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 дисциплине «Анализ Алгоритмов»

Тема

Студент Козырнов А.Д.

Группа ИУ7-52Б

Преподаватель Волкова Л. Л., Строганов Д.В.

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Матрицы представляют собой таблицы чисел, взаимосвязанных между собой [1].

Цель лабораторной работы — исследование алгоритмов умножения матриц следующими методами:

- классическим алгоритмом;
- алгоритмом Винограда;
- оптимизированного алгоритма Винограда.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- реализовать указанные алгоритмы;
- сравнение требуемого времени выполнения алгоритмов в тиках процессора;
- описать и обосновать полученные результаты.

1 Аналитическая часть

В данном разделе будут рассмотрены алгоритмы умножения матриц.

1.1 Описание алгоритмов

Пусть даны матрицы A с размерами $N \times M$ и B с размерами $M \times K$. В результате умножения матрицы A на матрицу B получается матрица C с размером $N \times K$.

1.1.1 Классический алгоритм умножения матриц

Пусть даны матрицы A размерностью $n \times m$ и матрица B размерностью $m \times k$:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1k} \\ b_{21} & b_{22} & \dots & b_{2k} \\ \dots & \dots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mk} \end{pmatrix} \tag{1.1}$$

Тогда умножением матрицы A на матрицу B называется, где матрица C:

$$C = A \times B = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1k} \\ c_{21} & c_{22} & \dots & c_{2k} \\ \dots & \dots & \ddots & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nk} \end{pmatrix}, \tag{1.2}$$

где

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj} \quad (i = \overline{1,n},\ j = \overline{1,k}) \tag{1.3}$$

1.1.2 Алгоритм Винограда умножения матриц

Пусть даны матрицы A и B, имеющие размерность 4×4.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix} \tag{1.4}$$

Для получение очередного элемента c_{ij} матрицы C в классическом алгоритме умножения матрицы выполняется по формуле:

$$c_{ij} = \begin{pmatrix} a_{n1} & a_{n2} & a_{n3} & a_{n4} \end{pmatrix} \times \begin{pmatrix} b_{j1} \\ b_{j2} \\ b_{j3} \\ b_{j4} \end{pmatrix}, \tag{1.5}$$

где $a_{ni},\ i = \overline{1,4}$ - элементы n-ой строки матрицы A; $b_{jk},\ k = \overline{1,4}$ - элементы j-ого столбца матрицы B.

В алгоритме Винограда для ускорения рассчетов снижается доля дорогих операций (умножения) и заменой их на сложение. Для достижения этой цели выполняется предварительная обработка. Запоминаются значения, что позволит заменить некоторые умножения сложением. Таким образом:

$$c_{ij} = (a_{n1}+b_{j2})(a_{n2}+b_{j1})+(a_{n3}+b_{j4})(a_{n4}+b_{j3})-a_{n1}a_{n2}-a_{n3}a_{n4}-b_{j1}b_{j2}-b_{j3}b_{j4}, \tag{1.6}$$

где элементы $a_{n1}a_{n2},\ a_{n3}a_{n4},\ b_{j1}b_{j2},\ b_{j3}b_{j4}$ - значения, которые получаются в предварительной обработке.

Вывод

В данном разделе были рассмотрены алгоритмы умножения матриц. Основные различия между алгоритмами - наличие предварительной обработки и количество операций умножения.

2 Конструкторская часть

В данном резделе будут представлены схемы алгоритмов умножения матриц, приведены оценки трудоемкости алгоритмов.

2.1 Представление алгоритмов

На рисунках 2.2 — 2.3 представлен классический алгоритм умножения матриц. На рисунках 2.2 — 2.5 представлены два алгоритма умножения матриц Винограда — обычный и оптимизированный.

