МГТУ им. Баумана

Лабораторная работа №2

По курсу: "Анализ алгоритмов"

Алгоритмы умножения матриц

Работу выполнила: Лаврова Анастасия, ИУ7-55Б

Преподаватели: Волкова Л.Л., Строганов Ю.В.

Москва, 2019

Оглавление

Введение

Цель работы: изучение алгоритмов умножения матриц. В данной лабораторной работе рассматривается стандартный алгоритм умножения матриц, алгоритм Винограда и модифицированный алгоритм Винограда. Также требуется изучить рассчет сложности алгоритмов, получить навыки в улучшении алгоритмов. Эти алгоритмы активно применяются во всех областях, применяющих линейную алгебру, таких как:

- компьютерная графика
- физика
- экономика

и так далее.

В ходе лабораторной работы предстоит:

- Изучить алгоритмы умножения матриц: стандартный и алгоритм Винограда
- Улучшить алгоритм Винограда
- Дать теоретическую оценку базового алгоритма умножения матриц, алгоритма Винограда и улучшенного алгоритма Винограда
- Реализовать три алгоритма умножения матриц на одном из языков программирования
- Сравнить алгоритмы умножения матриц

1 Аналитическая часть

Матрица - математический объект, эквивалентный двумерному массиву. Числа располагаются в матрице по строкам и столбцам. Если число столбцов в первой матрице совпадает с числом строк во второй, то эти две матрицы можно перемножить. У произведения будет столько же строк, сколько в первой матрице, и столько же столбцов, сколько во второй.

1.1 Классический алгоритм умножения матриц

Пусть даны две прямоугольные матрицы A и B размерности m на n и n на l соответсвенно:

$$\begin{bmatrix} a_{1,1} & ... & a_{1,n} \\ ... & ... & ... \\ a_{m,1} & ... & a_{m,n} \end{bmatrix}$$

$$\begin{bmatrix} b_{1,1} & ... & b_{1,l} \\ ... & ... & ... \\ b_{n,1} & ... & b_{n,l} \end{bmatrix}$$

В результате получим матрицу C размерности m на l:

$$\begin{bmatrix} c_{1,1} & ... & c_{1,l} \\ ... & ... & ... \\ c_{m,1} & ... & c_{m,l} \end{bmatrix}$$

$c_{i,j} = \sum\limits_{r=1}^{n} a_{i,r} \cdot b_{r,j}$ называется произведением матриц A и B.

1.2 Алгоритм Винограда

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

$V \cdot W = v_1 \cdot w_1 + v_2 \cdot w_2 + v_3 \cdot w_3 + v_4 \cdot w_4$

Это равенство можно переписать в виде:

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

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. Это означает, что над предварительно обработанными элементами нам придется выполнять лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения.

1.2.1 Вывод

Были рассмотрены поверхностно алгоритмы классического умножения матриц и алгоритм Винограда, основная разница которого — наличие предварительной обработки, а также уменьшение количества операций умножения.

2 Конструкторская часть

Требования к вводу:
На вход подаются две матрицы и их размерности
Требования к программе:
Корректное умножение двух матриц

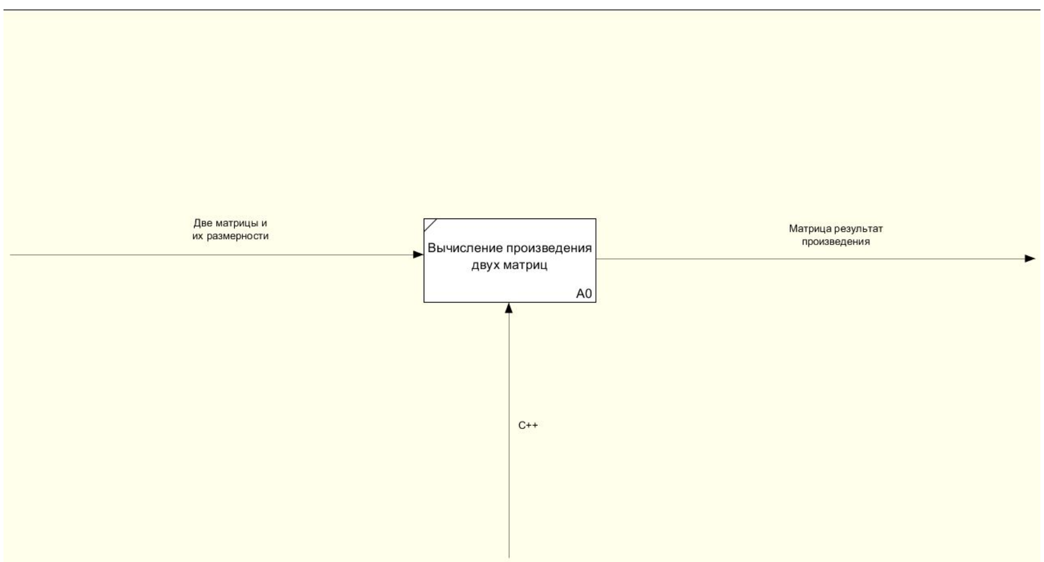

Рис. 2.1: IDEF0-диаграмма, описывающая алгоритм нахождения произведения двух матриц

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов.

