Содержание

Введение

Цель работы – провести сравнительный анализ эффективности различных алгоритмов умножения матриц: стандартного алгоритма, алгоритма Винограда и оптимизированного алгоритма Винограда. Также:

1. Замерить процессорное время.
2. Рассчитать трудоёмкость алгоритмов.
3. Оценить затраты по памяти.

Задачи:

1. Описать стандартный алгоритм умножения матриц.
2. Реализовать оптимизированный алгоритм Винограда для умножения матриц – замена умножение на два на двоичный сдвиг целого числа.
3. Реализовать оптимизированный алгоритм Винограда для умножения матриц – замена умножение на два на двоичный сдвиг целого числа.
4. Реализовать оптимизированный алгоритм Винограда для умножения матриц – включение части IV алгоритма в часть II.
5. Рассчитать трудоёмкость алгоритмов.
6. Оценить затраты памяти алгоритмов.
7. Выполнить замеры процессорного времени реализации всех алгоритмов.

1. Описание алгоритмов

1.1 Стандартный алгоритм умножения

Стандартный алгоритм умножения матриц представляет собой последовательность действий, выполненных с помощью трех вложенных циклов. Этот процесс начинается с итерации по строкам матрицы A[m × n].

В первом цикле мы проходим по каждой строке матрицы A, где переменная i принимает значения от 1 до m. Для каждой строки матрицы запускается второй цикл, который проходит по каждому столбцу матрицы B[p × q]. Переменная j в этом цикле принимает значения от 1 до q.

Третий вложенный цикл выполняет вычисление скалярного произведения i-й строки матрицы A и j-го столбца матрицы B. Переменная k в этом цикле принимает значения от 1 до n (что равно p согласно условию совпадения размерностей матриц для умножения). Внутри третьего цикла вычисляются произведения соответствующих элементов A_{ik} и B_{kj}, которые затем суммируются для получения элемента C_{ij} результирующей матрицы C. Листинг 1 показывает программную реализацию.

Листинг 1. Программная реализация стандартного алгоритма умножения матриц

```
vector<vector<int>> multiplyMatrixDefault(const
vector<vector<int>>& a, const vector<vector<int>>& b) {
    int l = a.size();
    int m = a[0].size();
    int n = b[0].size();

    vector<vector<int>> c(l, vector<int>(n, 0));

    for (int i = 0; i < l; i++) {
        for (int j = 0; j < n; j++) {
            for (int r = 0; r < m; r++) {
                c[i][j] += a[i][r] * b[r][j];
            }
        }
    }

    return c;
}
```

Формально, результат умножения матриц можно записать следующей формулой:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

1.2 Алгоритм Винограда

Алгоритм Винограда уменьшает количество умножений за счет предварительной обработки строк и столбцов матриц. Этот алгоритм состоит из нескольких ключевых этапов, которые позволяют значительно снизить количество необходимых умножений.

Первым этапом является расчет массива rowFactor, который хранит сумму произведений пар элементов для каждой строки матрицы A. Для каждой строки i матрицы A вычисляется сумма произведений элементов, стоящих на четных и следующих за ними нечетных позициях. Формально это можно записать как:

$$\text{rowFactor}[i] = \sum_{k=0}^{n/2} A_{i,2k} \cdot A_{i,2k+1}$$

Этот этап позволяет сократить число умножений на 50% для четных размеров.

Аналогично, для каждого столбца j матрицы B вычисляется сумма произведений пар элементов, стоящих на четных и следующих за ними нечетных позициях. Формально это можно записать как:

$$\text{colFactor}[j] = \sum_{k=0}^{p/2} B_{2k,j} \cdot B_{2k+1,j}$$

Третьим этапом является вычисление элементов результирующей матрицы C с использованием ранее вычисленных массивов rowFactor и colFactor. Для каждого элемента C_{ij} матрицы C применяется следующая формула:

$$C_{ij} = (\text{rowFactor}[i] + \text{colFactor}[j]) - \sum_{k=0}^{n/2} A_{i,2k} \cdot B_{2k,j} + A_{i,2k+1} \cdot B_{2k+1,j}$$

Если размер n матрицы A (и соответственно p матрицы B) нечетен, необходимо добавить дополнительный член к результату. Для каждого элемента C_{ij} добавляется произведение последних элементов соответствующей строки и столбца:

$$C_{ij} += A_{i,n-1} \cdot B_{p-1,j}$$

Этот шаг гарантирует корректное вычисление результата даже при нечетном размере матриц.