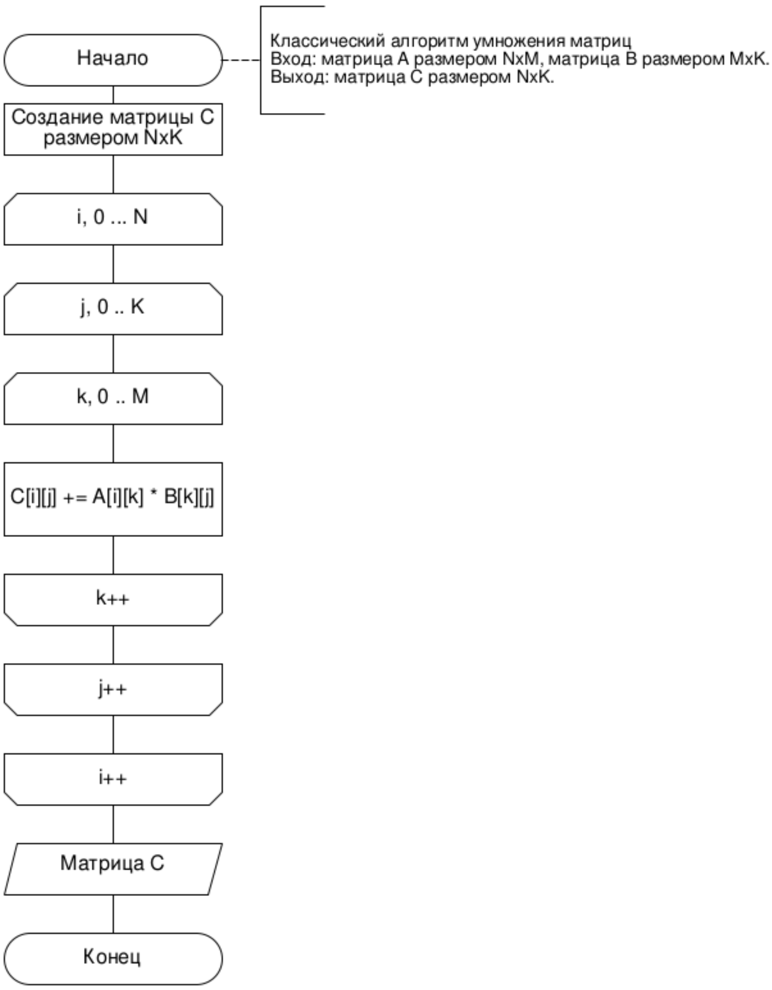

Рисунок 2.1 – Классический алгоритм умножения матриц

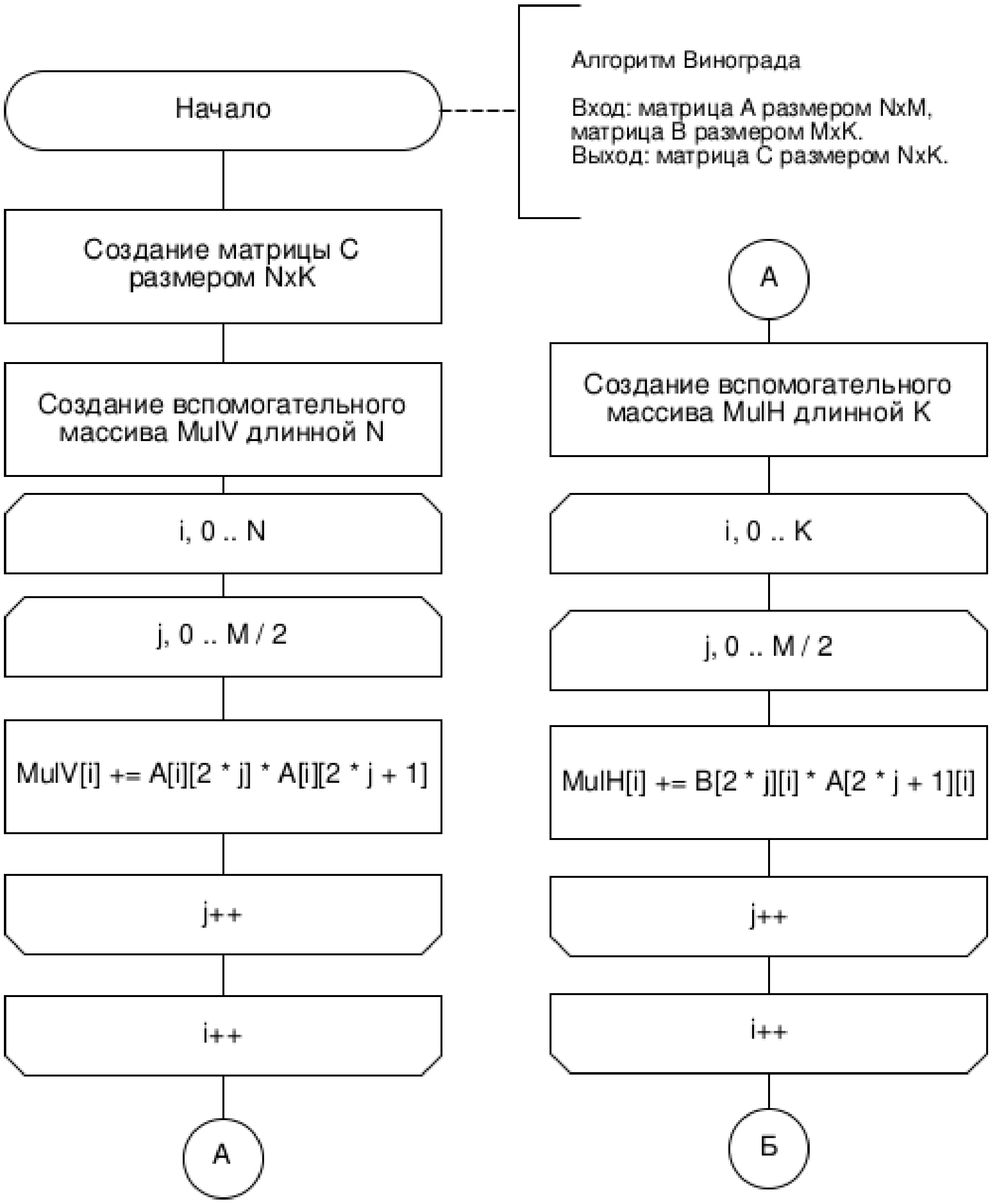

Рисунок 2.2 – Алгоритм Винограда. Часть 1

