Санкт-Петербургский государственный университет

Программная инженерия

Группа 24.М71-мм

Блочное Перемножение Матриц

Альшаеб Басель

Отчёт по учебной практике
в форме «Решение»

Преподаватель:
ст. преподаватель Штейнберг Борис Яковлевич

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:

2.2.1. 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;
}
```

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

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.

3.1. 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}:

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

3.2. 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];
            }
        }
    }
}
```

3.3. 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];
                    }
                }
            }
        }
    }
}
```

3.4. 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; k++) {
                            double a_val = get_A(A, i, k);
                            double b_val = get_B(B, k, j);
                            C[i * N + j] += a_val * b_val;
                        }
                    }
                }
            }
        }
    }
}
```

3.5. 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}:

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

3.6. 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. Доступ к элементам 1D-массивов

При хранении нижнетреугольных матриц в виде одномерных массивов важно использовать корректные формулы для доступа к элементам.

4.1. Матрица A (строчное хранение)

Матрица A хранится в памяти по строкам. Чтобы получить элемент $A[i][j]$ (если $i \geq j$), используется формула:

$$\text{index}_A = \frac{i(i+1)}{2} + j \tag{6}$$

Если $i < j$, то $A[i][j] = 0$ (так как элемент вне нижнего треугольника).

4.2. Матрица B (столбцовое хранение)

Матрица B хранится по столбцам. Индекс для доступа к элементу $B[i][j]$ (если $i \geq j$):

$$\text{index}_B = \frac{j(2N - j + 1)}{2} + (i - j) \tag{7}$$

Если $i < j$, то $B[i][j] = 0$.

4.3. Код реализации

```c
double get_A(double *A, int i, int j) {
    if (i < j) return 0.0;
    int index = (i * (i + 1)) / 2 + j;
    return A[index];
}

double get_B(double *B, int i, int j) {
    if (i < j) return 0.0;
    int index = (j * (2 * N - j + 1)) / 2 + (i - j);
    return B[index];
}
```

5. Измерение производительности и выбор размера блока

Для оценки производительности алгоритма перемножения блочных нижнетреугольных матриц мы использовали стандартную функцию `clock()` из библиотеки `time.h`. Все замеры производились при фиксированном размере матрицы $N = 2880$ и различных размерах блоков.

5.1. Фрагмент кода измерения времени

```
clock_t start = clock();
multiply_blocked(...); // вызовнужнойверсии
clock_t end = clock();
double cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
printf("\nExecution␣time:␣%f␣seconds\n", cpu_time_used);
```

5.2. Характеристики и компилятор

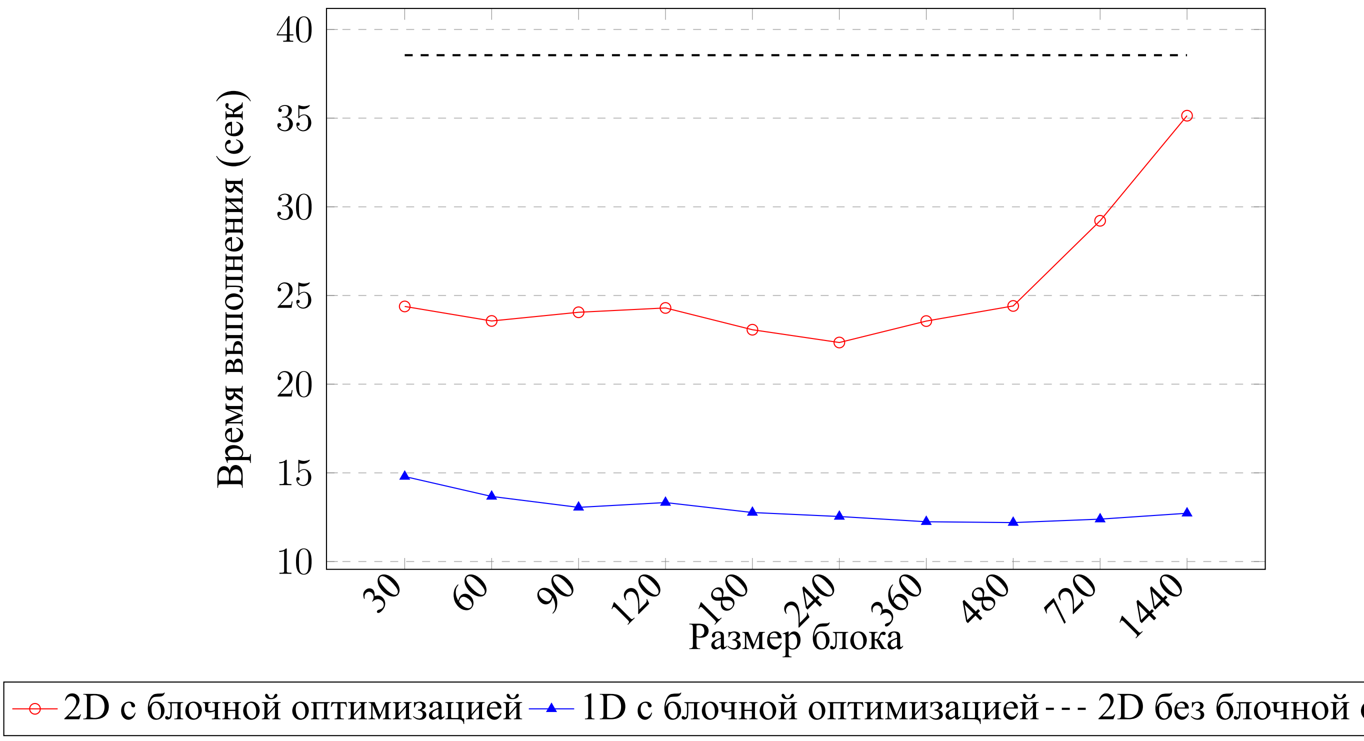
- **Компилятор:** GCC v.13.2.0, compiler option: `-Ofast`
- **ОС:** Microsoft Windows 11 Pro (Version 10.0.22631 Build 22631)
- **Процессор:** Intel® Core™ i7-10750H CPU @ 2.60GHz (up to 4.50 GHz Turbo)
  - 6 ядер и 12 потоков
  - L2: 1.5 MB
  - L3: 12 MB
- **Оперативная память (RAM):** 16 GB LPDDR4, clock speed: 2933 MHz

