = 0; i < b.col; i++) {
            colFactor[i] += b.arr[j][i] * b.arr[ind][i];
        }
    }
    //f1 + f2 = mid(8 + 9n + 9k)

    //f3(bad)  = 1 + 2 + n(4 + k(18 + 16mid)) = 3 + 4n + 18nk + 16nk*mid
    //f3(good) = 1 + 2 + n(4 + k(12 + 16mid)) = 3 + 4n + 12nk + 16nk*mid
    if(a.col % 2) { //1
        for(int i = 0; i < a.row; i++) { // 2 + n(4 + k(18 + 16mid))
            for(int j = 0; j < b.col; j++) { //2 + k(18 + 16mid)
                tmp = -(rowFactor[i] + colFactor[j]); //5
                for(int k = 0; k < last; k += 2) { //2 + 16mid
                    tmp += (a.arr[i][k+1] + b.arr[k][j]) *
                           (a.arr[i][k]   + b.arr[k+1][j]); //14
                }
                tmp += a.arr[i][last] * b.arr[last][j]; //6
                res.arr[i][j] = tmp; //3
            }
        }
    } else {
        for(int i = 0; i < a.row; i++) { // 2 + n(4 + k(12 + 16mid))
            for(int j = 0; j < b.col; j++) { //2 + k(12 + 16mid)
                tmp = -(rowFactor[i] + colFactor[j]); //5
                for(int k = 0; k < last; k += 2) { //2 + 16mid
                    tmp += (a.arr[i][k+1] + b.arr[k][j]) *
                           (a.arr[i][k]   + b.arr[k+1][j]); //14
                }
                res.arr[i][j] = tmp; // 3
            }
        }
    }

```

```
64.     return res;
65. }
```

3. Теоретическая оценка

Если m четное (лучший случай) mid = m/2:

$f_л$ = mid(8 + 9n + 9k) + 3 + 4n + 12nk + 16nk*mid
= 3 + 4n + 12nk + mid(8 + 9n + 9k + 16nk)
= 3 + 4n + 12nk + 0.5m(8 + 9n + 9k + 16nk)
= 3 + 4n + 12nk + 4m + 4.5nm + 4.5mk + 8mnk