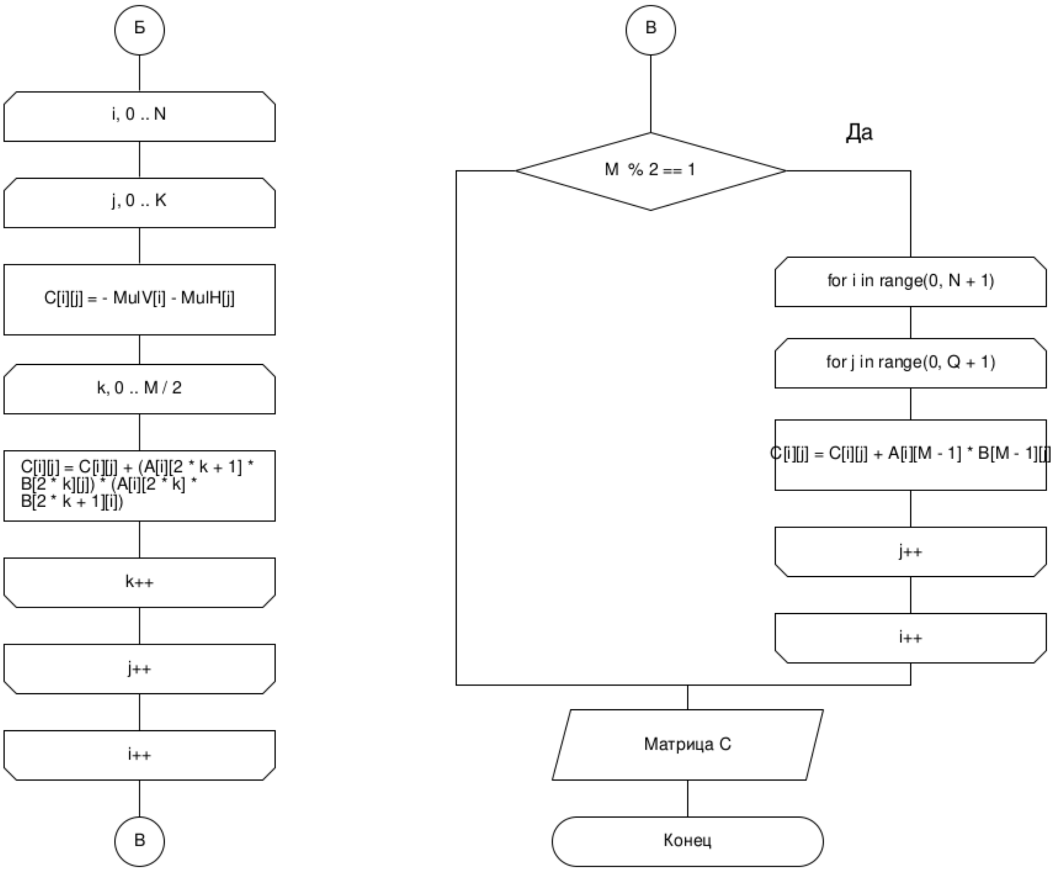

Рисунок 2.3 – Алгоритм Винограда. Часть 2

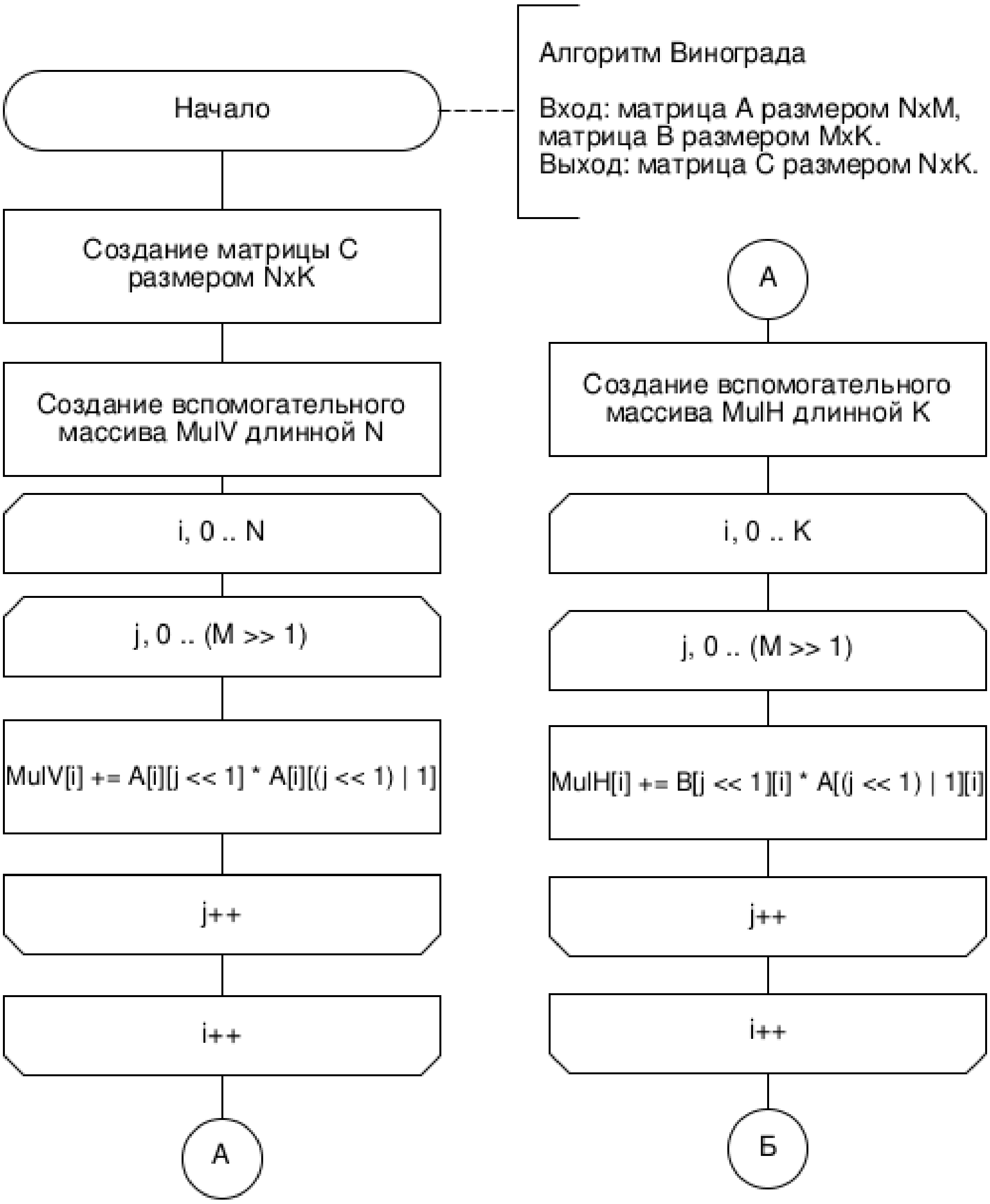

Рисунок 2.4 – Оптимизированный алгоритм Винограда. Часть 1