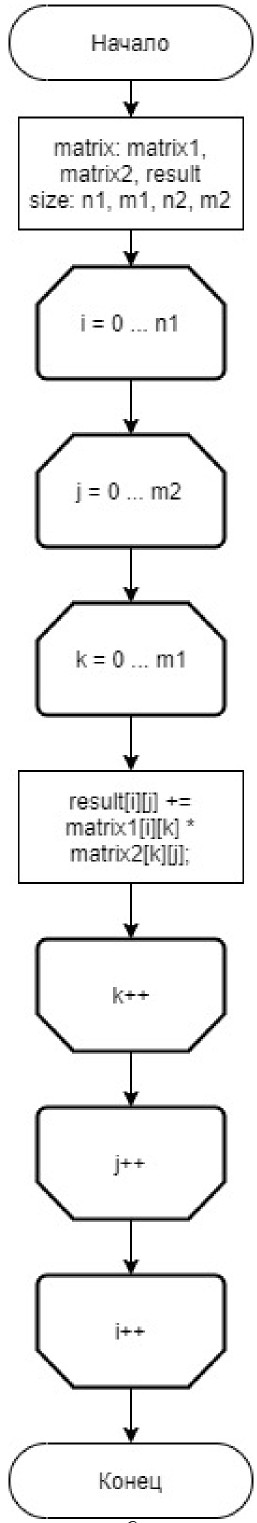

Рис. 2.2: Схема классического алгоритма умножения матриц

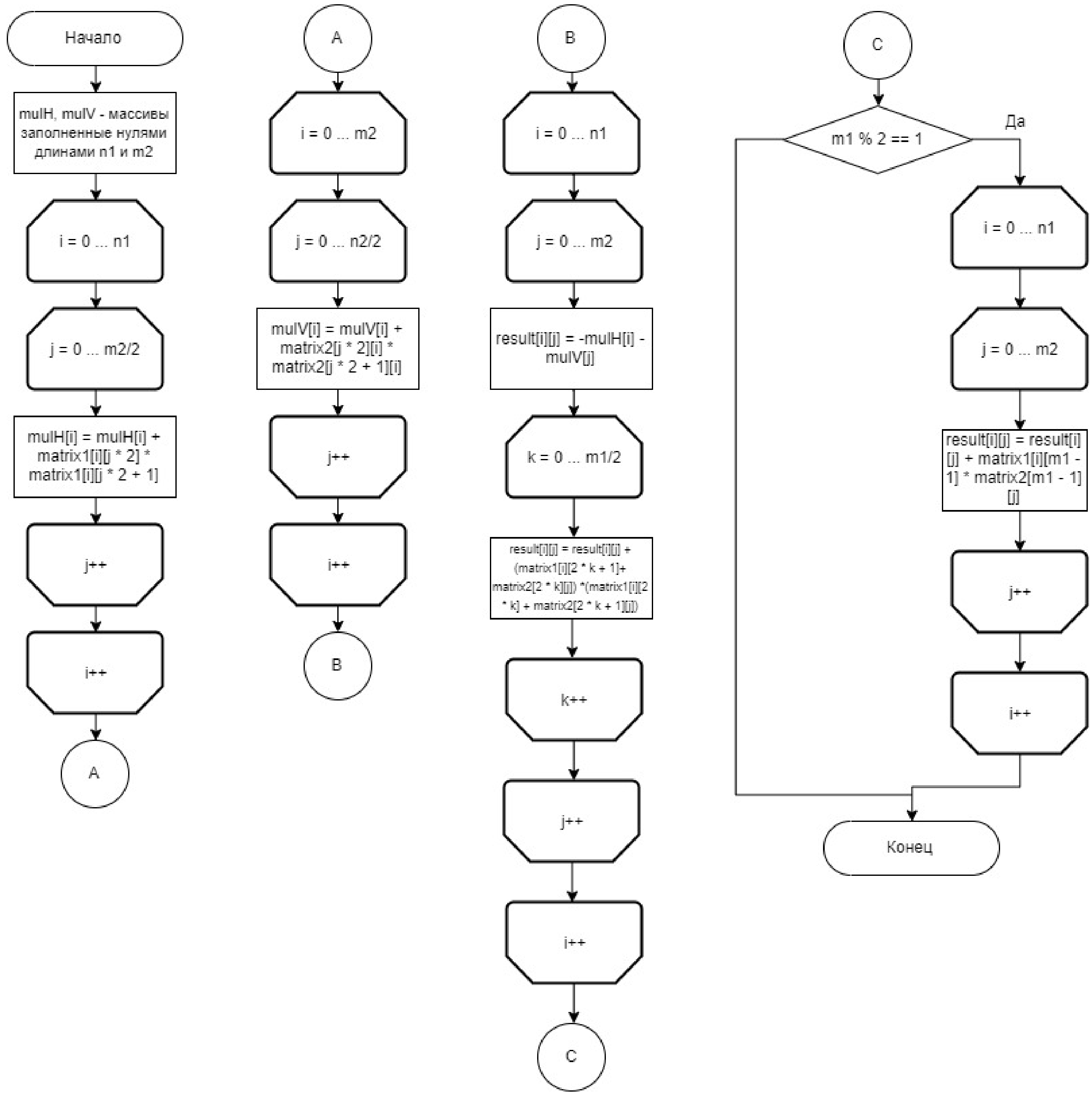

Рис. 2.3: Схема алгоритма Винограда

