Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по лабораторной работе № 2

по курсу «Анализ Алгоритмов»

на тему: «Умножение матриц (сложность)»

Студент ИУ7-53Б _____ Лысцев Н. Д.
(Группа) (Подпись, дата) (И. О. Фамилия)

Преподаватель _____ Волкова Л. Л.
(Подпись, дата) (И. О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Матрица в математике – таблица чисел, состоящая из определенного количества строк и столбцов.

Умножение матриц – одна из основных операций над матрицами. Оно используется в различных областях, включая машинное обучение, обработку изображений и многие другие.

Целью данной лабораторной работы является исследование классического алгоритма умножения матриц, умножения матриц с использованием алгоритма Винограда и его оптимизированной версии согласно варианту, а также умножения матриц с использованием алгоритма Штрассена.

Для достижения поставленной цели необходимо решить следующие задачи:

1) Изучить и описать алгоритмы классического умножения матриц и умножения матриц с использованием алгоритма Винограда и Штрассена.

2) Создать программное обеспечение, реализующее следующие алгоритмы:

   - классический алгоритм умножения матриц;
   - умножение матриц с использованием алгоритма Винограда;
   - умножение матриц с использованием оптимизированной версии алгоритма Винограда;
   - умножение матриц с использованием алгоритма Штрассена.

3) Провести анализ эффективности реализаций алгоритмов по памяти и по времени.

4) Провести оценку сложности алгоритмов и сказать влияние оптимизаций на характеристики программной реализации.

5) Обосновать полученные результаты в отчете к выполненной лабораторной работе.

1 Аналитический раздел

В данном разделе будут рассмотрены понятия матрицы, умножения двух матриц, классический алгоритм умножения матриц и умножение матриц с помощью алгоритма Винограда и Штрассена.

1.1 Матрица

Матрица – математический объект, записываемый в виде прямоугольной таблицы элементов кольца или поля (например, целых или комплексных чисел), которая представляет собой совокупность строк и столбцов, на пересечении которых находятся её элементы. Количество строк и столбцов матрицы задают размер матрицы [1].

$$A_{m\times n} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

Для матрицы определены следующие алгебраические операции:

1) Сложение матриц, имеющих один и тот же размер.

2) Умножение матрицы на число.

3) Умножение матриц подходящего размера.

Умножение двух матриц (обозначается: AB, реже $A \times B$) определяется следующим образом: каждый элемент результирующей матрицы – это сумма произведений элементов соответствующих строк первой матрицы и столбца второй матрицы. При этом количество столбцов в первой матрице должно совпадать с количеством строк во второй матрице. Операция умножения матриц в общем случае не коммутативна, то есть $AB \neq BA$.

1.2 Классический алгоритм умножения двух матриц

Классический алгоритм умножение двух матриц вытекает из определения умножения двух матриц и реализует формулу 1.1. Асимптотическая сложность такого алгоритма $O(n^3)$ для двух матриц порядка $n \times n$ [2].

Пусть даны две прямоугольные матрицы A и B размерности $m \times n$ и $n \times q$ соответственно:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \cdots & a_{lm} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix}.$$

Тогда матрица C размерностью $m \times q$:

$$C = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1q} \\ c_{21} & c_{22} & \cdots & c_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mq} \end{pmatrix}$$

где элемент результирующей матрицы c_{ij} определяется так:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj} \tag{1.1}$$

1.3 Алгоритм Винограда для умножения двух матриц

Анализируя классический алгоритм умножения двух матриц, можно увидеть, что каждый элемент результирующей матрицы представляет собой скалярное произведение соответствующей строки и соответствующего столбца исходной матрицы.Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее [3].

Рассмотрим 2 вектора: $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$.

Их скалярное произведение равно:

$$V \cdot W = v_1 \cdot w_1 + v_2 \cdot w_2 + v_3 \cdot w_3 + v_3 \cdot w_3 \tag{1.2}$$

Это равенство можно переписать в виде:

$$\begin{gathered} V \cdot W = (v_1 + w_2) \cdot (v_2 + w_1) + (v_3 + w_4) \cdot (v_4 + w_3) \\ -v_1 \cdot v_2 - v_3 \cdot v_4 - w_1 \cdot w_2 - w_3 \cdot w_4 \end{gathered} \tag{1.3}$$

Несмотря на то что второе выражение требует вычисления большего количества операций, чем стандартный алгоритм: вместо четырех умножений - шесть, а вместо трех сложений - десять, последние слагаемые в формуле 1.3 допускают предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй матрицы, что позволит для каждого элемента выполнять лишь два умножения и пять сложений, складывая затем только лишь с 2 предварительно посчитанными суммами соседних элементов текущих строк и столбцов. Из-за того, что операция сложения быстрее операции умножения в ЭВМ, на практике алгоритм должен работать быстрее стандартного.

В случае нечетного значения размера изначальной матрицы следует произвести еще одну операцию - добавление произведения последних элементов соответствующих строк и столбцов.

Алгоритм Винограда имеет асимптотическую сложность $O(n^{2.3755})$ для двух матриц порядка $n \times n$ [4].

1.4 Оптимизированный алгоритм Винограда для умножения двух матриц

При программной реализации алгоритма Винограда предлагается выполнить следующие оптимизации:

1) Заменить умножение на 2 на побитовый сдвиг влево.

2) Заменить выражение вида $x = x + k$ на выражение вида $x += k$.

3) Значение $\frac{Q}{2}$, используемое в циклах расчета предварительных данных, вычислить заранее.

1.5 Алгоритм Штрассена для умножения двух матриц

Алгоритм Штрассена – альтернатива классическому алгоритму умножения матриц. Суть данного алгоритма заключается в сокращении числа умножений (вместо 8-ми умножений в классическом алгоритме 7 в алгоритме Штрассена) путем подсчета дополнительных сумм и разностей. Если классический алгоритм для умножения двух матриц порядка $n \times n$ имеет сложность $O(n^3)$, то метод Штрассена требует $O(n^{2.807})$ [5].

Алгоритм Штрассена работает с квадратными матрицами, размер которых можно представить в виде степени двойки. В случае, если это не так, то матрица дополняется нулевыми элементами до квадратной матрицы ближайшего корректного размера.

Пусть матрицы A и B – квадратные матрицы размером $n \times n$, матрица C – матрица размером $n \times n$, являющаяся результатом умножения матриц A и B, где n – число, являющееся степенью двойки.

Порядок действий в алгоритме Штрассена:

1) Разделить входные матрицы A и B и выходную матрицу C на подматрицы размером $\frac{n}{2} \times \frac{n}{2}$:

$$A = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}, \quad B = \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}, \quad C = \begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix}.$$

где $a_{ij}, b_{ij}, c_{ij} \;\; i, j = \overline{0,1}$ – подматрицы размером $\frac{n}{2} \times \frac{n}{2}$;

2) Вычислить матрицы M_i, $i = \overline{1,7}$ по следующим формулам:

$$M_1 = (a_{00} + a_{11}) \cdot (b_{00} + b_{11}) \tag{1.4}$$

$$M_2 = (a_{10} + a_{11}) \cdot b_{00} \tag{1.5}$$

$$M_3 = (b_{01} - b_{11}) \cdot a_{00} \tag{1.6}$$

$$M_4 = a_{11} \cdot (b_{10} - b_{00}) \tag{1.7}$$

$$M_5 = (a_{00} + a_{01}) \cdot b_{11} \tag{1.8}$$

$$M_6 = (a_{10} - a_{00}) \cdot (b_{00} + b_{01}) \tag{1.9}$$

$$M_7 = (a_{01} - a_{11}) \cdot (b_{10} + b_{11}) \tag{1.10}$$

3) Вычислить подматрицы $c_{00}, c_{01}, c_{10}, c_{11}$ результирующей матрицы C по следующим формулам:

$$c_{00} = M_1 + M_4 - M_5 + M_7 \tag{1.11}$$

$$c_{01} = M_3 + M_5 \tag{1.12}$$

$$c_{10} = M_2 + M_4 \tag{1.13}$$

$$c_{11} = M_1 + M_3 - M_2 + M_6 \tag{1.14}$$

В программной реализации данного алгоритма используется рекурсия. Метод позволяет рекурсивно делить матрицы на подматрицы до тех пор, пока порядок не сранет равным 2. Далее происходит классическое умножение [6].