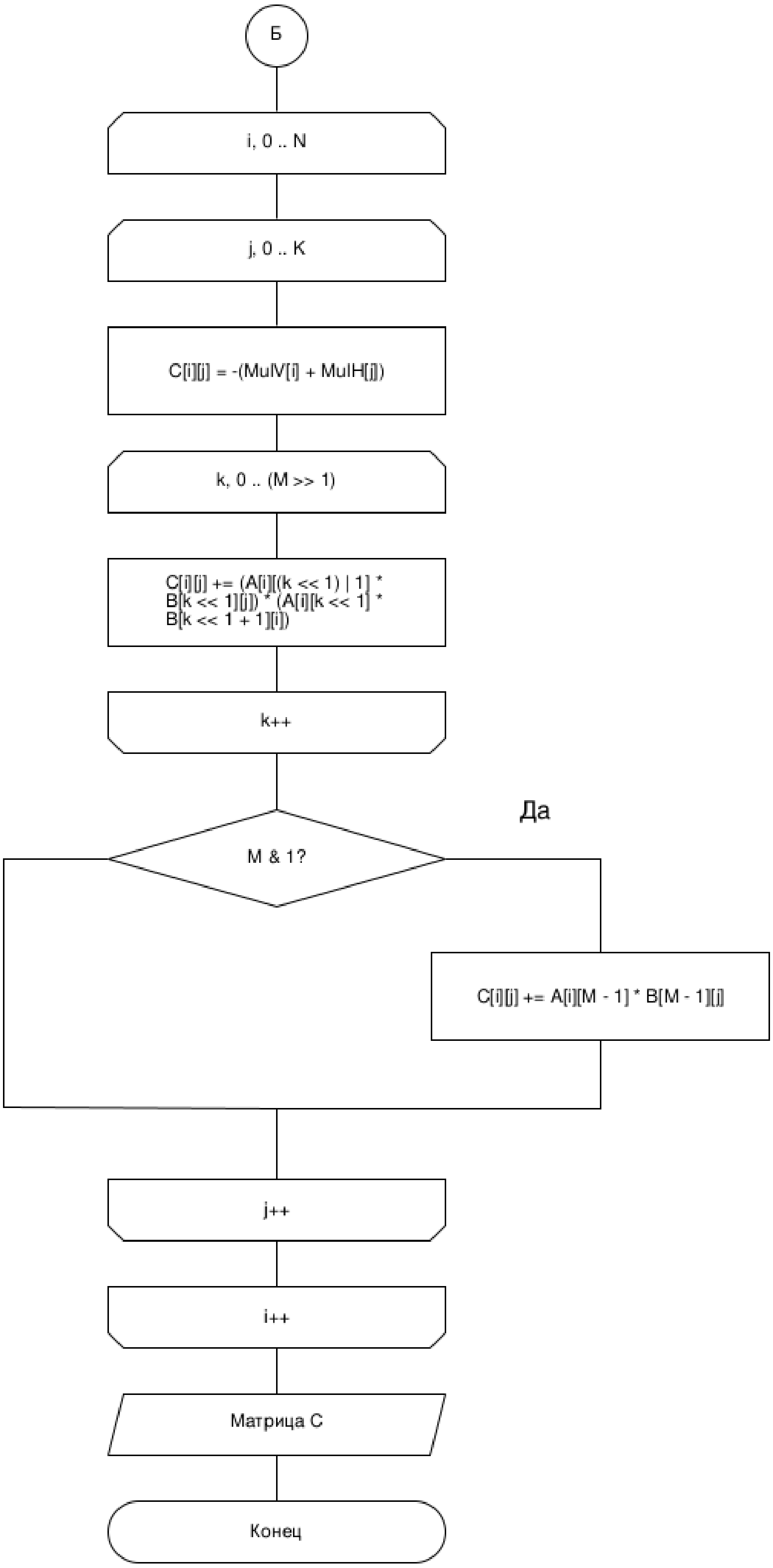

Рисунок 2.5 – Оптимизированный алгоритм Винограда. Часть 2

2.2 Трудоемкость алгоритмов

В следующих подразделах будут рассмотрены трудоемкость алгоритмов умножения матриц. Во всех последующих алгоритмах не будем учитывать инициализацию матрицы, в которую записывается результат, потому что данное действие есть во всех алгоритмах.