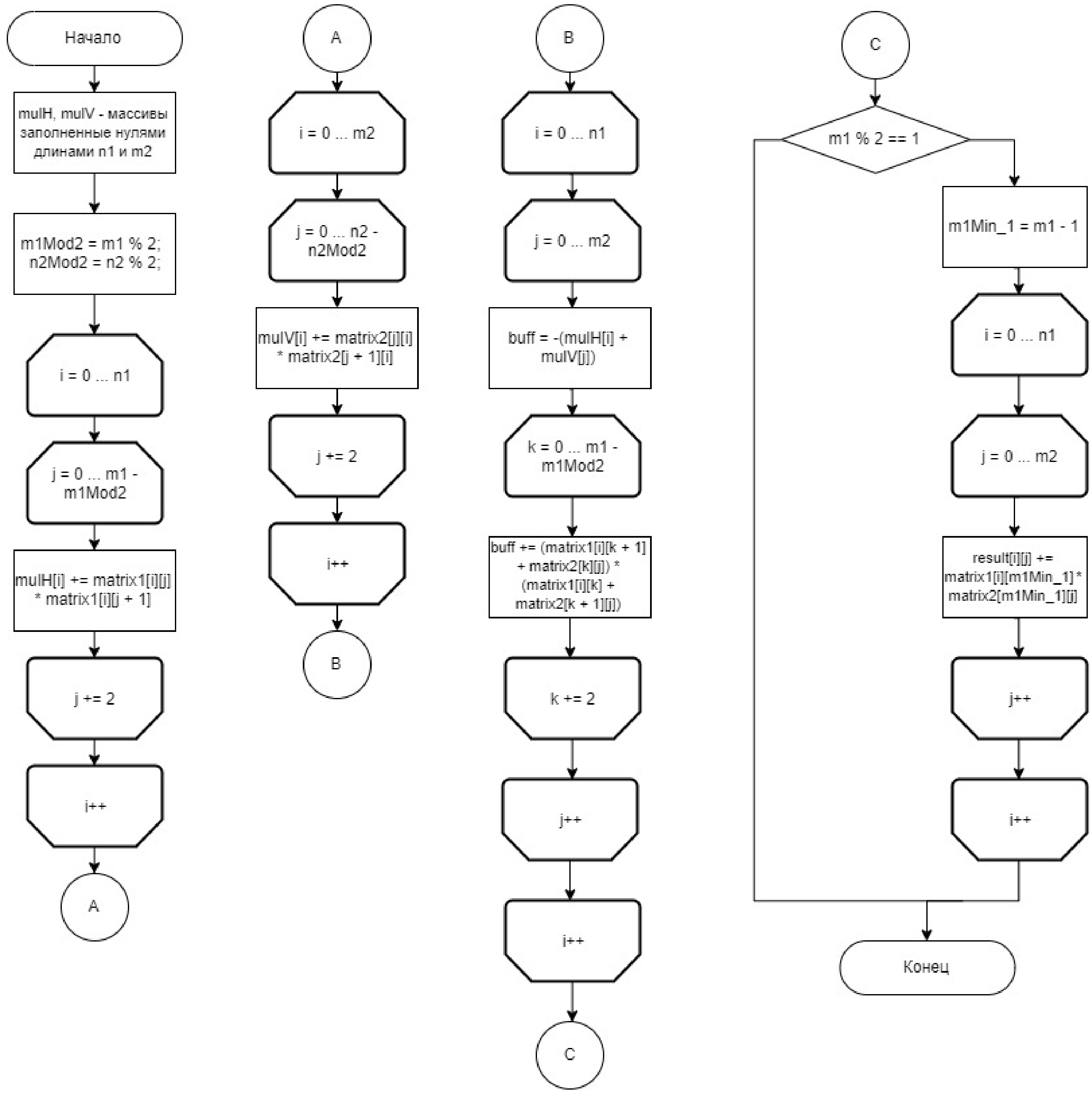

Рис. 2.4: Схема оптимизированного алгоритма Винограда

2.2 Трудоемкость алгоритмов

Введем модель трудоемкости для оценки алгоритмов:

- базовые операции стоимостью 1 — +, -, *, /, =, ==, <=, >=, !=, +=, []
- оценка трудоемкости цикла for от 0 до N с шагом 1 $F_{for} = 2 + N \cdot (2 + F_{body})$, где F_{body} - тело цикла
- стоимость условного перехода применим за 0, стоимость вычисления условия остаётся

Оценим трудоемкость алгоритмов по коду программы

2.2.1 Классический алгоритм

Рассмотрим трудоемкость классического алгоритма:

$10MNQ + 4MQ + 4M + 2$

2.2.2 Алгоритм Винограда

Рассмотрим трудоемкость алгоритма Винограда:

Трудоемкость алгоритма Винограда:

Первый цикл: $15/2 \cdot MN + 5 \cdot M + 2$

Второй цикл: $15/2 \cdot MN + 5 \cdot M + 2$

Третий цикл: $13 \cdot MNQ + 12 \cdot MQ + 4 \cdot M + 2$

Условный переход: $\begin{bmatrix} 2 & \text{, в случае невыполнения условия} \\ 15 \cdot QM + 4 \cdot M + 2 & \text{, в случае выполнения условия} \end{bmatrix}$

Итого: $15/2 \cdot MN + 5 \cdot M + 2 + 15/2 \cdot MN + 5 \cdot M + 2 + 13 \cdot MNQ + 12 \cdot MQ + 4 \cdot M + 2 + \begin{bmatrix} 2 & \text{, в случае невыполнения условия} \\ 15 \cdot QM + 4 \cdot M + 2 & \text{, в случае выполнения условия} \end{bmatrix}$

2.2.3 Оптимизированный алгоритм Винограда

Рассмотрим трудоемкость алгоритма Винограда:

Трудоемкость алгоритма Винограда:

Первый цикл: $11/2 \cdot MN + 4 \cdot M + 2$

Второй цикл: $11/2 \cdot MN + 4 \cdot M + 2$

Третий цикл: $15/2 \cdot MNQ + 9 \cdot MQ + 4 \cdot M + 2$

Условный переход: $\begin{bmatrix} 1 & \text{, в случае невыполнения условия} \\ 10 \cdot QM + 4 \cdot M + 2 & \text{, в случае выполнения условия} \end{bmatrix}$

Итого: $11/2 \cdot MN + 4 \cdot M + 2 + 11/2 \cdot MN + 4 \cdot M + 2 + 15/2 \cdot MNQ + 9 \cdot MQ + 4 \cdot M + 2 + \begin{bmatrix} 1 & \text{, в случае невыполнения условия} \\ 10 \cdot QM + 4 \cdot M + 2 & \text{, в случае выполнения условия} \end{bmatrix}$

3 Технологическая часть

3.1 Выбор ЯП

Для реализации программ я выбрала язык программирования C++, так имею большой опыт работы с ним. Среда разработки - Visual Studio.

Для замера процессорного времени используется функция, возвращающая количество тиков.

Листинг 3.1: Функция получения тиков

```
unsigned long long getTicks(void)
{
    unsigned long long d;
    __asm__ __volatile__ ("rdtsc" : "=A" (d) );
    return d;
}
```

3.2 Реализация алгоритма

Листинг 3.2: Функция классического умножения матриц

```
int** mult_m(int** matrix1, int n1, int m1, int** matrix2,
   int n2, int m2)
{
  int** result = NULL;
  if (matrix1 && matrix2 && m1 == n2)
  {
    result = new_m(n1, m2);
```

```
    for (int i = 0; i < n1; i++)
      for (int j = 0; j < m2; j++)
        for (int k = 0; k < m1; k++)
          result[i][j] += matrix1[i][k] * matrix2[k][j];
  }
  return result;
}
```

Листинг 3.3: Алгоритм Винограда

```
int** Vinograd(int** matrix1, int n1, int m1, int** matrix2
    , int n2, int m2)
{
  int* mulH = (int*)malloc(n1 * sizeof(int));
  for (int i = 0; i < n1; i++)
    mulH[i] = 0;

  int* mulV = (int*)malloc(m2 * sizeof(int));
  for (int i = 0; i < m2; i++)
    mulV[i] = 0;

  int** result = new_m(n1, m2);

  for (int i = 0; i < n1; i++)
  {
    for (int j = 0; j < m1 / 2; j++)
    {
      mulH[i] = mulH[i] + matrix1[i][j * 2] * matrix1[i][j
          * 2 + 1];
    }
  }

  for (int i = 0; i < m2; i++)
  {
    for (int j = 0; j < n2 / 2; j++)
    {
      mulV[i] = mulV[i] + matrix2[j * 2][i] * matrix2[j * 2
          + 1][i];
    }
  }