Если m нечетное (худший случай) mid = (m - 1) / 2:

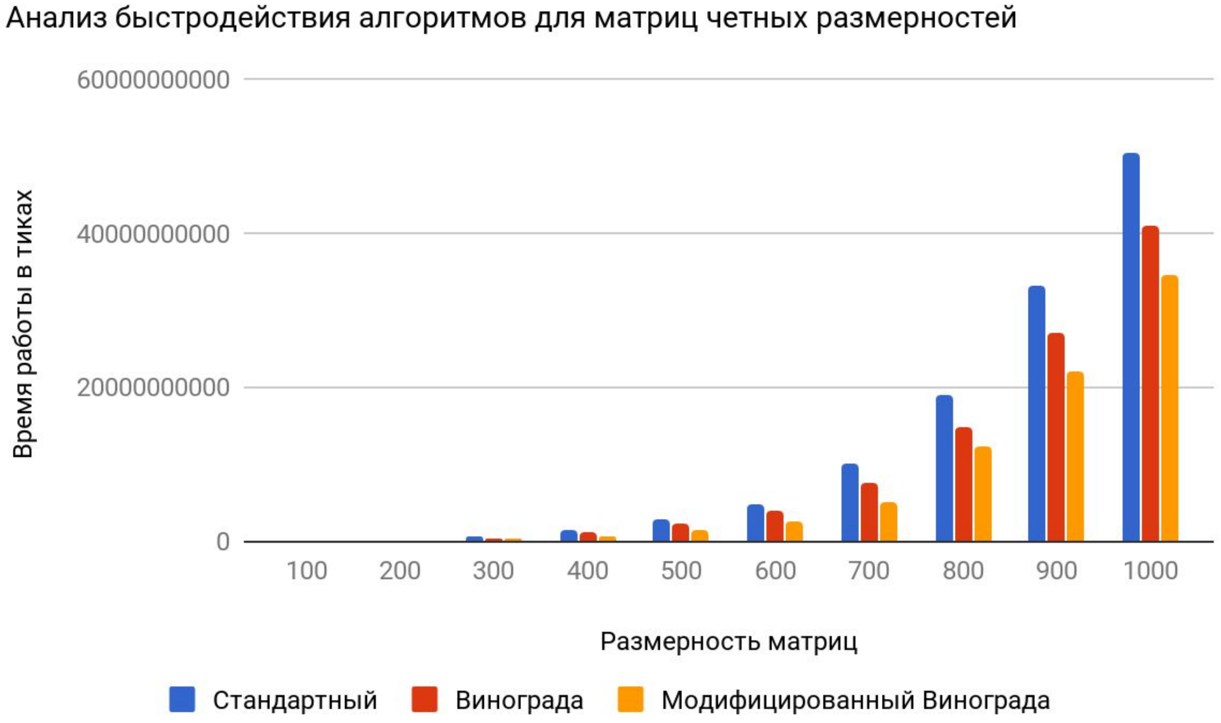
f_x = mid(8 + 9n + 9k) + 3 + 4n + 18nk + 16nk*mid
= 3 + 4n + 18nk + mid(8 + 9n + 9k + 16nk)
= 3 + 4n + 18nk + (m-1)/2(8 + 9n + 9k + 16nk)
= 3 + 4n + 18nk + 4m + 4.5nm + 4.5mk + 8mnk - 4 - 4.5n - 4.5k - 8nk
= -1 + 0.75n + 10nk - 2.25k + 4m + 4.5nm + 4.5mk + 8mnk

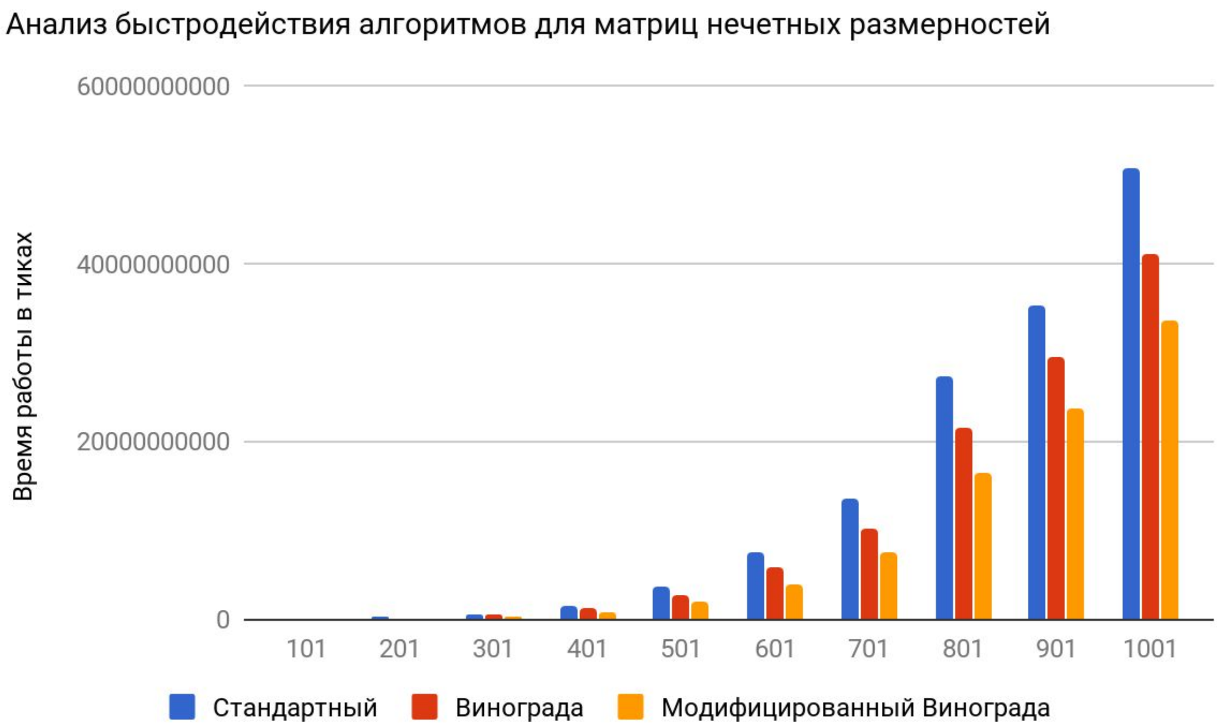
Трудоемкость в среднем:

f = ($f_л$ + f_x) / 2 = 1 + 2.375n + 11nk + 0.425n + 4m + 4.5nm + 4.5mk + 8mnk

Сравнение алгоритмов

Для сравнения алгоритмов было посчитано время работы для матриц размерностью 100×100, 200×200, …, 1000×1000 и 101×101, 202×202, …, 1001×1001.





Проанализировав построенные графики, можно сделать выводы:

- модифицированный алгоритм Винограда работает быстрее чем алгоритм Винограда, а алгоритм Винограда работает быстрее чем стандартный алгоритм.
- чем больше размерность матриц, тем больший выигрыш по времени дает алгоритм Винограда по сравнению со стандартным алгоритмом.
- результаты теоретических оценок и результаты эксперимента не совпадают, так как в случае алгоритма Винограда компилятор оптимизирует большинство одинаковых вычислений в цикле, что позволяет достичь трудоемкости O(n,m,k)=9nmk. Тогда как реализация стандартного и модифицированного алгоритма написана уже примерно с учетом возможных оптимизаций.

Заключение

Во время выполнения работы были изучены и реализованы алгоритмы умножения матриц: стандартный и алгоритм Винограда, а также модифицированный алгоритм Винограда. Было произведено сравнение этих методов.