2.2.1 Классический алгоритм

Трудоемкость

— внешнего цикла $i = \overline{1, N}$:

$$f_i = 2 + N \cdot (f_{ibody} + 2); \tag{2.1}$$

— внутреннего цикла $j = \overline{1, K}$:

$$f_{ibody} = 2 + K \cdot (f_{jbody} + 2); \tag{2.2}$$

— внутреннего цикла $k = \overline{1, M}$:

$$f_{jbody} = 2 + M \cdot (12 + 2). \tag{2.3}$$

Тогда, если объединить (2.1) — (2.3) получим:

$$f = 2 + N \cdot (4 + K \cdot (4 + 14M)) \tag{2.4}$$

Или, если упростить:

$$f = 14NKM + 4NK + 4N + 2 \approx 14NKM \tag{2.5}$$

2.2.2 Алгоритм Винограда

Трудоемкость

— создание и инициализация массивов $MulH$ и $MulV$:

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

— Заполнение массива $MulH$:

$$f_H = 2 + N(2 + \frac{M}{2} \cdot 17); \tag{2.7}$$

— Заполнение массива $MulV$:

$$f_V = 2 + K(2 + \frac{M}{2} \cdot 17); \tag{2.8}$$

— основной цикл умножения для четных размеров матриц:

$$f_{cycle} = 2 + N(2 + K(11 + 14M)) \tag{2.9}$$

— цикла умножения последней нечетной строки и последнего нечетного столбца:

$$f_{odd} = \begin{cases} 2, & \text{четный размер} \\ 2 + N(2 + 14K), & \text{иначе} \end{cases} \tag{2.10}$$

Для нечетного (худшего) случая трудоемкость будет:

$$f_1 = f_{init} + f_H + f_V + f_{cycle} + f_{odd} \approx 14NMK \tag{2.11}$$

2.2.3 Оптимизированный алгоритм Винограда

Трудоемкость

— создание и инициализация массивов $MulH$ и $MulV$:

$$f_{init} = M + N; \tag{2.12}$$

— Заполнение массива $MulH$:

$$f_H = 2 + N(2 + \frac{M}{2} \cdot 11); \tag{2.13}$$

— Заполнение массива $MulV$:

$$f_V = 2 + K(2 + \frac{M}{2} \cdot 11); \tag{2.14}$$

— цикл умножения для четных размеров матриц:

$$f_{cycleeven} = 2 + N(2 + K(11 + 8.5 \cdot M)) \tag{2.15}$$

— цикл умножения для нечетных размеров матриц:

$$f_{cycleodd} = 2 + N(2 + K(22 + 8.5 \cdot M)) \tag{2.16}$$

Для нечетного (худшего) случая трудоемкость будет:

$$f_1 = f_{init} + f_H + f_V + f_{cycleodd} \approx 8.5 \cdot NMK \tag{2.17}$$

ВЫВОД

В данном разделе были рассмотрены представления алгоритмов и трудоемкость их выполнения.

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинги кода

3.1 Требования к программному обеспечению

Входные данные: матрица A и B, где количество столбцов первой матрицы равно количеству строк второй матрицы;

Выходные данные: матрица C.

3.2 Средства реализации

Для реализации алгоритмов был выбран язык C, потому что можно просто контролировать количество выделяемой и используемой памяти в микронотроллере.

3.3 Реализация алгоритмов

В листингах 3.1 — 3.3 предоставлены листинги реализации алгоритмов умножения матриц.

Листинг 3.1 – Классический алгоритм умножения матриц

```
matrix_t *matrix_mult(const matrix_t *A, const matrix_t *B)
{
    if (!A || !B || A->cols != B->rows)
    {
        return NULL;
    }
    matrix_t *C = matrix_alloc(A->rows, B->cols);
    if (!C)
    {
        return NULL;
    }
```

```c
    float **data_C = C->data;
    float **data_A = A->data;
    float **data_B = B->data;
    for (size_t i = 0; i < A->rows; i++)
    {
        for (size_t j = 0; j < B->cols; j++)
        {
            // data_C[i][j] = 0.0f;
            for (size_t k = 0; k < A->cols; k++)
            {
                data_C[i][j] = data_C[i][j] + data_A[i][k] *
                    data_B[k][j];
            }
        }
    }
    return C;
}
```