```

```
    for (int i = 0; i < n1; i++)
    {
        for (int j = 0; j < m2; j++)
        {
            result[i][j] = -mulH[i] - mulV[j];
            for (int k = 0; k < m1 / 2; k++)
            {
                result[i][j] = result[i][j] + (matrix1[i][2 * k +
                    1]+ matrix2[2 * k][j]) * (matrix1[i][2 * k] +
                    matrix2[2 * k + 1][j]);
            }
        }
    }

    if (m1 % 2)
    {
        for (int i = 0; i < n1; i++)
        {
            for (int j = 0; j < m2; j++)
            {
                result[i][j] = result[i][j] + matrix1[i][m1 - 1] *
                    matrix2[m1 - 1][j];
            }
        }
    }
    free(mulH);
    free(mulV);
    return result;
}
```

Листинг 3.4: Оптимизированный алгоритм Винограда

```
int** Vinograd_optim(int** matrix1, int n1, int m1, int**
    matrix2, int n2, int m2)
{
    int* mulH = (int*)malloc(n1 * sizeof(int));
    for (int i = 0; i < n1; i++)
        mulH[i] = 0;

    int* mulV = (int*)malloc(m2 * sizeof(int));
    for (int i = 0; i < m2; i++)
```

```
    mulV[i] = 0;

  int** result = new_m(n1, m2);

  int m1Mod2 = m1 % 2;
  int n2Mod2 = n2 % 2;

  for (int i = 0; i < n1; i++)
  {
    for (int j = 0; j < (m1 - m1Mod2); j += 2)
    {
      mulH[i] += matrix1[i][j] * matrix1[i][j + 1];
    }
  }

  for (int i = 0; i < m2; i++)
  {
    for (int j = 0; j < (n2 - n2Mod2); j += 2)
    {
      mulV[i] += matrix2[j][i] * matrix2[j + 1][i];
    }
  }

  for (int i = 0; i < n1; i++)
  {
    for (int j = 0; j < m2; j++)
    {
      int buff = -(mulH[i] + mulV[j]);
      for (int k = 0; k < (m1 - m1Mod2); k += 2)
      {
        buff += (matrix1[i][k + 1] + matrix2[k][j]) * (
            matrix1[i][k] + matrix2[k + 1][j]);
      }
      result[i][j] = buff;
    }
  }

  if (m1Mod2)
  {
    int m1Min_1 = m1 - 1;
```

```
    for (int i = 0; i < n1; i++)
    {
      for (int j = 0; j < m2; j++)
      {
        result[i][j] += matrix1[i][m1Min_1] * matrix2[
          m1Min_1][j];
      }
    }
  }

  free(mulH);
  free(mulV);
  return result;
}
```

3.2.1 Оптимизация алгоритма Винограда

1. Избавиться от деления в цикле;
2. Замена $mulH[i] = mulH[i] + \ldots$ на $mulH[i] + = \ldots$ (аналогично для $mulV[i]$);

Листинг 3.5: Оптимизации алгоритма Винограда №1 и №2

```
int m1Mod2 = m1 % 2;
int n2Mod2 = n2 % 2;

  for (int i = 0; i < n1; i++)
  {
    for (int j = 0; j < (m1 - m1Mod2); j += 2)
    {
      mulH[i] += matrix1[i][j] * matrix1[i][j + 1];
    }
  }

  for (int i = 0; i < m2; i++)
  {
    for (int j = 0; j < (n2 - n2Mod2); j += 2)
    {
      mulV[i] += matrix2[j][i] * matrix2[j + 1][i];
```

```
    }
  }
```

3. Накопление результата в буфер, а вне цикла сброс буфера в ячейку матрицы

Листинг 3.6: Оптимизации алгоритма Винограда №3

```
for (int i = 0; i < n1; i++)
  {
    for (int j = 0; j < m2; j++)
    {
      int buff = -(mulH[i] + mulV[j]);
      for (int k = 0; k < (m1 - m1Mod2); k += 2)
      {
        buff += (matrix1[i][k + 1] + matrix2[k][j]) * (
            matrix1[i][k] + matrix2[k + 1][j]);
      }
      result[i][j] = buff;
    }
  }
```

4 Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов.
Первый эксперимент производится для лучшего случая на матрицах с размерами от 100 x 100 до 1000 x 1000 с шагом 100.