Пусть $M(n)$ – количество умножений, выполняемых алгоритмом Штрассена для умножения двух матриц размерами $n \times n$. Тогда для алгоритма Штрассена справедливо следующее рекуррентное соотношение [5]:

$$M(n) = 7 \cdot M(\frac{n}{2}) \text{ при } n \geq 1, M(1) = 1 \tag{1.15}$$

1.6 Оптимизированный алгоритм Штрассена для умножения двух матриц

При программной реализации алгоритма Штрассена предлагается выполнить следующие оптимизации:

1) Заменить умножение на 2 на побитовый сдвиг влево.

2) Заменить выражение вида $x = x + k$ на выражение вида $x += k$.

Вывод

В данном разделе были рассмотрены понятия матрицы и операции умножения, классического алгоритма умножения матриц и алгоритма умножения матриц с помощью алгоритма Винограда и Штрассена, а также были приведены варинты оптимизаций алгоритма Винограда и алгоритма Штрассена.

2 Конструкторский раздел

В данном разделе будут разработаны алгоритмы классического умножения матриц, умножения матриц с использованием алгоритма Винограда и его оптимизированной версии, а так же алгоритма Штрассена для умножения матриц и его оптимизированной версии и приведены схемы алгоритмов их реализации. Также будет приведена оценка трудоемкости данных алгоритмов.

2.1 Разработка алгоритма классического умножения матриц

На рисунке 2.1 приведена схема алгоритма классического умножения двух матриц.

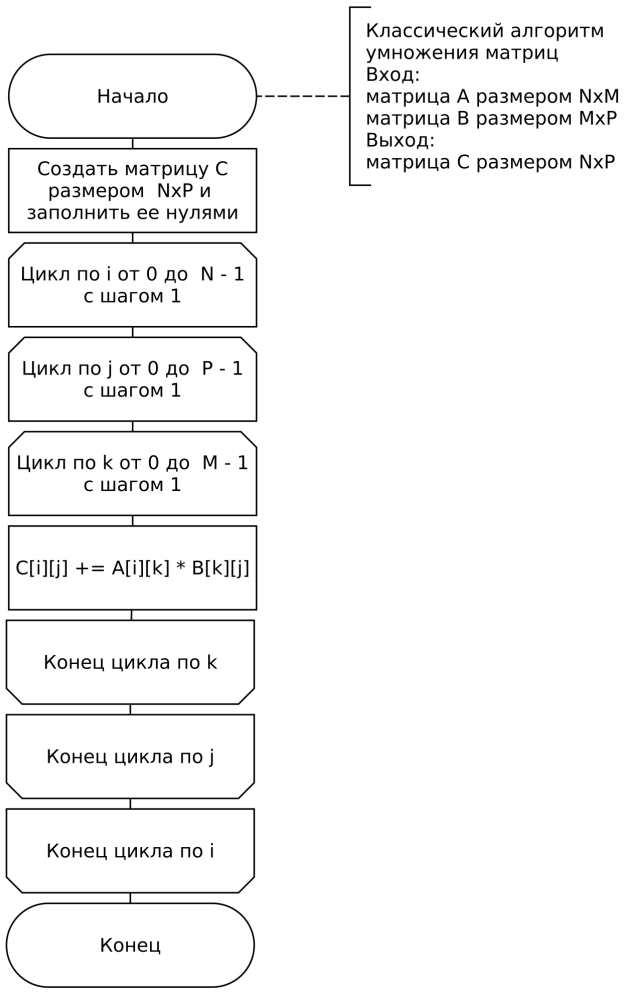

Рисунок 2.1 – Схема алгоритма классического умножения двух матриц

2.2 Разработка алгоритма Винограда для умножения двух матриц

На рисунке 2.2 приведена схема алгоритма Винограда для умножения двух матриц.

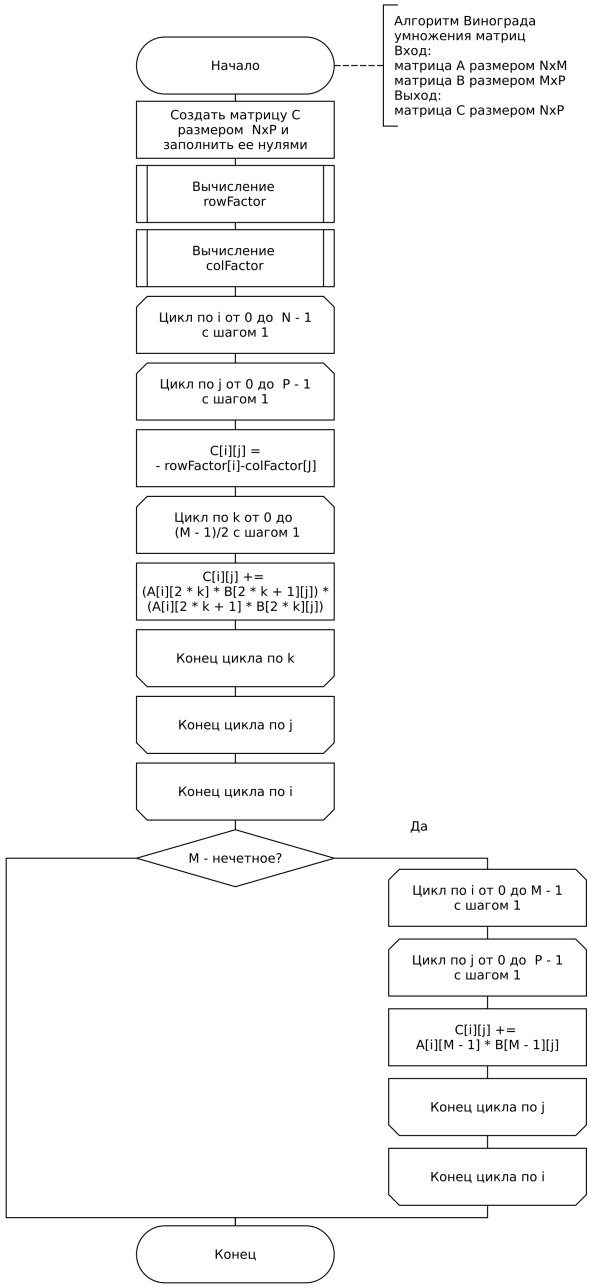

Рисунок 2.2 – Схема алгоритма Винограда для умножения двух матриц

На рисунке 2.3 приведена схема подпрограммы вычисления сумм произведений пар соседних элементов строк матрицы.