Листинг 3.2 – Алгоритм умножения матриц Винограда

```c
matrix_t *matrix_mult_vinograd(const matrix_t *A, const matrix_t *B)
{
    if (!A || !B || A->cols != B->rows)
    {
        return NULL;
    }
    size_t N = A->rows, M = A->cols, K = B->cols;
    float *MulH = (float *)calloc(N, sizeof(float));
    if (!MulH)
    {
        return NULL;
    }
    float *MulV = (float *)calloc(K, sizeof(float));
    if (!MulV)
    {
        free(MulH);
        return NULL;
    }
    matrix_t *C = matrix_alloc(A->rows, B->cols);
    if (!C)
    {
        free(MulH);
```

```
        free(MulV);
        return NULL;
    }
    float **data_C = C->data;
    float **data_A = A->data;
    float **data_B = B->data;
    for (size_t i = 0; i < N; i++)
    {
        for (size_t k = 0; k < M / 2; k++)
        {
            MulH[i] = MulH[i] + data_A[i][2 * k] * data_A[i][2 * k + 1];
        }
    }
    for (size_t i = 0; i < K; i++)
    {
        for (size_t k = 0; k < M / 2; k++)
        {
            MulV[i] = MulV[i] + data_B[2 * k][i] * data_B[2 * k + 1][i];
        }
    }
    for (size_t i = 0; i < N; i++)
    {
        for (size_t j = 0; j < K; j++)
        {
            data_C[i][j] = -MulH[i] - MulV[j];
            for (size_t k = 0; k < M / 2; k++)
            {
                data_C[i][j] = data_C[i][j] +
                                              (data_A[i][2 * k + 1] +
                                                 data_B[2 * k][j]) //*
                                            * (data_A[i][2 * k] +
                                               data_B[2 * k + 1][j]);
            }
        }
    }
    if (M % 2) //if M is odd
    {
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < K; j++)
            {
```

```
                    data_C[i][j] = data_C[i][j] + data_A[i][M - 1] *
                        data_B[M - 1][j];
                }
            }
        }
        free(MulH);
        free(MulV);
        return C;
}
```

Листинг 3.3 – Оптимизированный алгоритм умножения матриц Винограда

```
matrix_t *matrix_mult_vinograd_opt(const matrix_t *A, const matrix_t *B)
{
    if (!A || !B || A->cols != B->rows)
    {
        return NULL;
    }
    size_t N = A->rows, M = A->cols, K = B->cols;
    size_t M2 = (M >> 1);
    float *MulH = (float *)calloc(N, sizeof(float));
    if (!MulH)
    {
        return NULL;
    }
    float *MulV = (float *)calloc(K, sizeof(float));
    if (!MulV)
    {
        free(MulH);
        return NULL;
    }
    matrix_t *C = matrix_alloc(A->rows, B->cols);
    if (!C)
    {
        free(MulH);
        free(MulV);
        return NULL;
    }
    float **data_C = C->data;
    float **data_A = A->data;
    float **data_B = B->data;
    for (size_t i = 0; i < N; i++)
```

```
    {
        for (size_t k = 0; k < M2; k++)
        {
            MulH[i] += data_A[i][k << 1] * data_A[i][(k << 1) | 0x1];
        }
    }
    for (size_t i = 0; i < K; i++)
    {
        for (size_t k = 0; k < M2; k++)
        {
            MulV[i] += data_B[(k << 1)][i] * data_B[(k << 1) | 0x1][i];
        }
    }
    for (size_t i = 0; i < N; i++)
    {
        for (size_t j = 0; j < K; j++)
        {
            data_C[i][j] -= (MulH[i] + MulV[j]);
            for (size_t k = 0; k < M2; k++)
            {
                data_C[i][j] +=
                    (data_A[i][(k << 1) | 0x1] + data_B[(k << 1)][j]) //*
                    * (data_A[i][(k << 1)] + data_B[(k << 1) | 0x1][j]);
            }
            if (M & 1) //if M is odd
            {
                data_C[i][j] += data_A[i][M - 1] * data_B[M - 1][j];
            }
        }
    }
    free(MulH);
    free(MulV);
    return C;
}
```

ВЫВОД

В данном разделе были представлены средства реализации, требования к программному обеспечению, технические характеристики и реализация алгоритмов.

4 Исследовательская часть

4.1 Технические характеристики

Характеристики используемого оборудования:

- Микроконтроллер STM32F303 [2]

4.2 Время выполнения алгоритмов

В таблице 4.1 приведено время выполнения алгоритмов для разных квадратных матриц. На рисунке 4.1 показаны графики выполнения алгоритмов умножения матриц в тиках процессора.