6. Таблица с результатами

Таблица 2: Время выполнения (в секундах) для различных размеров блоков при $N = 2880$

Размер блока	**2D без блокировки**	**2D с блокировкой**	**1D с блокировкой**
-	38.550	-	-
30	-	24.383	14.793
60	-	23.564	13.670
90	-	24.054	13.054
120	-	24.298	13.325
180	-	23.070	12.763
240	-	**22.349**	12.538
360	-	23.558	12.243
480	-	24.411	**12.195**
720	-	29.213	12.390
1440	-	35.138	12.722

7. График сравнения производительности



8. Вывод

Наилучшее время показал размер блока 64, при котором достигнута оптимальная балансировка между числом операций и эффективным использованием кэш-памяти. При слишком малых блоках увеличивается накладная нагрузка на циклы, а при слишком больших — снижается локальность данных.

9. Приложение 1

Листинг 7: Приложение 1D массивов с блочной оптимизацией

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <malloc.h>

#define N 2880

int block_sizes[] = {30, 60, 90, 120, 180, 240, 360, 480, 720, 1440};
int num_block_sizes = sizeof(block_sizes) / sizeof(block_sizes[0]);

#define EL (N * (N + 1) /2)
// init matrix A (Lower triangular, stored in block-row order)
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;
}

// init matrix B (Lower triangular, stored in block-column order)
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;
}

// matrix C (dense n×n)
double* init_matrix_C() {
    double *C = (double*)calloc(N * N, sizeof(double));
    return C;
}

// Access elements of A (stored in block-row order)
```

```c
double get_A(double *A, int i, int j) {
    if (i < j) return 0.0;
    int index = (i * (i + 1)) / 2 + j;
    return A[index];
}

// Access elements of B (stored in block-column order)
double get_B(double *B, int i, int j) {
    if (i < j) return 0.0;
    int index = (j * (2 * N - j + 1)) / 2 + (i - j);
    return B[index];
}

void multiply_blocked_optimized(double *A, double *B, double *C, int block_size) {
    for (int ii = 0; ii < N; ii += block_size) {
        for (int jj = 0; jj <= ii; jj += block_size) {
            for (int kk = jj; kk <= ii; kk += block_size) {
                // Determine actual block boundaries
                int i_end = fmin(ii + block_size, N);
                int j_end = fmin(jj + block_size, N);
                int k_end = fmin(kk + block_size, N);

                // Process current block
                for (int i = ii; i < i_end; i++) {
                    for (int j = jj; j <= i && j < j_end; j++) {
                        double sum = 0.0;
                        int k_start = fmax(kk, j);
                        for (int k = k_start; k <= i && k < k_end; k++) {
                            // Direct access to matrix elements
                            double a = A[(i*(i+1))/2 + k];
                            double b = B[(j*(2*N-j+1))/2 + (k-j)];
                            sum += a * b;
                        }
                        C[i*N + j] += sum;
                    }
                }
            }
        }
    }
}

// Print functions
void print_matrix_A(double *M, int is_triangular) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            double val = (is_triangular && j > i) ? 0.0 : get_A(M, i , j);
```

```
            printf("%6.4f␣", val);
        }
        printf("\n");
    }
}

void print_matrix_B(double *M, int is_triangular) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            double val = (is_triangular && j > i) ? 0.0 : get_B(M, i , j);
            printf("%6.4f␣", val);
        }
        printf("\n");
    }
}

void print_matrix(double *M, int is_triangular) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            double val = (is_triangular && j > i) ? 0.0 : M[i * N + j];
            printf("%6.4f␣", val);
        }
        printf("\n");
    }
}

int main() {
    srand(1);

    // Initialize matrices (A is block-row, B is block-column)
    double *A = init_matrix_A();
    double *B = init_matrix_B();
    // double *C = init_matrix_C();
    double *C = (double*)_aligned_malloc(N * N * sizeof(double), 64);
    for (int i = 0; i < N * N; i++) C[i] = 0.0;

    clock_t start,end;
    double cpu_time_used;