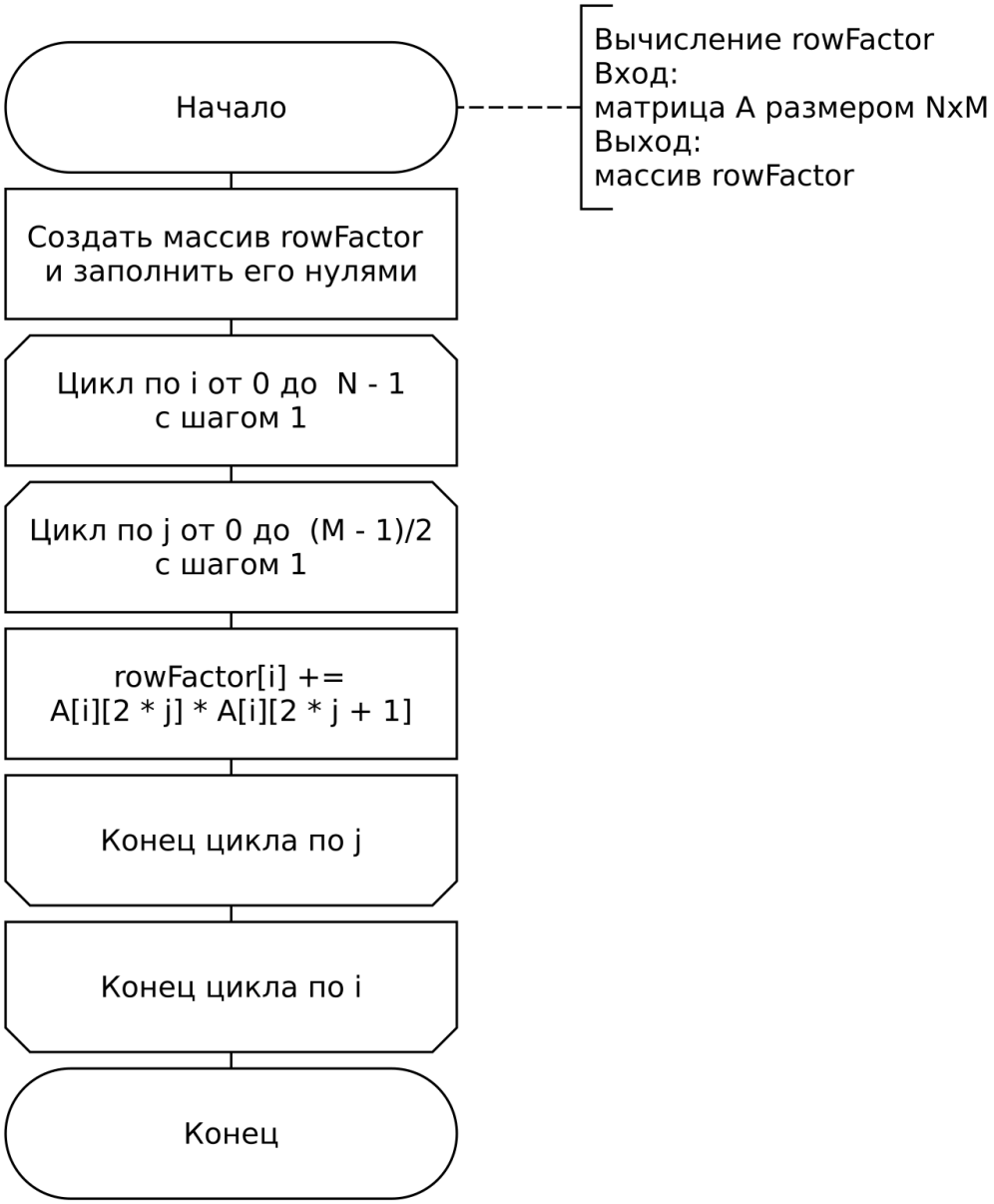

Рисунок 2.3 – Схема подпрограммы вычисления сумм произведений пар соседних элементов строк матрицы

На рисунке 2.4 приведена схема подпрограммы вычисления сумм произведений пар соседних элементов строк матрицы.

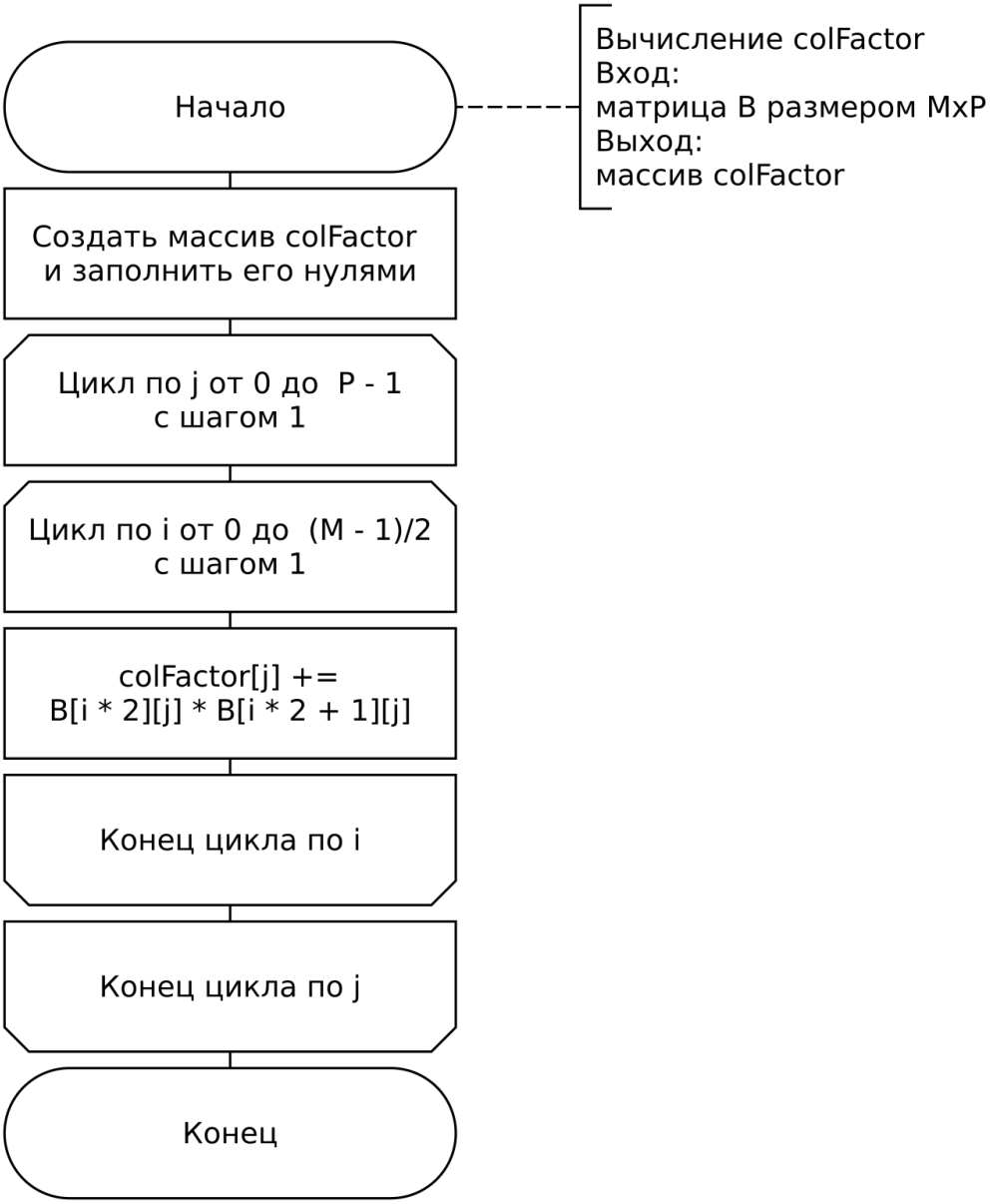

Рисунок 2.4 – Схема подпрограммы вычисления сумм произведений пар соседних элементов столбцов матрицы

2.3 Разработка алгоритма Штрассена для умножения двух матриц

На рисунке 2.5 приведена схема алгоритма Штрассена для умножения двух матриц.

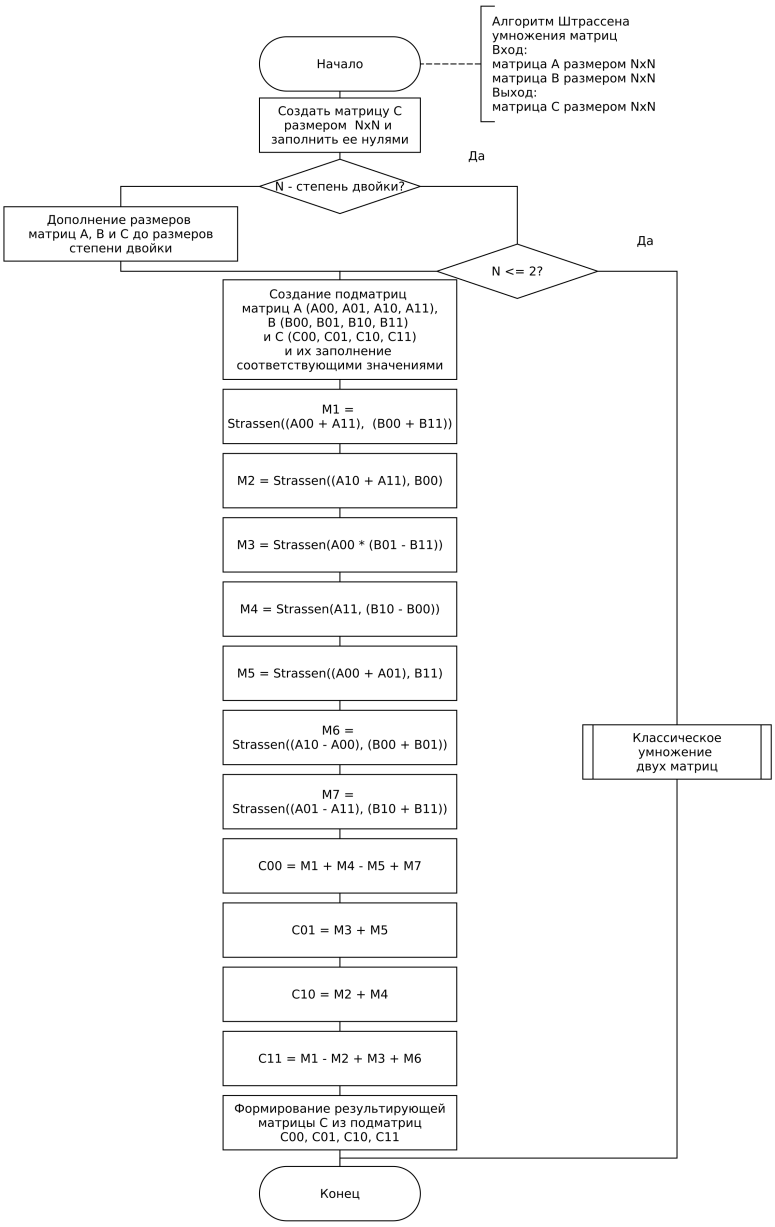

Рисунок 2.5 – Схема алгоритма Штрассена умножения двух матриц

2.4 Оценка трудоемкости алгоритмов

2.4.1 Модель вычислений для проведения оценки трудоемкости алгоритмов

Была введена модель вычислений для определения трудоемкости каждого отдельного взятого алгоритма сортировки.