Для четного n формула вычисления элемента C_{ij} можно записать следующим образом:

$$C_{ij} = \sum_{k=0}^{n/2} (u_{2k} + w_{2k+1})(u_{2k+1} + w_{2k}) - \sum_{k=0}^{n/2} u_{2k}\, u_{2k+1} - \sum w_{2k}\, w_{2k+1}$$

где u_{2k} и w_{2k} — элементы строк матрицы A и столбцов матрицы B соответственно. В листинге 2 показано, как программно реализовано умножение матриц по алгоритму Винограда.

Листинг 2. Реализация функции multiplyMatrixVinograd

```
vector<vector<int>> multiplyMatrixVinograd(const
vector<vector<int>>& a, const vector<vector<int>>& b) {
    int n = a.size();
    int m = a[0].size();
    int k = b[0].size();

    vector<int> rowFactor(n, 0);
    vector<int> colFactor(k, 0);
    vector<vector<int>> c(n, vector<int>(k, 0));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m / 2; ++j) {
            rowFactor[i] += a[i][2 * j] * a[i][2 * j + 1];
        }
    }

    for (int i = 0; i < k; ++i) {
        for (int j = 0; j < m / 2; ++j) {
            colFactor[i] += b[2 * j][i] * b[2 * j + 1][i];
        }
    }

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < k; ++j) {
            c[i][j] = -(rowFactor[i] + colFactor[j]);
            for (int l = 0; l < m / 2; ++l) {
                c[i][j] += (a[i][2 * l] + b[2 * l + 1][j]) *
(a[i][2 * l + 1] + b[2 * l][j]);
            }
        }
    }

    if (m % 2 == 1) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < k; ++j) {
                c[i][j] += a[i][m - 1] * b[m - 1][j];
            }
        }
    }

    return c;
}
```

Выводы:

1) алгоритм Винограда уменьшает количество умножений с $O(m \times p \times q)$ до $O(m \times p \times {}^{n}/_{2})$;

2) но требует дополнительной памяти для хранения массивов rowFactor и colFactor.

Таким образом, алгоритм Винограда обеспечивает более эффективное умножение матриц за счет предварительной обработки данных и сокращения количества умножений.

1.3 Оптимизированный алгоритм – замена умножения на 2 на двоичный сдвиг и интеграция обработки нечётных элементов

Оптимизированный алгоритм Винограда основан на классическом варианте, но включает дополнительные улучшения, направленные на снижение временной сложности и минимизацию накладных расходов. Оптимизации включают:

1) замену арифметических операций на битовые сдвиги:
   вместо операции умножения на 2 (2 * k) используется двоичный сдвиг (k << 1), что ускоряет вычисления, так как сдвиг битов выполняется быстрее умножения. Аналогично, в некоторых случаях вместо деления на 2 применяется сдвиг вправо (k >> 1);
2) интеграция этапа обработки нечётных элементов в основной цикл:
   этап IV (обработка нечётного числа элементов n) объединяется с основным этапом III. Это позволяет избежать дополнительных циклов и упростить код, включив обработку нечётных случаев (когда n не делится на 2) в основной цикл через дополнительные проверки.

В листинге 3 приведена программная реализация оптимизированного алгоритма Винограда.

Листинг 3. Реализация оптимизированной функции Винограда

```
vector<vector<int>> multiplyMatrixVinogradWithShift(const
vector<vector<int>>& a, const vector<vector<int>>& b) {
    int n = a.size();
    int m = a[0].size();
    int k = b[0].size();

    vector<int> rowFactor(n, 0);
    vector<int> colFactor(k, 0);
    vector<vector<int>> c(n, vector<int>(k, 0));

    for (int i = 0; i < n; ++i) {
        for (int j = 1; j < m; j += 2) {
            rowFactor[i] += a[i][j] * a[i][j - 1];
        }
```

```
    }

    for (int i = 0; i < k; ++i) {
        for (int j = 1; j < m; j += 2) {
            colFactor[i] += b[j][i] * b[j - 1][i];
        }
    }

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < k; ++j) {
            c[i][j] = -(rowFactor[i] + colFactor[j]);
            for (int l = 1; l < m; l += 2) {
                c[i][j] += (a[i][l - 1] + b[l][j]) * (a[i][l] + b[l - 1][j]);
            }
        }
    }

    if (m % 2 == 1) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < k; ++j) {
                c[i][j] += a[i][m - 1] * b[m - 1][j];
            }
        }
    }

    return c;
}
```