Таблица 4.1 – Время работы алгоритмов (в тиках процессора)

Размер матриц	Классический	Виноград	Виноград оптимизированный
2	0.98	0.75	0.73
3	0.83	0.79	0.74
4	1.01	0.90	0.90
5	1.18	1.15	1.23
6	1.68	1.56	1.50
7	2.10	2.00	2.05
8	2.80	2.46	2.49
9	3.72	3.21	3.36
10	5.60	4.18	5.36
10	5.04	4.50	4.50
21	41.45	30.15	31.04
32	143.91	91.93	90.18
43	316.11	213.24	218.61
54	592.50	367.35	403.39
65	937.01	608.54	620.80
76	1531.09	960.95	967.34
87	2216.74	1423.29	1448.46
98	3178.86	2044.33	2059.81
109	4462.48	2832.23	2927.09

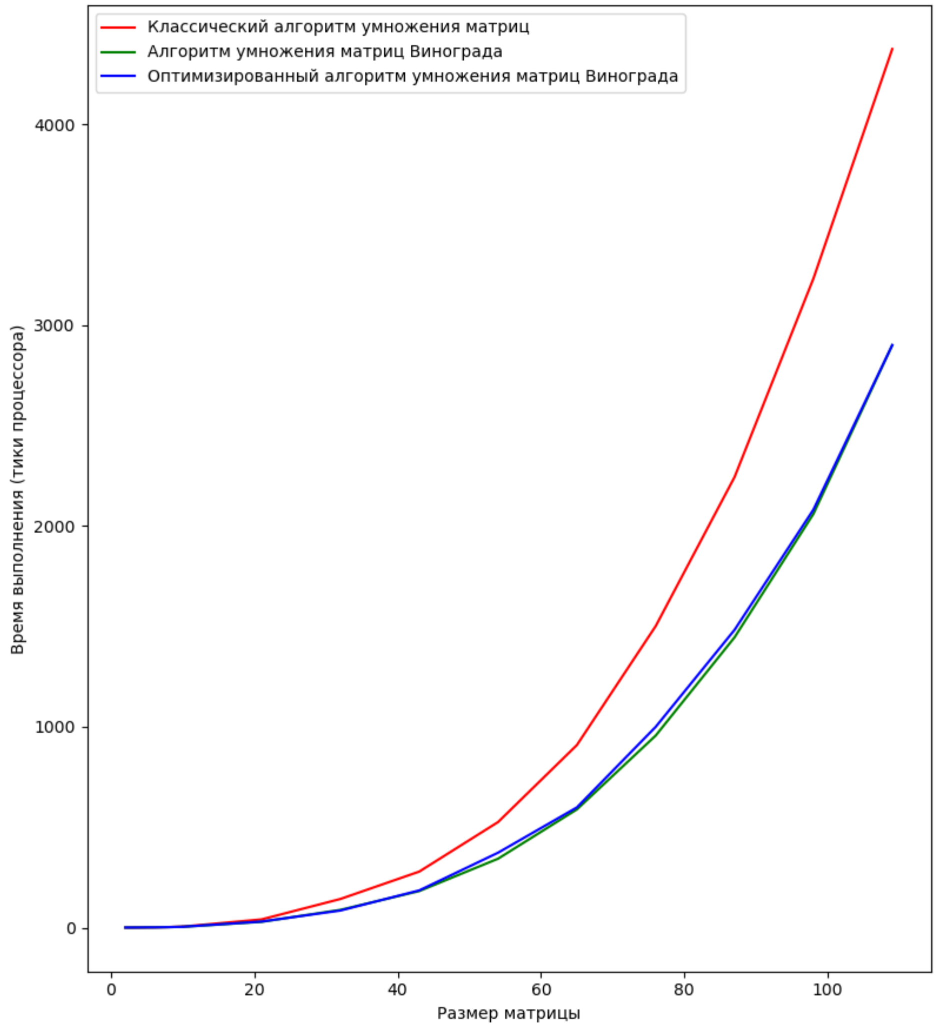

Рисунок 4.1 – Графики в тиках процессора

4.3 Вывод

Сравнения проводились на квадратных матрицах четного и нечетного размеров. Во всех случаях классический алгоритм проигрывает алгоритму Винограда по количеству необходимых тиков процессора на выполнения задачи. Оптимизированная и обычная версии алгоритма Винограда используют довольно одинаковое количество тиков процессора, отчего можно сказать, что их производительность примерно одинаковая. Возможно, это произошло по причине оптимизации компилятора.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной лабораторной работы были решены следующие задачи:

— реализованы указанные алгоритмы;

— сравнено требуемое время выполнения алгоритмов в тиках процессора;

— описаны и обоснованы полученные результаты.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Матрицы: примеры [Электронный ресурс] URL: https://spravochnick.ru/matematika/matricy_primery_s_resheniem_i_obyasneniem/ (дата обращения: 27.09.24)

[2] Микроконтроллер STM32F303 Discovery [Электронный ресурс] URL: https://www.st.com/en/evaluation-tools/stm32f3discovery.html#st-also-like (дата обращения: 27.09.24)