1) Трудоемкость базовых операций имеет:

   — равную 1:

$$+, -, =, + =, - =, ==, ! =, <, >, <=, >=, [], ++, --, \\ \&\&, >>, <<, ||, \&, | \tag{2.1}$$

   — равную 2:

$$*, /, \%, * =, / =, \% = \tag{2.2}$$

2) Трудоемкость условного оператора:

$$f_{if} = f_{\text{условия}} + \begin{cases} min(f_1, f_2), & \text{лучший случай} \\ max(f_1, f_2), & \text{худший случай} \end{cases} \tag{2.3}$$

3) Трудоемкость цикла:

$$f_{for} = f_{\text{инициализация}} + f_{\text{сравнения}} + M_{\text{итераций}} \cdot (f_{\text{тело}} + \\ + f_{\text{инкремент}} + f_{\text{сравнения}}) \tag{2.4}$$

4) Трудоемкость передачи параметра в функции и возврат из функции равны 0.

2.4.2 Трудоемкость классического алгоритма умножения двух матриц

Для стандартного алгоритма умножения матриц трудоемкость будет слагаться из:

— внешнего цикла по $i \in [1 \dots N]$, трудоёмкость которого: $f = 2 + N \cdot (2 + f_{body})$;

— цикла по $j \in [1 \dots P]$, трудоёмкость которого: $f = 2 + 2 + P \cdot (2 + f_{body})$;

— цикла по $k \in [1 \dots M]$, трудоёмкость которого: $f = 2 + 2 + 14M$;

Поскольку трудоемкость стандартного алгоритма равна трудоемкости внешнего цикла, то:

$$\begin{aligned} f_{standart} &= 2 + N \cdot (2 + 2 + P \cdot (2 + 2 + M \cdot (2 + 8 + 1 + 1 + 2))) = \\ &= 2 + 4N + 4NP + 14NMP \approx 14NMP = O(N^3) \end{aligned} \tag{2.5}$$

2.4.3 Трудоемкость алгоритма Винограда для умножения двух матриц

При вычислении трудоемкости алгоритма Винограда учитывается следующее:

— создание и инициализация массивов $rowFactor$ и $colFactor$, трудоёмкость которых указана в формуле (2.6);

$$f_{init} = N + M \tag{2.6}$$

— заполнение массива $rowFactor$, трудоёмкость которого указана в формуле (2.7);

$$\begin{aligned} f_{rowFactor} &= 2 + N \cdot (4 + \frac{M}{2} \cdot (4 + 6 + 1 + 2 + 3 \cdot 2)) = \\ &= 2 + 4N + \frac{19NM}{2} = 2 + 4N + 9,5NM \end{aligned} \tag{2.7}$$

— заполнение массива $colFactor$, трудоёмкость которого указана в формуле (2.8);

$$f_{colFactor} = 2 + P \cdot (4 + \frac{M}{2} \cdot (4 + 6 + 1 + 2 + 3 \cdot 2)) = \\ = 2 + 4P + \frac{19PM}{2} = 2 + 4P + 9,5PM \tag{2.8}$$

— цикл заполнения для чётных размеров, трудоёмкость которого указана в формуле (2.9);

$$f_{cycle} = 2 + N \cdot (4 + P \cdot (2 + 7 + 4 + \frac{M}{2} \cdot (4 + 28))) = \\ = 2 + 4N + 13NP + \frac{32NPM}{2} = 2 + 4N + 13NP + 16NPM \tag{2.9}$$

— цикла, который дополнительно нужен для подсчёта значений при нечётном размере матрицы, трудоемкость которого указана в формуле (2.10);

$$f_{check} = 3 + \begin{cases} 0, & \text{чётная} \\ 2 + M \cdot (4 + P \cdot (2 + 14)), & \text{иначе} \end{cases} \tag{2.10}$$

Тогда для худшего случая (нечётный общий размер матриц) имеем:

$$f_{worst} = f_{init} + f_{rowFactor} + f_{colFactor} + f_{cycle} + f_{check} \approx 16NMP = O(N^3) \tag{2.11}$$

Для лучшего случая (чётный общий размер матриц) имеем:

$$f_{best} = f_{init} + f_{rowFactor} + f_{colFactor} + f_{cycle} + f_{check} \approx 16NMP = O(N^3) \tag{2.12}$$

2.4.4 Трудоемкость оптимизированного алгоритма Винограда для умножения двух матриц

Трудоемкость оптимизированного алгоритма Винограда состоит из:

— кэширования значения $\frac{M}{2}$ в циклах, которое равно 3;

— создания и инициализации массивов $rowFactor$ и $colFactor$ (2.6);

— заполнения массива $rowFactor$, трудоёмкость которого (2.7);

— заполнения массива $colFactor$, трудоёмкость которого (2.8);

— цикла заполнения для чётных размеров, трудоёмкость которого указана в формуле (2.13);

$$\begin{aligned} f_{cycle} = 2 + N \cdot (4 + P \cdot (4 + 7 + \frac{M}{2} \cdot (2 + 10 + 5 + 2 + 4))) = \\ = 2 + 4N + 11NP + \frac{23NPM}{2} = 2 + 4N + 11NP + 11,5 \cdot NPM \end{aligned} \tag{2.13}$$

— условия, которое нужно для дополнительных вычислений при нечётном размере матрицы, трудоемкость которого указана в формуле (2.14);

$$f_{check} = 3 + \begin{cases} 0, & \text{чётная} \\ 2 + N \cdot (4 + P \cdot (2 + 10)), & \text{иначе} \end{cases} \tag{2.14}$$

Тогда для худшего случая (нечётный общий размер матриц) имеем:

$$f_{worst} = 3 + f_{init} + f_{atmp} + f_{btmp} + f_{cycle} + f_{check} \approx 11NMP = O(N^3) \tag{2.15}$$

Для лучшего случая (чётный общий размер матриц) имеем:

$$\begin{aligned} f_{best} = 3 + f_{init} + f_{rowFactor} + f_{colFactor} \\ + f_{cycle} + f_{check} \approx 11NMP = O(N^3) \end{aligned} \tag{2.16}$$

2.4.5 Трудоемкость алгоритма Штрассена для умножения двух матриц

Пусть

— REC – трудоемкость рекурсивного алгоритма;

— DIR – трудоемкость прямого решения;

— DIV – трудоемкость разбиения ввода (N) на несколько частей;

— COM – трудоемкость объединения решений.

Тогда трудоемкость рекурсивного алгоритма считается по следующей формуле:

$$REC(N) = \begin{cases} DIR(N), & N \leq N_0 \\ DIV(N) + \sum_{i=1}^{n} REC(F[i]) + COM(N), & N > N_0 \end{cases} \quad (2.17)$$

где N – число входных элементов, N_0 – наибольшее число, определяющее тривиальный случай (прямое решение), n – число рекурсивных вызовов для данного N, $F[i]$ – число входных элементов для данного i.

Для расчета трудоемкости алгоритма Штрассена предположим, что размеры переданных матриц – степени двойки.