2. Средства реализации и архитектура

Технические характеристики устройства, на котором выполнялись замеры:

1. Операционная система Windows 11.
2. Оперативная память 32 ГБ.
3. Процессор Intel(R) Core(TM) i7-7600U @ 2.80 ГГц , архитектура x64.
4. Компилятор MinGW.
5. Среда разработки Clion.
6. Язык программирования: C++.

Условия тестирования:

1. Ноутбук подключен к сети электропитания;
2. Система нагружена только средой разработки CLion и тестируемой программой;
3. std::chrono::high_resolution_clock (Windows использует QueryPerformanceCounter на уровне компилятора gc++) [2].

3. Сложности реализованных алгоритмов

3.1 Временная сложность

Стандартный алгоритм умножения матриц основан на тройном вложении циклов. Его временная сложность определяется следующими компонентами:

1) внешний цикл по строкам матрицы A: 2 + m * (2 + $f_{тела}$), где $f_{тела}$ — трудоемкость внутренних циклов;

2) средний цикл по столбцам матрицы B: 2 + p * (2 + $f_{тела}$);

3) внутренний цикл по элементам векторов: 2 + 10 * K, где K = $^{n}/_{2}$ (так как операция умножения оценивается в 2 единицы).

Объединяя все компоненты, общая трудоемкость стандартного алгоритма:

$$f_{стд} = 2 + 4 \times m + 4 \times m \times p + 10 \times m \times q \times K \approx O(m \times p \times K)$$

Алгоритм Винограда снижает количество умножений за счет предварительной обработки данных, но требует дополнительных сложений. Его трудоемкость включает:

1) инициализация массивов rowFactor и colFactor: m + n;

2) расчет rowFactor: для каждой строки i:

$$f_{row} = 2 + \frac{n}{2} \times (2 + 11 \times m)$$

общая трудоемкость: $f_{row} \times m$;

3) расчет colFactor: аналогично для столбцов j:

$$f_{col} = 2 + \frac{n}{2} \times (2 + 11 \times q)$$

общая трудоемкость: $f_{col} \times 1$;

4) основной цикл:

$$f_{цикл} = 2 + m \times \left(4 + q \times \left(11 + \frac{K}{2} \times 23\right)\right)$$

5) обработка нечетного члена:

$$f_{last} \begin{cases} 2 \\ 4 + m \cdot (4 + 14 \times q) \end{cases}$$

Это вдвое эффективнее стандартного алгоритма при $K = \frac{N_A}{2}$, так как основная сложность сокращается до $O\left(m \times q \times {}^{n}/_{2}\right)$.

Оптимизации (битовые сдвиги, объединение циклов) снижают константы в формулах:

1) инициализация массивов rowFactor и colFactor: m + n;

2) расчет rowFactor: для каждой строки i:

$$f_{row} = 2 + \frac{n}{2} \times (2 + 8 \times m)$$

общая трудоемкость: $f_{row} \times m$;

3) расчет colFactor: аналогично для столбцов j:

$$f_{col} = 2 + \frac{n}{2} \times (2 + 8 \times q)$$

общая трудоемкость: $f_{col} \times 1$;

4) основной цикл:

$$f_{цикл} = 2 + m \times \left(4 + q \times \left(11 + \frac{K}{2} \times 18\right)\right)$$

5) обработка нечетного члена:

$$f_{last} \begin{cases} 1 \\ 4 + m \cdot (4 + 10 \times q) \end{cases}$$

Это на 23% эффективнее базового алгоритма Винограда.

В таблице 1 представлены замеры процессорного времени работы реализации всех трех алгоритмов.