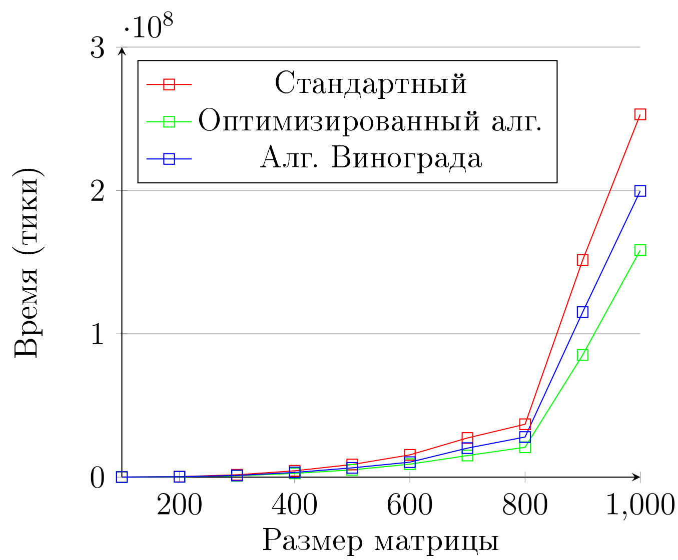

Рисунок 4.1. График времени работы алгоритмов на матрицах четной размерности

Второй эксперимент производится для худшего случая, когда поданы матрицы с нечетными размерами от 101 x 101 до 1001 x 1001 с шагом 100.

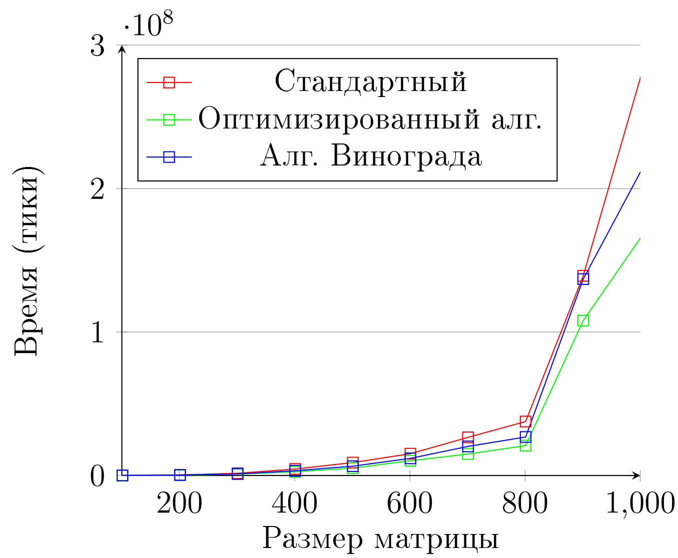

Рисунок 4.2. График времени работы алгоритмов на матрицах нечетной размерности

По результатам тестирования все рассматриваемые алгоритмы реализованы правильно. Самым медленным алгоритмом оказался алгоритм классического умножения матриц, а самым быстрым — оптимизированный алгоритм Винограда

4.2 Тестовые данные

Проверка работы алгоритмов на примерах:

- матрицы размерностью 0 на 0
- матрицы размерностью 1 на 1
- матрицы размерностью 2 на 2
- матрицы размерностью 3 на 3