Тогда трудоемкость алгоритма Штрассена определяется следующим образом:

— Для матрицы, размером $N \leq 2$ трудоемкость определяется как и в случае классического алгоритма умножения матриц, то есть согласно формуле 2.5

— Для матриц размером $N > 2$ определяется так:

1) Трудоемкость разбиения ввода (N) на части. Каждый следующий вызов берется размерность матрицы в 2 раза меньше предыдущей,

и происходит создание соответствующих подматриц и заполнение их значениями.

$$DIV(N) = 1 + 8 \cdot (3 + \frac{N}{2} \cdot ((3 + \frac{N}{2} \cdot (5 + 2 + 1)) + 2 + 1) = \\ 16 \cdot N^2 + 24 \cdot N + 25 \tag{2.18}$$

2) Трудоемкость вычисления матриц $M_i,\ \ i = \overline{1,7}$ (обозначим ее буквой $G = G(N)$):

$$G(N) = 10 \cdot (2 + \frac{N}{2} \cdot (2 + \frac{N}{2} \cdot (8 + 1 + 1) + 1 + 1)) + \\ +7 \cdot REC(\frac{N}{2}) \tag{2.19}$$

где, так как $N = 2^k$ и согласно с 1.15

$$REC(\frac{N}{2}) = REC(2^{k-1}) = 7 \cdot M(2^{k-2}) = \ldots 7^{i-1} M(2^{k-i}) = \ldots \\ 7^{k-1} M(2^{k-k}) = 7^{k-1} \tag{2.20}$$

подставляя $k = \log_2(N)$ получаем, что

$$REC(\frac{N}{2}) = \frac{N^{\log_2(7)}}{7} \tag{2.21}$$

Таким образом, трудоемкость вычисления матриц $M_i,\ \ i = \overline{1,7}$ определяется следующей формулой:

$$G(N) = 10 \cdot (10 \cdot (\frac{N}{2})^2 + 4 \cdot \frac{N}{2} + 2) + N^{\log_2(7)} = \\ 25 \cdot N^2 + 20 \cdot N + 20 + N^{\log_2(7)} \tag{2.22}$$

3) Трудоемкость объединения решений, а именно формирование результирующей матрицы из вычисленных матриц $M_i,\ \ i = \overline{1,7}$

$$
\begin{gathered}
COM(N) = 8 \cdot (2 + \frac{N}{2} \cdot (2 + \frac{N}{2} \cdot (8 + 1 + 1) + 1 + 1)) + \\
4 \cdot (3 + \frac{N}{2} \cdot ((3 + \frac{N}{2} \cdot (5 + 2 + 1)) + 2 + 1) = \\
28 \cdot N^2 + 28 \cdot N + 28
\end{gathered}
\tag{2.23}
$$

Таким образом, для матриц размером $N > 2$ трудоемкость алгоритма Штрассена согласно 2.17 определяется так:

$$
\begin{gathered}
f_{strassen}(N) = DIV(N) + G(N) + COM(N) = \\
16 \cdot N^2 + 24 \cdot N + 25 + 25 \cdot N^2 + 20 \cdot N + 20 + N^{\log_2(7)} + \\
28 \cdot N^2 + 28 \cdot N + 28 = \\
N^{\log_2(7)} + 69 \cdot N^2 + 72 \cdot N + 73 \approx N^{\log_2(7)} = O(N^{\log_2(7)})
\end{gathered}
\tag{2.24}
$$

2.4.6 Трудоемкость оптимизированного алгоритма Штрассена для умножения двух матриц

При программной реализации алгоритма Штрассена не нашлось мест для применения предложенных по варианту оптимизаций, поэтому трудоемкость алгоритма Штрассена осталась такой же, как и в предыдущем пункте.

Вывод

В данном разделе были построены схемы алгоритмов классического умножения матриц, умножения матриц с использованием алгоритма Винограда и алгоритма Штрассена. Также были приведены оценки трудоемкости этих алгоритмов.

Согласно расчетам трудоемкости, наиболее эффективным оказался алгоритм Штрассена. Трудоемкость оптимизированной версии алгоритма Винограда в 1.5 раза меньше, чем у его неоптимизированной версии и в 1.27 раз маньше, чем у классического алгоритма.

3 Технологический раздел

В данном разделе будут перечислены средства реализации, листинги кода и функциональные тесты.

3.1 Средства реализации

В качестве языка программирования для этой лабораторной работы был выбран $C++$ [7] по следующим причинам:

- в $C++$ есть встроенный модуль *ctime*, предоставляющий необходимый функционал для замеров процессорного времени;
- в стандартной библиотеке $C++$ есть оператор $sizeof$, позволяющий получить размер переданного объекта в байтах. Следовательно, $C++$ предоставляет возможности для проведения точных оценок по используемой памяти.

В качестве функции, которая будет осуществлять замеры процессорного времени, будет использована функция *clock_gettime* из встроенного модуля *ctime* [8].

3.2 Сведения о модулях программы

Программа состоит из шести модулей:

1) `algorithms.cpp` — модуль, хранящий реализации алгоритмов умножения матриц;
2) `processTime.cpp` — модуль, содержащий функцию для замера процессорного времени;
3) `memoryMeasurements.cpp` — модуль, содержащий функции, позволяющие провести сравнительный анализ использования памяти в реализациях алгоритмов умножения матриц;

4) `timeMeasurements.cpp` — модуль, содержащий функции, позволяющие провести сравнительный анализ использования времени в реализациях алгоритмов умножения матриц;

5) `main.cpp` — файл, содержащий точку входа в программу;

6) `task7` — модуль, содержащий набор скриптов для проведения замеров программы по времени и памяти и построения графиков по полученным данным.

3.3 Реализации алгоритмов

В листингах 3.1 - 3.7 приведены реализации алгоритмов умножения матриц.

Листинг 3.1 – Реализация классического алгоритма умножения двух матриц

```
vector<vector<int>> matrixMulClassic(
    vector<vector<int>> &mtr1,
    vector<vector<int>> &mtr2,
    int cRow, int cCol, int cColRes)
{

    vector<vector<int>> mtrRes(cRow, vector<int>(cColRes, 0));

    for (int i = 0; i < cRow; ++i)
        for (int j = 0; j < cColRes; ++j)
            for (int k = 0; k < cCol; ++k)
                mtrRes[i][j] = mtrRes[i][j] + mtr1[i][k] *
                   mtr2[k][j];

    return mtrRes;
}
```

Листинг 3.2 – Реализация алгоритма Винограда для умножения двух матриц

```
vector<vector<int>> matrixMulVinograd(
    vector<vector<int>> &mtr1,
    vector<vector<int>> &mtr2,
    int cRow, int cCol, int cColRes)
{
    vector<vector<int>> mtrRes(cRow, vector<int>(cColRes, 0));
    vector<int> rowFactor(cRow, 0), colFactor(cRow, 0);

    for (int i = 0; i < cRow; ++i)
        for (int j = 0; j < cCol / 2; ++j)
            rowFactor[i] = rowFactor[i] + mtr1[i][2 * j] *
               mtr1[i][2 * j + 1];

    for (int j = 0; j < cColRes; ++j)
        for (int i = 0; i < cCol / 2; ++i)
            colFactor[j] = colFactor[j] + mtr2[i * 2][j] * mtr2[i
               * 2 + 1][j];

    for (int i = 0; i < cRow; ++i)
        for (int j = 0; j < cColRes; ++j)
        {
            mtrRes[i][j] = -rowFactor[i] - colFactor[j];

            for (int k = 0; k < cCol / 2; ++k)
                mtrRes[i][j] = mtrRes[i][j] + (mtr1[i][2 * k] +
                   mtr2[2 * k + 1][j]) * (mtr1[i][2 * k + 1] +
                   mtr2[2 * k][j]);
        }

    if (cCol % 2 != 0)
        for (int i = 0; i < cCol; ++i)
            for (int j = 0; j < cColRes; ++j)
                mtrRes[i][j] = mtrRes[i][j] + mtr1[i][cCol - 1] *
                   mtr2[cCol - 1][j];

    return mtrRes;
}
```

Листинг 3.3 – Реализация оптимизированного алгоритма Винограда для умножения двух матриц

```
vector<vector<int>> matrixMulVinogradWithOpt(
    vector<vector<int>> &mtr1,
    vector<vector<int>> &mtr2,
    int cRow, int cCol, int cColRes)
{
    vector<vector<int>> mtrRes(cRow, vector<int>(cColRes, 0));
    vector<int> rowFactor(cRow, 0), colFactor(cRow, 0);
    int cCol_half = cCol / 2;

    for (int i = 0; i < cRow; ++i)
        for (int j = 0; j < cCol_half; ++j)
            rowFactor[i] += mtr1[i][j << 1] * mtr1[i][(j << 1) +
               1];
    for (int j = 0; j < cColRes; ++j)
        for (int i = 0; i < cCol_half; ++i)
            colFactor[j] += mtr2[i << 1][j] * mtr2[(i << 1) +
               1][j];

    for (int i = 0; i < cRow; ++i)
        for (int j = 0; j < cColRes; ++j)
        {
            mtrRes[i][j] = -rowFactor[i] - colFactor[j];

            for (int k = 0; k < cCol_half; ++k)
                mtrRes[i][j] += (mtr1[i][k << 1] + mtr2[(k << 1) +
                   1][j]) * (mtr1[i][(k << 1) + 1] + mtr2[k <<
                   1][j]);
        }
    if (cCol % 2 != 0)
        for (int i = 0; i < cCol; ++i)
            for (int j = 0; j < cColRes; ++j)
                mtrRes[i][j] += mtr1[i][cCol - 1] * mtr2[cCol -
                   1][j];

    return mtrRes;
}
```