Таблица 1. Замеры процессорного времени в тактах

Размер	Тип	Стандартный	Виноград	Оптимизированный
10	best	0.000097	0.000029	0.000070
100	best	0.038226	0.030149	0.034413
200	best	0.233554	0.467542	0.279101
300	best	0.869784	0.396789	0.323317
400	best	0.804498	0.694902	0.690929
500	best	1.780230	1.405530	1.425760
11	worst	0.000033	0.000032	0.000031
101	worst	0.011333	0.010325	0.010144
201	worst	0.094988	0.079141	0.088006
301	worst	0.336416	0.279232	0.313079
401	worst	0.835309	0.727166	0.715785
501	worst	1.784020	1.540560	1.598770

На рисунке 1 приведены графики сравнения времени работы программ для всех трех алгоритмов для квадратной матрицы с четным N (best case) (такты процессора переведены в секунды).

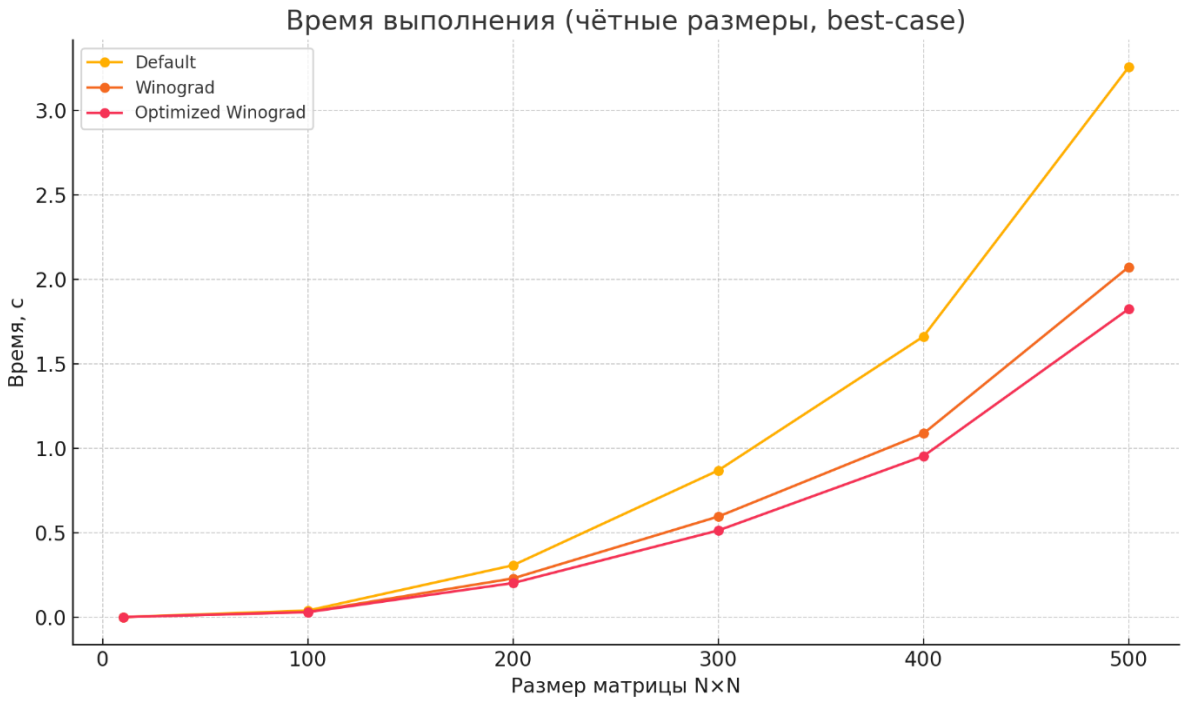

Рис. 1.

На рисунке 2 приведены графики уже для нечетного N (worst case).

