Санкт-Петербургский государственный университет

Программная инженерия

Группа 24.М71-мм

Блочное Перемножение Матриц

Альшаеб Басель

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
ст. преподаватель

Санкт-Петербург
2025

Оглавление

Постановка задачи

Необходимо реализовать программу перемножения двух нижне-треугольных матриц A и B, результатом которой будет матрица C, вычисляемая по формуле:

$$C = A \times B \tag{1}$$

$$A = \begin{bmatrix} A_{11} & 0 & 0 & \cdots & 0 \\ A_{21} & A_{22} & 0 & \cdots & 0 \\ A_{31} & A_{32} & A_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & A_{n3} & \cdots & A_{nn} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & 0 & 0 & \cdots & 0 \\ B_{21} & B_{22} & 0 & \cdots & 0 \\ B_{31} & B_{32} & B_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ B_{n1} & B_{n2} & B_{n3} & \cdots & B_{nn} \end{bmatrix}$$

Каждая матрица имеет размерность $N \times N$.

В отчёте рассматриваются три версии реализации на языке C:

1. **Версия 1:** Матрицы хранятся в виде двумерных массивов, при этом нулевые элементы не участвуют в вычислениях.

2. **Версия 2:** Аналогично первой версии, но с использованием блочной обработки (blocking), улучшающей эффективность кэширования.

3. **Версия 3:** Матрицы хранятся в виде одномерных массивов: матрица A — в блочно-строчном порядке (row-major), а матрица B — в блочно-столбцовом порядке (column-major).

Во всех версиях исключаются ненужные умножения на ноль, с учётом нижне-треугольной структуры исходных матриц.

1. Введение

В данной работе рассматривается задача умножения двух **нижне-треугольных квадратных матриц** A и B размером $N \times N$. Результатом операции является матрица C, также размером $N \times N$, вычисляемая по стандартной формуле матричного умножения:

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} \cdot B_{kj} \tag{2}$$

Однако с учётом того, что A и B являются **нижне-треугольными**, значительное количество элементов в этих матрицах равно нулю. Поэтому, если использовать стандартный алгоритм, будет выполнено множество **избыточных операций умножения на ноль**, что снижает эффективность программы как по скорости, так и по использованию памяти.

Для оптимизации этой задачи были применены следующие подходы:

1.1. Избежание вычислений с нулями

Вместо полной обработки всех N^3 операций в формуле, программа учитывает только **ненулевые элементы**, опираясь на свойства нижне-треугольных матриц:

- Для матрицы A: $A_{ik} = 0$, если $k > i$
- Для матрицы B: $B_{kj} = 0$, если $j > k$

Следовательно, при вычислении C_{ij}, где $j > i$, вся строка становится нулевой, и её можно не вычислять вообще. Сокращённая формула:

$$C_{ij} = \sum_{k=j}^{i} A_{ik} \cdot B_{kj}, \quad \text{только если } j \leq i \tag{3}$$

1.2. Эффективное хранение с использованием одномерных массивов

Чтобы дополнительно **сэкономить память** и повысить **локальность доступа к данным**, нижне-треугольные матрицы A и B хранятся в виде **одномерных массивов**:

- Матрица A хранится в блочно-строчном порядке (block-row order)
- Матрица B хранится в блочно-столбцовом порядке (block-column order)

Такой способ хранения позволяет избежать выделения памяти под нули и упростить блочную обработку, особенно при реализации кэш-оптимизированных и параллельных алгоритмов. Количество значащих (ненулевых) элементов в каждой из матриц равно:

$$\frac{N(N+1)}{2} \tag{4}$$

Что даёт выигрыш по памяти почти в 2 раза по сравнению с полными N^2 элементами, особенно при больших N.

2. Размещение матриц

При реализации перемножения двух нижнетреугольных матриц мы использовали два различных способа хранения данных: двумерное (2D) и одномерное (1D) представления.

2.1. Двумерный массив (2D array)

Это традиционный способ, в котором вся матрица хранится как массив $N \times N$, включая нули выше главной диагонали:

$$A = \begin{bmatrix} A_{11} & 0 & 0 & 0 \\ A_{21} & A_{22} & 0 & 0 \\ A_{31} & A_{32} & A_{33} & 0 \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}$$

В памяти такая матрица хранится в виде:

A11, 0, 0, 0, A21, A22, 0, 0, A31, A32, A33, 0, A41, A42, A43, A44

Листинг 1: Инициализация нижнетреугольной матрицы в 2D массиве

```
double **A = (double**)malloc(N * sizeof(double*));
for (int i = 0; i < N; i++) {
    A[i] = (double*)malloc(N * sizeof(double));
}
for (int i = 0; i < N; i++) {
    for (int j = 0; j <= i; j++) {
        A[i][j] = ((double)rand() / RAND_MAX) * 10.0;
    }
}
```

Хотя этот способ удобен, он неэффективен с точки зрения использования памяти, поскольку хранит ненужные нули.