Листинг 3.4 – Реализация функций, необходимых для работы алгоритма Штрассена

```
void split(vector<vector<int>> &A, vector<vector<int>> &B,
           int row, int col, int d)
{
    for (int i1 = 0, i2 = row; i1 < d; i1++, i2++)
        for (int j1 = 0, j2 = col; j1 < d; j1++, j2++)
            B[i1][j1] = A[i2][j2];
}

void join(vector<vector<int>> &A, vector<vector<int>> &B,
          int row, int col, int d)
{
    for (int i1 = 0, i2 = row; i1 < d; i1++, i2++)
        for (int j1 = 0, j2 = col; j1 < d; j1++, j2++)
            B[i2][j2] = A[i1][j1];
}

void add(vector<vector<int>> &A,
         vector<vector<int>> &B,
         vector<vector<int>> &C, int d)
{
    for (int i = 0; i < d; i++)
        for (int j = 0; j < d; j++)
            C[i][j] = A[i][j] + B[i][j];
}

void sub(vector<vector<int>> &A,
         vector<vector<int>> &B,
         vector<vector<int>> &C, int d)
{
    for (int i = 0; i < d; i++)
        for (int j = 0; j < d; j++)
            C[i][j] = A[i][j] - B[i][j];
}
```

Функции *split* выполняет заполнение переданной подматрицы B необходимыми значениями из основной матрицы A, а функция *join* выполняет заполнение

переданной результирующей матрицы B необходимыми значениями из подматрицы A. Функции *add* и *sub* выполняют сложение и вычитание матриц A и B, и результат записывается в матрицу C.

Листинг 3.5 – Реализация алгоритма Штрассена (начало)

```
void matrixMulStrassen(vector<vector<int>> &A,
                       vector<vector<int>> &B,
                       vector<vector<int>> &C, int d)
{
    if (!(d && !(d & (d - 1)))) // если d не степень двойки
    {
        d = pow(2, ceil(log2(d)));
        A.resize(d), B.resize(d), C.resize(d);

        for (int i = 0; i < d; ++i)
            A[i].resize(d), B[i].resize(d), C[i].resize(d);
    }

    if (d <= 86)
    {
        C = classicMul::matrixMulClassic(A, B, d, d, d);
        return;
    }

    int new_d = d / 2;
    vector<int> inside(new_d);

    vector<vector<int>> A11(new_d, inside);
    vector<vector<int>> A12(new_d, inside);
    vector<vector<int>> A21(new_d, inside);
    vector<vector<int>> A22(new_d, inside);
    vector<vector<int>> B11(new_d, inside);
    vector<vector<int>> B12(new_d, inside);
    vector<vector<int>> B21(new_d, inside);
    vector<vector<int>> B22(new_d, inside);
    vector<vector<int>> C11(new_d, inside);
    vector<vector<int>> C12(new_d, inside);
    vector<vector<int>> C21(new_d, inside);
    vector<vector<int>> C22(new_d, inside);
```

Листинг 3.6 – Реализация алгоритма Штрассена (продолжение)

```
    split(A, A11, 0, 0, new_d);
    split(A, A12, 0, new_d, new_d);
    split(A, A21, new_d, 0, new_d);
    split(A, A22, new_d, new_d, new_d);
    split(B, B11, 0, 0, new_d);
    split(B, B12, 0, new_d, new_d);
    split(B, B21, new_d, 0, new_d);
    split(B, B22, new_d, new_d, new_d);

    vector<vector<int>> result1(new_d, inside);
    vector<vector<int>> result2(new_d, inside);

    add(A11, A22, result1, new_d);
    add(B11, B22, result2, new_d);
    vector<vector<int>> M1(new_d, inside);
    matrixMulStrassen(result1, result2, M1, new_d);

    add(A21, A22, result1, new_d);
    vector<vector<int>> M2(new_d, inside);
    matrixMulStrassen(result1, B11, M2, new_d);

    sub(B12, B22, result2, new_d);
    vector<vector<int>> M3(new_d, inside);
    matrixMulStrassen(A11, result2, M3, new_d);

    sub(B21, B11, result2, new_d);
    vector<vector<int>> M4(new_d, inside);
    matrixMulStrassen(A22, result2, M4, new_d);

    add(A11, A12, result1, new_d);
    vector<vector<int>> M5(new_d, inside);
    matrixMulStrassen(result1, B22, M5, new_d);

    sub(A21, A11, result1, new_d);
    add(B11, B12, result2, new_d);
    vector<vector<int>> M6(new_d, inside);
    matrixMulStrassen(result1, result2, M6, new_d);
```

Листинг 3.7 – Реализация алгоритма Штрассена (конец)

```
    sub(A12, A22, result1, new_d);
    add(B21, B22, result2, new_d);
    vector<vector<int>> M7(new_d, inside);
    matrixMulStrassen(result1, result2, M7, new_d);

    add(M1, M4, result1, new_d);
    add(result1, M7, result2, new_d);
    sub(result2, M5, C11, new_d);

    add(M3, M5, C12, new_d);

    add(M2, M4, C21, new_d);

    sub(M1, M2, result1, new_d);
    add(M3, M6, result2, new_d);
    add(result1, result2, C22, new_d);

    join(C11, C, 0, 0, new_d);
    join(C12, C, 0, new_d, new_d);
    join(C21, C, new_d, 0, new_d);
    join(C22, C, new_d, new_d, new_d);
}
```

3.4 Функциональные тесты

В таблице 3.1, 3.2 и 3.3 приведены функциональные тесты для разработанных алгоритмов умножения матриц. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты для классического алгоритма умножения матриц

Входные данные		**Результат для классического алгоритма**	
Матрица 1	**Матрица 2**	**Ожидаемый результат**	**Фактический результат**
$\begin{pmatrix} 1 & 5 & 7 \\ 2 & 6 & 8 \\ 3 & 7 & 9 \end{pmatrix}$	$\begin{pmatrix} \end{pmatrix}$	Сообщение об ошибке	Сообщение об ошибке
$\begin{pmatrix} 1 & 5 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$	Сообщение об ошибке	Сообщение об ошибке
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$
$\begin{pmatrix} 3 & 5 \\ 2 & 1 \\ 9 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	$\begin{pmatrix} 23 & 31 & 39 \\ 6 & 9 & 12 \end{pmatrix}$	$\begin{pmatrix} 23 & 31 & 39 \\ 6 & 9 & 12 \end{pmatrix}$
$\begin{pmatrix} 10 \end{pmatrix}$	$\begin{pmatrix} 35 \end{pmatrix}$	$\begin{pmatrix} 350 \end{pmatrix}$	$\begin{pmatrix} 350 \end{pmatrix}$

Таблица 3.2 – Функциональные тесты для умножения матриц по алгоритму Винограда

Входные данные		**Результат для алгоритма Винограда**	
Матрица 1	**Матрица 2**	**Ожидаемый результат**	**Фактический результат**
$\begin{pmatrix} 1 & 5 & 7 \\ 2 & 6 & 8 \\ 3 & 7 & 9 \end{pmatrix}$	$\begin{pmatrix} \end{pmatrix}$	Сообщение об ошибке	Сообщение об ошибке
$\begin{pmatrix} 1 & 5 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$	Сообщение об ошибке	Сообщение об ошибке
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$
$\begin{pmatrix} 3 & 5 \\ 2 & 1 \\ 9 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	$\begin{pmatrix} 23 & 31 & 39 \\ 6 & 9 & 12 \end{pmatrix}$	$\begin{pmatrix} 23 & 31 & 39 \\ 6 & 9 & 12 \end{pmatrix}$
$\begin{pmatrix} 10 \end{pmatrix}$	$\begin{pmatrix} 35 \end{pmatrix}$	$\begin{pmatrix} 350 \end{pmatrix}$	$\begin{pmatrix} 350 \end{pmatrix}$

Таблица 3.3 – Функциональные тесты для умножения матриц по алгоритму Штрассена

Входные данные		**Результат для алгоритма Штрассена**	
Матрица 1	**Матрица 2**	**Ожидаемый результат**	**Фактический результат**
$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$	$\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$	$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$	$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$
$\begin{pmatrix} 1 & 5 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$	Сообщение об ошибке	Сообщение об ошибке
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$
$\begin{pmatrix} 3 & 5 \\ 2 & 1 \\ 9 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	Сообщение об ошибке	Сообщение об ошибке
$\begin{pmatrix} 10 \end{pmatrix}$	$\begin{pmatrix} 35 \end{pmatrix}$	$\begin{pmatrix} 350 \end{pmatrix}$	$\begin{pmatrix} 350 \end{pmatrix}$

Вывод

В данном разделе были реализованы и протестированы 4 алгоритма: классический алгоритм умножения, алгоритм Винограда для умножения двух матриц, оптимизированный алгоритм Винограда для умножения двух матриц и алгоритм Штрассена для умножения двух матриц.