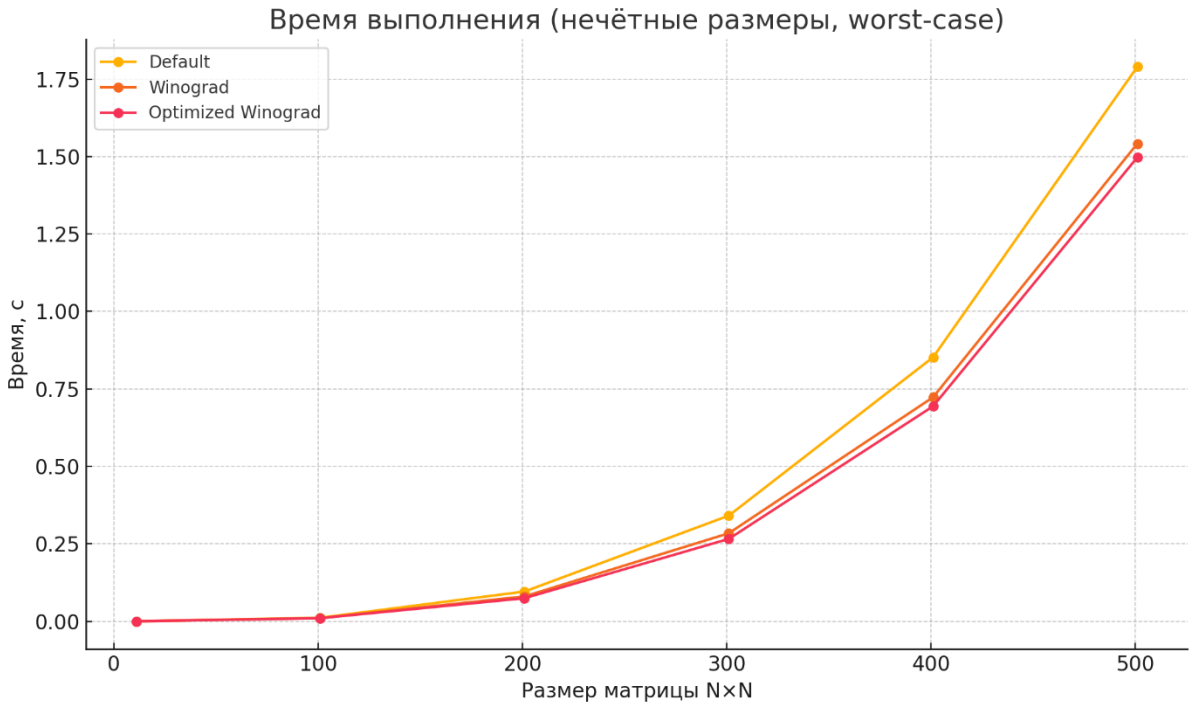

Рис. 2.

3.2 Занимаемая память

Стандартный алгоритм требует минимального объема памяти:

$$O(m \times n) + O(p \times q) + O(x \times y)$$

Матрицы A, B и C хранятся в памяти без дополнительных структур.

Алгоритм Винограда требует дополнительной памяти для хранения предварительных вычислений:

1) массивы rowFactor и colFactor:

$$\text{rowFactor: } O\left(M_A \times \frac{N_A}{2}\right)$$

$$\text{colFactor: } O\left(N_B \times \frac{N_A}{2}\right)$$

2) общая память:

$$O(m \times n) + O(p \times q) + O(x \times y) + O\left(m \times \frac{n}{2} + q \times \frac{n}{2}\right)$$

Оптимизации снижают память за счет:

1) сокращения размера массивов за счет битовых операций;

2) интеграции этапов III и IV, уменьшая количество дополнительных циклов.

Однако общая потребность остается выше, чем у стандартного алгоритма:

$$O(m \times n) + O(p \times q) + O(x \times y) + O\left(\frac{m \times n}{2} + \frac{p \times q}{2}\right)$$

Заключение

В ходе выполнения лабораторной работы были реализованы три алгоритма умножения матриц: классический алгоритм, алгоритм Винограда и оптимизированная версия алгоритма Винограда. Проведённый теоретический анализ показал, что алгоритм Винограда снижает количество умножений практически вдвое по сравнению со стандартным алгоритмом, а оптимизация уменьшает накладные расходы за счёт использования битовых операций и интеграции этапов обработки нечётных элементов. Результаты экспериментальных замеров подтвердили теоретические выкладки:

1) алгоритм Винограда демонстрирует выигрыш по времени на средних и больших размерностях матриц в чётных случаях;
2) оптимизированная версия работает ещё быстрее за счёт сокращения количества арифметических операций;
3) худшие случаи (нечётные размеры) показывают, что эффективность алгоритмов Винограда падает, но остаётся выше базового алгоритма.

Список использованных источников

1. Умножение матриц // Коллекция алгоритмов [Электронный ресурс] URL: http://algolib.narod.ru/Math/Matrix.html (дата обращения: 03.05.2025).
2. QueryPerformanceCounter function (Windows) // документация Microsoft по Win32 API [Электронный ресурс] URL: https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter (дата обращения: 11.04.2025)