    // printf("Matrix A (Lower Triangular, Block-Row Order):\n");
    // print_matrix_A(A, 1);

    // printf("\nMatrix B (Lower Triangular, Block-Column Order):\n");
    // print_matrix_B(B, 1);

    for (int idx = 0;idx < num_block_sizes; idx++) {
```

```c
        int BLOCK_SIZE = block_sizes[idx];

        // Multiplication
        start = clock();
        multiply_blocked_optimized(A, B, C, BLOCK_SIZE);
        end = clock();

        // printf("\nResult Matrix C (Lower Triangular):\n");
        // print_matrix(C, 1);

        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
        printf("\nBlock size: %d , Execution time: %f seconds\n", BLOCK_SIZE, cpu_time_used);
    }
    // Free allocated memory
    free(A);
    free(B);
    free(C);

    return 0;
}
```

10. Приложение 2

Листинг 8: Приложение 2D массивов с блочной оптимизацией

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 2880
//#define BLOCK_SIZE 4

int block_sizes[] = {30, 60, 90, 120, 180, 240, 360, 480, 720, 1440};
int num_block_sizes = sizeof(block_sizes) / sizeof(block_sizes[0]);

// Function to allocate memory for matrix A (Lower triangular)
double** init_matrix() {
    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;
        }
    }
    return A;
}

// Function to allocate and initialize matrix C (dense n×n)
double** init_matrix_C() {
    double **C = (double**)calloc(N, sizeof(double));
    for (int i=0 ; i<N ; i++) {
        C[i] = (double*)calloc(N, sizeof(double));
    }

    return C;
}

void multiply_blocked(double **A, double **B, double ***C, int BLOCK_SIZE) {
    for (int bi = 0; bi < N; bi += BLOCK_SIZE) { // Block row
        for (int bj = 0; bj <= bi; bj += BLOCK_SIZE) { // Block column
            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];
                    }
```

```
                }
            }
        }
    }

}

void print_matrix(double **M) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%6.4f ", M[i][j]);
        }
        printf("\n");
    }
}

void free_matrix(double** A);

int main() {
    // Initialization of arrays
    srand(1);

    double **A = init_matrix();
    double **B = init_matrix();
    double **C = init_matrix_C();
    clock_t start,end;
    double cpu_time_used;

    // Print Matrixes
    //printf("\nMatrix A (Lower Triangular, 2-dim Array):\n");
    //print_matrix(A);

    //printf("\nMatrix B (Lower Triangular, 2-dim Array):\n");
    //print_matrix(B);

    for (int idx = 0;idx < num_block_sizes; idx++) {
        int BLOCK_SIZE = block_sizes[idx];

        start = clock();

        // Multiplication
        multiply_blocked(A, B, &C, BLOCK_SIZE);

        end = clock();
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

```
        //printf("\nResult Matrix C (Lower Triangular, 2-dim Array):\n");
        //print_matrix(C);

        printf("\nBlock size: %d , Execution time: %f seconds\n", BLOCK_SIZE,cpu_time_used);
    }

    free_matrix(A);
    free_matrix(B);
    free_matrix(C);

    return 0;
}
void free_matrix(double** A){
    for(int i = 0; i < N; i++) {
        free(A[i]);
    }
    free(A);
}
```

11. Приложение 3

Листинг 9: Приложение 2D массивов без блочной оптимизации

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 2880

// Function to allocate memory for matrix A (Lower triangular)
double** init_matrix() {
    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;
        }
    }
    return A;
}

// Function to allocate and initialize matrix C (dense n×n)
double** init_matrix_C() {
    double **C = (double**)calloc(N, sizeof(double));
    for (int i=0 ; i<N ; i++) {
        C[i] = (double*)calloc(N, sizeof(double));
    }

    return C;
}

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];
            }
        }
    }
}

void print_matrix(double **M) {
    for (int i = 0; i < N; i++) {
```

```
        for (int j = 0; j < N; j++) {
            printf("%6.4f ", M[i][j]);
        }
        printf("\n");
    }
}

void free_matrix(double** A);

int main() {
    // Initialization of arrays
    srand(1);

    double **A = init_matrix();
    double **B = init_matrix();
    double **C = init_matrix_C();
    clock_t start,end;
    double cpu_time_used;

    // Print Matrixes
    //printf("\nMatrix A (Lower Triangular, 2-dim Array):\n");
    //print_matrix(A);

    //printf("\nMatrix B (Lower Triangular, 2-dim Array):\n");
    //print_matrix(B);

    start = clock();

    // Multiplication
    multiply_blocked(A, B, &C);

    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    //printf("\nResult Matrix C (Lower Triangular, 2-dim Array):\n");
    //print_matrix(C);

    printf("\nExecution time: %f seconds\n", cpu_time_used);

    free_matrix(A);
    free_matrix(B);
    free_matrix(C);

    return 0;
}
void free_matrix(double** A){
```

```
    for(int i = 0; i < N; i++) {
        free(A[i]);
    }
    free(A);
}
```