4 Исследовательский раздел

В данном разделе будут проведены сравнения реализаций алгоритмов умножения матриц по времени работы и по затрачиваемой памяти.

4.1 Технические характеристики

Технические характеристики устройства, на котором проводились исследования:

– операционная система: Ubuntu 22.04.3 LTS x86_64 [9];

– оперативная память: 16 Гб;

– процессор: 11th Gen Intel® Core™ i7-1185G7 @ 3.00ГГц × 8.

4.2 Время выполнения алгоритмов

Время работы алгоритмов измерялось с использованием функции *clock_gettime* из встроенного модуля *ctime*.

Замеры времени для каждого размера матрицы проводились 1000 раз. На вход подавались случайно сгенерированные матрицы заданного размера.

Таблица 4.1 – Замер времени для матриц размером от 8 до 256

Линейный размер, штуки	**Время, мкс**			
	Классический	**Виноград**	**Виноград (опт)**	**Штрассен**
8	6.21	6.35	6.19	6.22
10	14.27	10.63	9.60	49.77
16	44.47	37.21	34.00	47.09
20	92.43	78.02	66.18	364.69
32	363.93	298.19	251.25	365.21
40	707.57	565.64	489.92	2 820.93
50	1 352.12	1 109.68	927.32	2 884.98
64	2 867.77	2 238.58	1 920.84	2 877.87
80	5 463.36	4 403.55	3 695.43	22 619.08
128	27 344.14	21 422.10	18 168.62	25 578.16
256	218 888.80	167 395.60	133 006.20	178 033.00

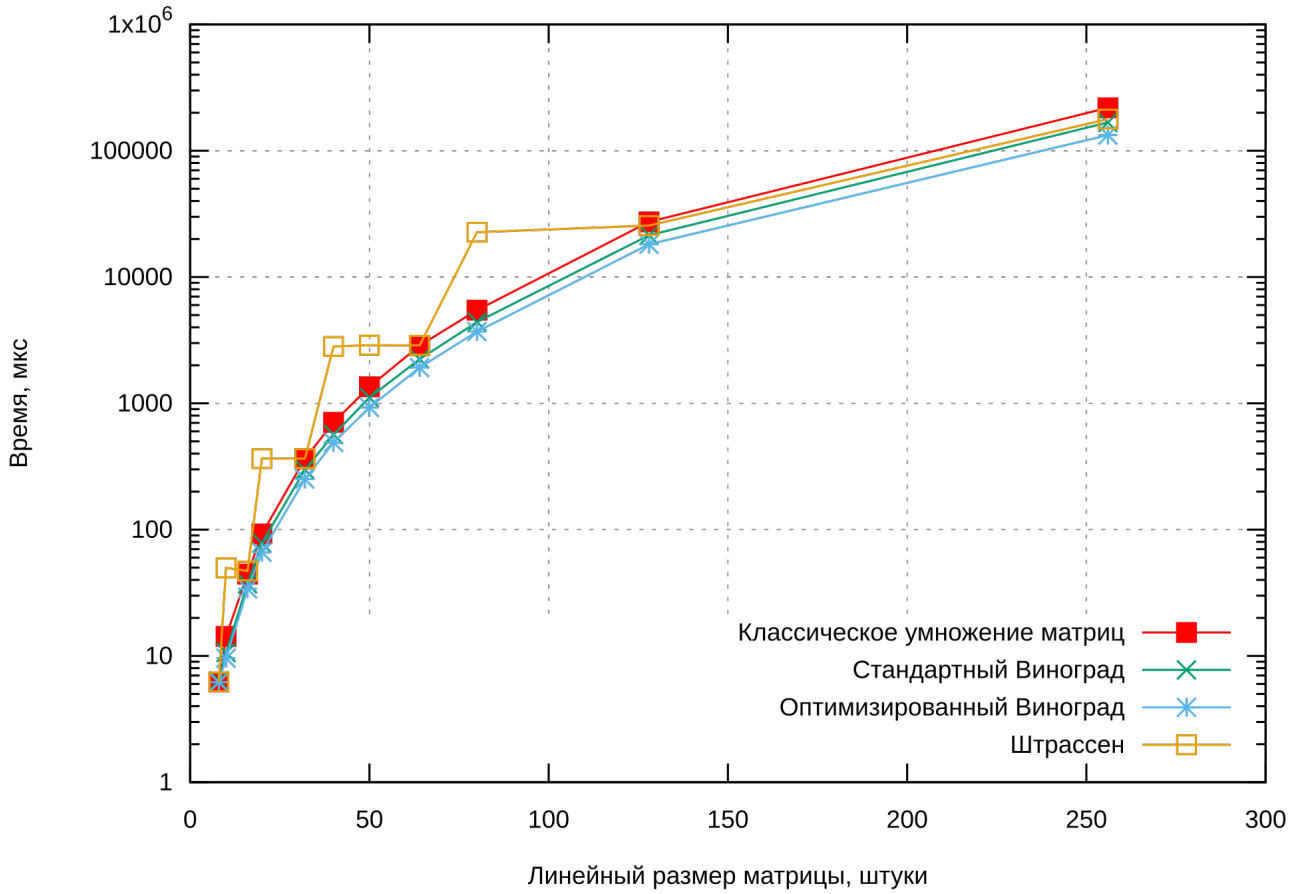

Рисунок 4.1 – Результаты замеров времени работы алгоритмов для матриц размером от 8 до 256

Исходя из полученных в таблице 4.1 данных можно понять, что наиболее быстрым алгоритмом умножения из всех четырех является оптимизированный алгоритм Винограда: на больших размерах он работает в 1.64 раза быстрее классического алгоритма, в 1.25 раз быстрее своей стандартной версии, в 1.33 раза быстрее алгоритма Штрассена.

Алгоритм Штрассена оказался самым неэффективным по времени среди всех алгоритмов: на размерах матриц, отличных от степени двойки, он проигрывает всем остальным алгоритмам, а на больших размерах, являющихся размерами двойки, данный алгоритм быстрее классического всего в 1.22 раза. Скачок на графике 4.1 у алгоритма Штрассена связан с тем, что на размерах, отличных от степени двойки, производится перевыделение памяти и увеличения размеров матриц до ближайшей большей степени двойки. Выиграть по времени у алгоритма Винограда и его оптимизированной версии не получилось ни на

одной размерности матриц.

4.3 Использование памяти

Таблица 4.2 – Замер памяти для матриц размером от 10 до 100

Линейный размер, штуки	**Память, Кб**			
	Классический	**Виноград**	**Виноград (опт)**	**Штрассен**
10	1.51	1.63	1.64	17.45
20	5.26	5.46	5.47	88.61
30	11.36	11.63	11.64	106.89
40	19.79	20.15	20.16	438.78
50	30.57	31.01	31.06	481.90
60	43.70	44.21	44.22	534.40
70	59.17	59.76	59.77	2 031.60
80	76.98	77.65	77.66	2 120.66
90	97.13	97.88	97.89	2 221.45
100	119.63	120.46	120.46	2 333.95