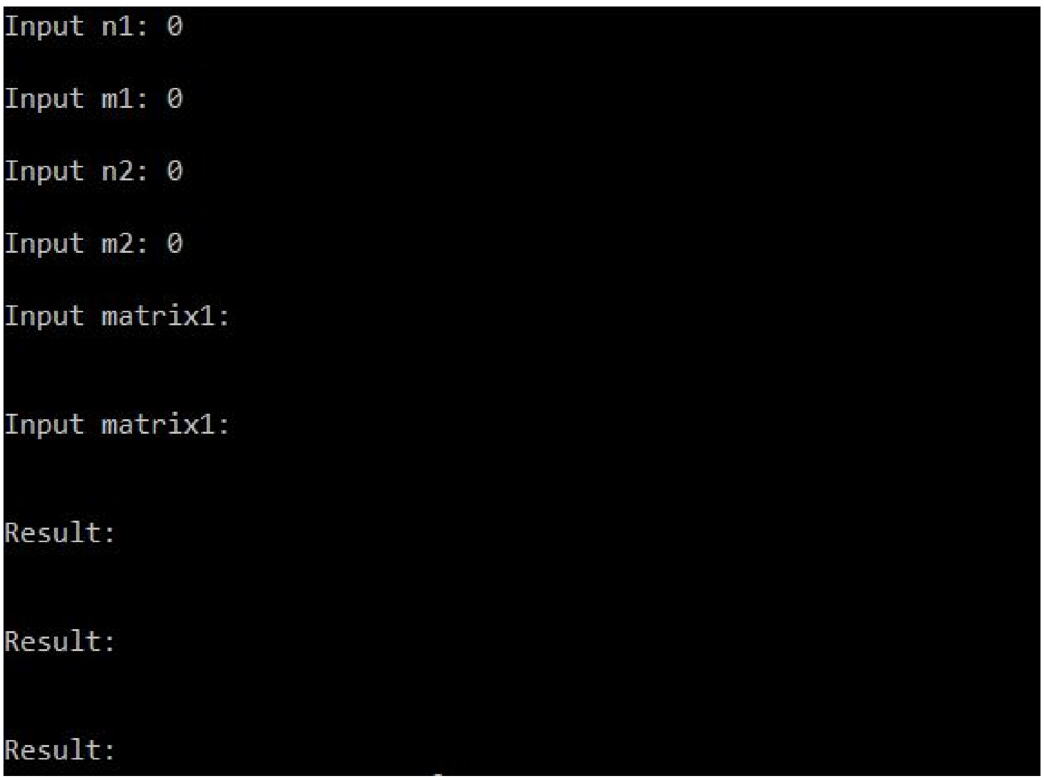

Рис. 4.1: Пример работы алгоритма с матрицами размерностью 0 на 0

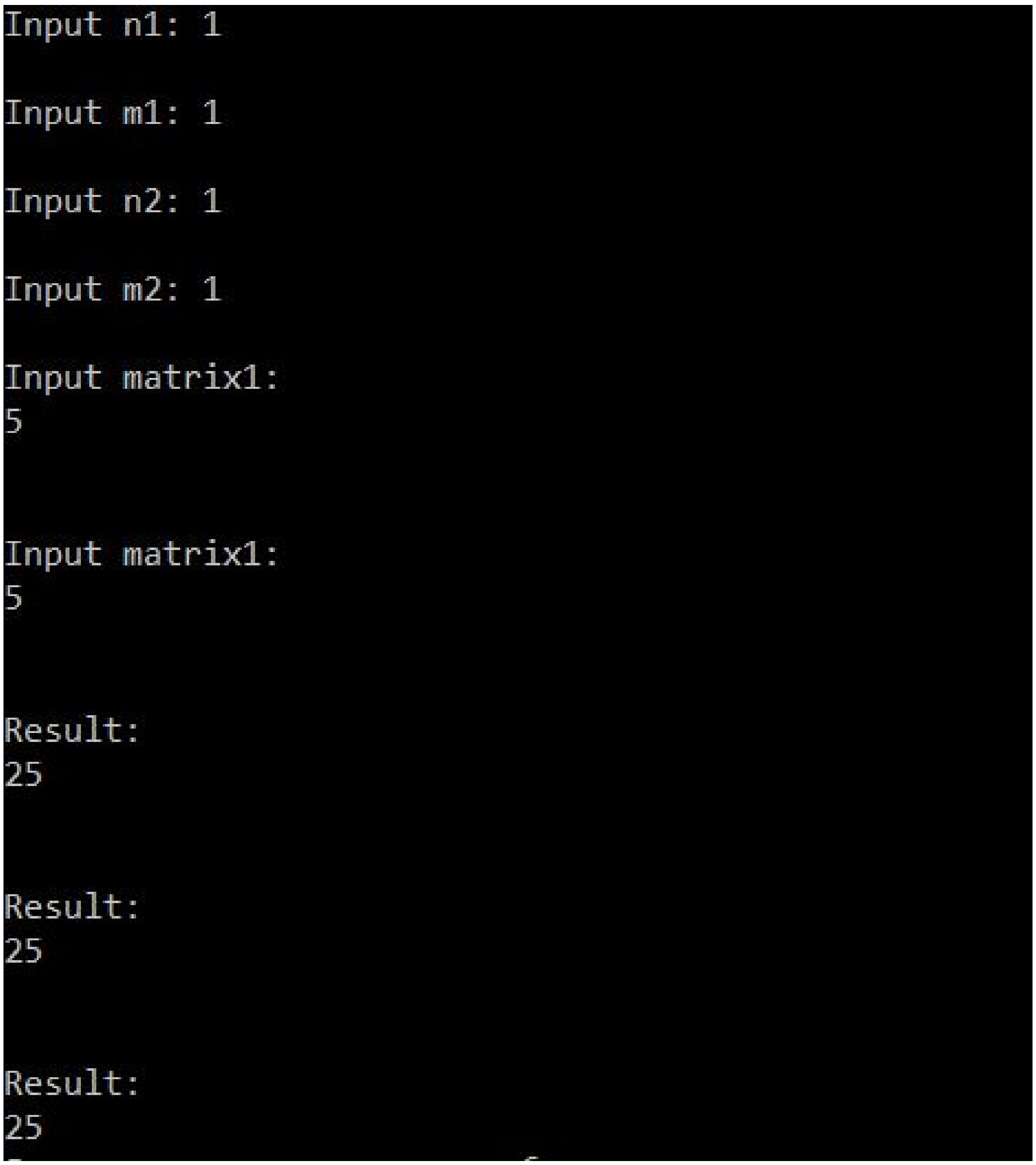

Рис. 4.2: Пример работы алгоритма с матрицами размерностью 1 на 1

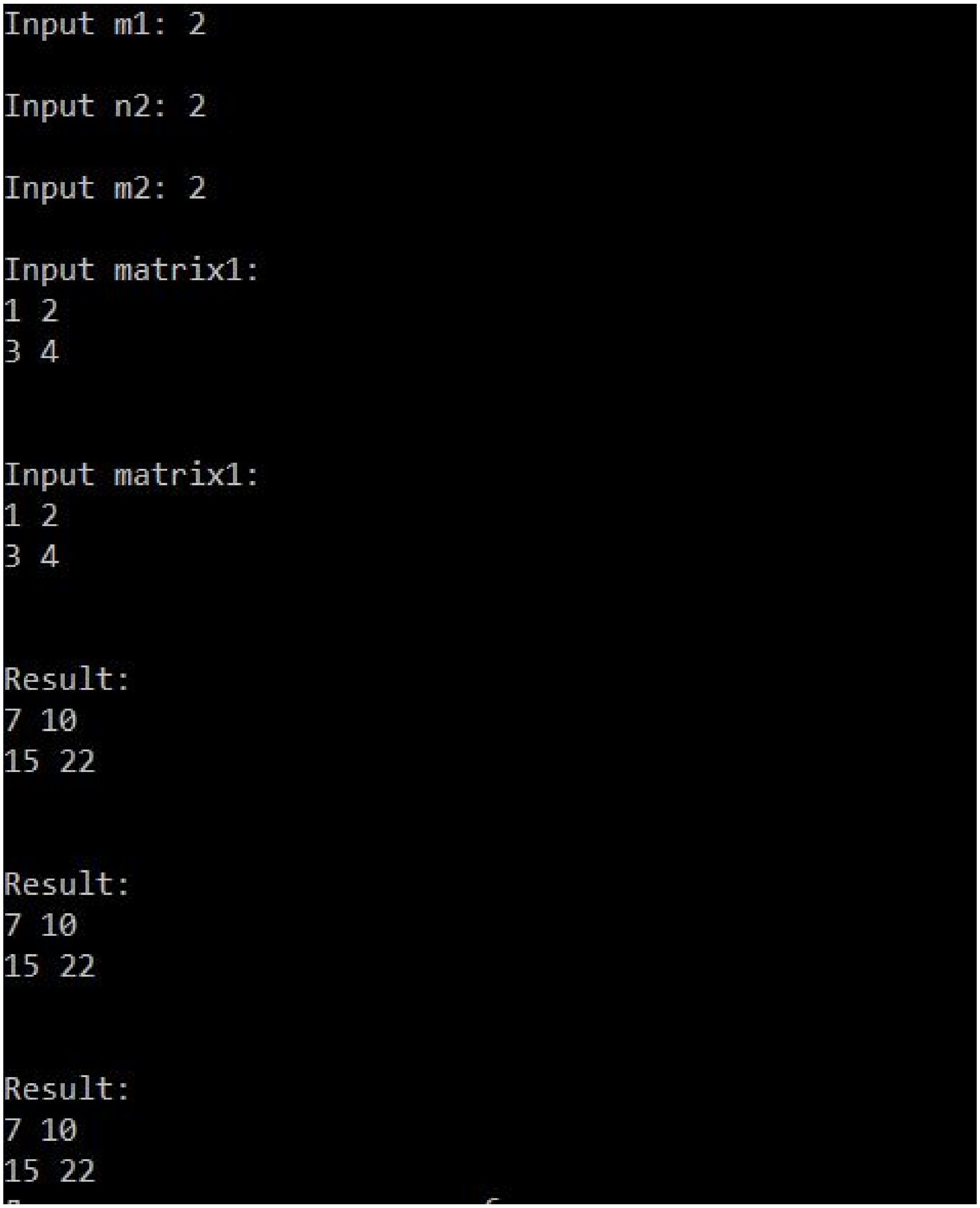

Рис. 4.3: Пример работы алгоритма с матрицами размерностью 2 на 2

```
Input n1: 3

Input m1: 3

Input n2: 3

Input m2: 3

Input matrix1:
1 2 3
4 5 6
7 8 9


Input matrix1:
10 11 12
13 14 15
16 17 18


Result:
84 90 96
201 216 231
318 342 366


Result:
84 90 96
201 216 231
318 342 366


Result:
84 90 96
201 216 231
318 342 366
```

Рис. 4.4: Пример работы алгоритма с матрицами размерностью 3 на 3

Заключение

В ходе лабораторной работы я изучила алгоритмы умножения матриц: стандартный и алгоритм Винограда, оптимизировала алгоритм Винограда, дала теоретическую оценку базового алгоритма умножения матриц, алгоритма Винограда и улучшенного алгоритма Винограда, реализовала три алгоритма умножения матриц на языке программирования и сравнила алгоритмы умножения матриц.