2.2. Одномерный массив (1D array)

Чтобы избежать хранения нулей, мы применили два различных варианта размещения элементов нижнетреугольной матрицы в памяти:

Матрица A — блочно-построчное хранение (block-row order)

В этом представлении ненулевые элементы хранятся последовательно по строкам:

Листинг 2: Инициализация матрицы A в 1D массиве

```
#define EL (N * (N + 1) / 2)

double* init_matrix_A() {
    double *A = (double*)malloc(EL * sizeof(double));
    for (int i = 0, k = 0; i < N; i++) {
        for (int j = 0; j <= i; j++, k++) {
            A[k] = ((double)rand() / RAND_MAX) * 10.0;
        }
    }
    return A;
}
```

Матрица B — блочно-по-столбцам (block-column order)

Для лучшего кэширования при блочном перемножении матрица B хранится в колонном порядке:

Листинг 3: Инициализация матрицы B в 1D массиве

```
#define EL (N * (N + 1) / 2)

double* init_matrix_B() {
    double *B = (double*)malloc(EL * sizeof(double));
    for (int i = 0, k = 0; i < N; i++) {
        for (int j = 0; j <= i; j++, k++) {
            B[(j * (2 * N - j + 1)) / 2 + (i - j)] = ((double)rand() / RAND_MAX) * 10.0;
        }
    }
    return B;
}
```

Такой подход позволяет повысить производительность за счёт упорядоченного обращения к памяти при блочной обработке.

2.3. Порядок размещения блоков в памяти

- **Матрица A (строчный порядок):** $A_{11}, A_{12}, A_{22}, A_{13}, A_{23}, A_{33}, \ldots, A_{1n_1}, A_{2n_1}, \ldots, A_{n_1n_1}$
- **Матрица B (столбцовый порядок):** $B_{11}, B_{21}, B_{31}, \ldots, B_{n_11}, B_{22}, B_{32}, \ldots, B_{n_12}, B_{33}, \ldots$

2.4. Сравнение подходов

Подход хранения	**Преимущества**	**Недостатки**
Двумерный массив (2D)	Простая реализация, доступ по индексам	Хранит лишние нули, неэффективное использование памяти
1D (строчный порядок для A)	Эффективное хранение, меньше памяти	Сложность индексации
1D (столбцовый порядок для B)	Улучшение кэширования при блочной обработке	Более сложная адресация

Таблица 1: Сравнение способов хранения матриц

3. Алгоритм Перемножения

В данной работе рассматриваются три версии алгоритма перемножения двух нижнетреугольных матриц. Каждая версия реализует один и тот же базовый алгоритм, но с различиями в представлении и доступе к элементам матриц, а также в методе блочной оптимизации.

Общий алгоритм перемножения нижнетреугольных матриц

Пусть A и B — нижнетреугольные матрицы размера $N \times N$. Тогда их произведение $C = AB$ также будет нижнетреугольной матрицей. Формула для вычисления элемента C_{ij}:

$$C_{ij} = \sum_{k=j}^{i} A_{ik} \cdot B_{kj}, \quad \text{где } 0 \leq j \leq i < N \tag{5}$$

Версия 1: Двумерный массив без блочной оптимизации

Листинг 4: Алгоритм перемножения без блокировки для двумерных массивов

```
void multiply_blocked(double **A, double **B, double ***C) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j <= i; j++) {
            for (int k = j; k <= i; k++) {
                (*C)[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Версия 2: Двумерный массив с блочной оптимизацией

Листинг 5: Алгоритм перемножения с блокировкой для двумерных массивов

```
void multiply_blocked(double **A, double **B, double ***C) {
    for (int bi = 0; bi < N; bi += BLOCK_SIZE) {
        for (int bj = 0; bj <= bi; bj += BLOCK_SIZE) {
            for (int i = bi; i < bi + BLOCK_SIZE && i < N; i++) {
                for (int j = bj; j <= i && j < bj + BLOCK_SIZE; j++) {
                    for (int k = j; k <= i; k++) {
                        (*C)[i][j] += A[i][k] * B[k][j];
                    }
                }
            }
        }
    }
}
```

Версия 3: Одномерные массивы с блочной оптимизацией и пользовательским доступом

Для доступа к элементам A и B используются функции `get_A` и `get_B`, которые учитывают способ хранения (строчный и столбцовый порядки).

Листинг 6: Алгоритм перемножения с блокировкой для 1D массивов

```
void multiply_blocked(double *A, double *B, double *C) {
    for (int i1 = 0; i1 < N; i1 += BLOCK_SIZE) {
        for (int j1 = 0; j1 <= i1; j1 += BLOCK_SIZE) {
            for (int k1 = j1; k1 <= i1; k1 += BLOCK_SIZE) {
                for (int i = i1; i < i1 + BLOCK_SIZE && i < N; i++) {
```

```
                    for (int j = j1; j <= i && j < j1 + BLOCK_SIZE; j++
                        for (int k = k1; k <= i && k < k1 + BLOCK_SIZE;
                            double a_val = get_A(A, i, k);
                            double b_val = get_B(B, k, j);
                            C[i * N + j] += a_val * b_val;
                        }
                    }
                }
            }
        }
    }
}
```

Пример работы алгоритма

Рассмотрим пример перемножения двух нижнетреугольных матриц размера 3×3:

$$A = \begin{bmatrix} a_{00} & 0 & 0 \\ a_{10} & a_{11} & 0 \\ a_{20} & a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{00} & 0 & 0 \\ b_{10} & b_{11} & 0 \\ b_{20} & b_{21} & b_{22} \end{bmatrix}$$

Тогда элемент C_{21} вычисляется как:

$$C_{21} = A_{20}B_{01} + A_{21}B_{11}$$

А элемент C_{22}:

$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

Вывод

Блочная оптимизация позволяет повысить эффективность использования кэш-памяти за счёт улучшения локальности данных. Использование одномерных массивов усложняет доступ к данным, но даёт преимущества при более эффективном использовании памяти.

4. Заключение

Проект MyGym направлен на создание удобной платформы для автоматизации управления фитнес-клубами. В документе изложены основные аспекты проекта, структура команды, сроки и критерии приемки. План будет обновляться по мере выполнения этапов.