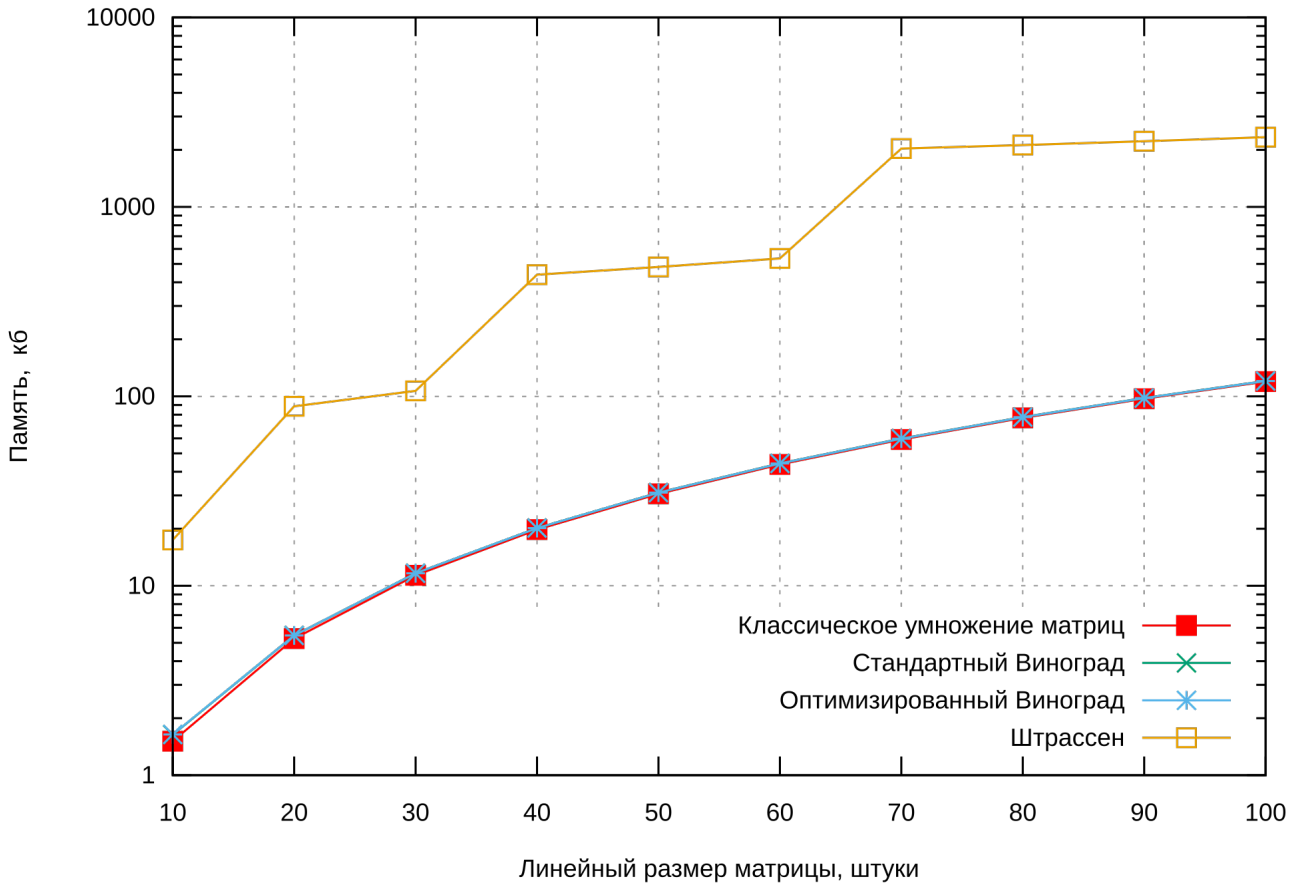

Рисунок 4.2 – Результаты замеров расходов памяти алгоритмов для матриц размером от 10 до 100

Анализируя таблицу 4.2 можно увидеть, что самым эффективным по памяти является классический алгоритм. Это обусловлено тем, что в этом алгоритме нет дополнительных переменных, которые нужны в других алгоритмах.

Алгоритм Штрассена, как и в случае с оценкой алгоритмов по времени, является самым не эффективным: при размере матриц 10×10 он расходует памяти в среднем в 11 раз больше, чем любой другой алгоритм. Это связано с тем, что при каждом рекурсивном вызове для подматриц выделяется память под их хранение, а также выделяется память для хранения матриц M_i, $i = \overline{1, 7}$.

Вывод

В данном разделе были проведены замеры времени работы, а также расчеты используемой памяти реализаций алгоритмов умножения матриц.

Исходя из результатов, полученных при оценках памяти и времени работы алгоритмов, можно сказать, что самым эффективным по времени и по памяти является оптимизированная версия алгоритма Винограда, а самым неэффективным по тем же характеристикам является алгоритм Штрассена.

Применение оптимизаций замены умножения на 2 на побитовый сдвиг влево и замены выражения вида $x = x + k$ на выражение вида $x+= k$ в алгоритме Винограда позволили существенно уменьшить время работы алгоритма по сравнению с его неоптимизированной версией, однако расход используемой памяти был незначительно увеличен дополнительной 4-х байтовой целочисленной переменной.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были решены следующие задачи:

1) Изучены и описаны алгоритмы классического умножения матриц и умножения матриц с использованием алгоритма Винограда и Штрассена.

2) Создано программное обеспечение, реализующее следующие алгоритмы:

   – классический алгоритм умножения матриц;

   – умножение матриц с использованием алгоритма Винограда;

   – умножение матриц с использованием оптимизированной версии алгоритма Винограда;

   – умножение матриц с использованием алгоритма Штрассена.

3) Проведен анализ эффективности реализаций алгоритмов по памяти и по времени.

4) Проведена оценка сложности алгоритмов и сказать влияние оптимизаций на характеристики программной реализации.

5) Подготовлен отчет по лабораторной работе.

Цель данной лабораторной работы, а именно исследование классического алгоритма умножения матриц, умножения матриц с использованием алгоритма Винограда и его оптимизированной версии согласно варианту, а также умножения матриц с использованием алгоритма Штрассена, также была достигнута.

Согласно теоретическим расчетам трудоемкости алгоритмов умножения матриц наименее трудоемким оказался алгоритм Штрассена, наиболее трудоемким – неоптимизированная версия алгоритма Винограда, однако результаты оценок работы алгоритмов по памяти и по времени оказались противоположными: оптимизированная версия алгоритма Винограда оказалась самой эффективной по времени среди всех алгоритмов, в то время как алгоритм Штрассена оказался самым неэффективным по времени. По количеству расходуемой памяти самым

эффективным оказался классический алгоритм, а самым неээффективным – алгоритм Штрассена. Это связано с тем, что в классическом алгоритме умножения двух матриц не происходит вычисления промежуточных слагаемых, в то время как при каждом рекурсивном вызове в алгоритме Штрассена для подматриц выделяется память под их хранение, а также выделяется память для хранения матриц M_i, $i = \overline{1,7}$.

Для двух матриц порядка $n \times n$ алгоритм Винограда имеет асимптотическую сложность $O(n^{2.3755})$, классический алгоритм – $O(n^3)$, алгоритм Штрассена – $O(n^{2.807})$.

Применение оптимизаций в алгоритме Винограда позволили уменьшить время работы алгоритма, но незначительно увеличить расход используемой памяти.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Матрица, её история и применение [Электронный ресурс]. — Режим доступа: https://urok.1sept.ru/articles/637896 (дата обращения: 11.10.2022).
2. Развитие алгоритмов матричного умножения [Электронный ресурс]. — Режим доступа: https://novainfo.ru/article/18648 (дата обращения: 4.11.2022).
3. Умножение матриц [Электронный ресурс]. — Режим доступа: https://algolib.narod.ru/Math/Matrix.html (дата обращения: 04.11.2023).
4. Реализация алгоритма умножения матриц по винограду на языке Haskell [Электронный ресурс]. — Режим доступа: https://cyberleninka.ru/article/n/realizatsiya-algoritma-umnozheniya-matrits-po-vinogradu-na-yazyke-haskell (дата обращения: 11.10.2022).
5. Алгоритм Штрассена для умножения матриц [Электронный ресурс]. — Режим доступа: https://elibrary.ru/download/elibrary_23140890_32362344.pdf (дата обращения: 29.10.2022).
6. Метод Штрассена [Электронный ресурс]. — Режим доступа: https://elib.belstu.by/bitstream/123456789/31480/1/Cherenkov_Metod.pdf (дата обращения: 29.10.2022).
7. Справочник по языку C++ [Электронный ресурс]. — Режим доступа: https://learn.microsoft.com/ru-ru/cpp/cpp/cpp-language-reference?view=msvc-170 (дата обращения: 28.09.2022).
8. clock_getres [Электронный ресурс]. — Режим доступа: https://pubs.opengroup.org/onlinepubs/9699919799/functions/clock_getres.html (дата обращения: 28.09.2022).
9. Ubuntu 22.04.3 LTS (Jammy Jellyfish) [Электронный ресурс]. — Режим доступа: https://releases.ubuntu.com/22.04/ (дата обращения: 28.09.2022).