МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра вычислительной техники

**

**ЛАБОРАТОРНАЯ РАБОТА №3**

«Оптимизация доступа к памяти»

**по дисциплине: «*Организация электронных вычислительных машин и вычислительных систем*»**

|  |  |
| --- | --- |
| Выполнил:  Студент гр. «АБ-320», «АВТФ»  *Сычук Алексей Александрович*  «18» апреля 2025г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | Проверил:  Ассистент кафедры ЗИ  *Скороходов Федор*  «\_\_\_» \_\_\_\_\_\_ 2025г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) |

Новосибирск 2025

**Задание:**

1. На языке С/С++/C# реализовать функцию DGEMM BLAS последовательное умножение двух квадратных матриц с элементами типа double. Обеспечить возможность задавать размерности матриц в качестве аргумента командной строки при запуске программы. Инициализировать начальные значения матриц случайными числами.
2. Провести серию испытаний и построить график зависимости времени выполнения программы от объёма входных данных. Например, для квадратных матриц с числом строк/столбцов 1000, 2000, 3000, … 10000.
3. Оценить предельные размеры матриц, которые можно перемножить на вашем вычислительном устройстве.
4. Реализовать дополнительную функцию DGEMM\_opt\_1, в которой выполняется оптимизация доступа к памяти, за счет построчного перебора элементов обеих матриц.
5. \* Реализовать дополнительную функцию DGEMM\_opt\_2, в которой выполняется оптимизация доступа к памяти, за счет блочного перебора элементов матриц. Обеспечить возможность задавать блока, в качестве аргумента функции.
6. \*\* Реализовать дополнительную функцию DGEMM\_opt\_3, в которой выполняется оптимизация доступа к памяти, за счет векторизации кода.
7. Оценить ускорение умножения для матриц фиксированного размера, например,1000х1000, 2000х2000, 5000х5000, 10000х10000. \* Для блочного умножения матриц определить размер блока, при котором достигается максимальное ускорение.
8. С помощью профилировщика для исходной программы и каждого способа оптимизации доступа к памяти оценить количество промахов при работе к КЭШ памятью (cache-misses).
9. Подготовить отчет, отражающий суть, этапы и результаты проделанной работы.

**Задание 1**

Для решения задачи был выбран ЯП С++. Поскольку до компиляции программы неизвестен размер матрицы, то будем использовать динамические массивы при реализации. Создание матриц и их заполнение случайными числами происходят в функциях createMatrix и fillRandomMatrix соответственно. Функции представлены на рисунке 1.1 и 1.2. С полным исходным кодом можно ознакомиться в приложении А.

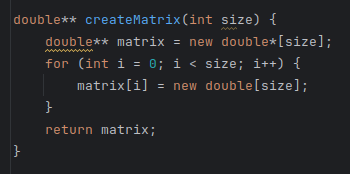


Рисунок 1.1 – Функция создания матрицы

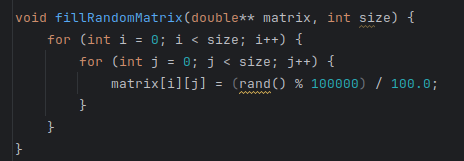


Рисунок 1.2 – Функция заполнения матрицы

Размер матрицы можно задать как аргумент командной строки и ввести после самого запуска (Рисунок 1.3).

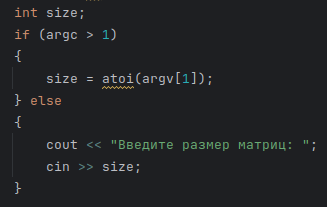


Рисунок 1.3 – Получение размера матриц

Перемножение матриц происходит базовым перебором каждой из ячеек, что представлено на рисунке 1.4.

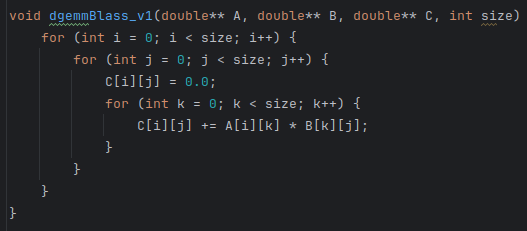


Рисунок 1.4 – Перемножение матриц

**Задание 2**

Проведем ряд испытаний, передав в качестве аргумента размеры матриц от 1000 до 10000 с шагом 1000 и получим время выполнения перемножения для каждого из чисел (Рисунок 2.1).

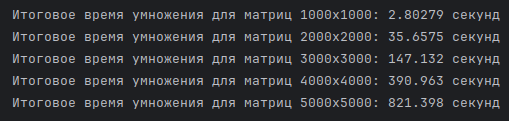


Рисунок 2.1 – Результат работы для различных входных данных

Как можем заметить, рост времени происходит настолько быстро, что матрицы 10000х10000 будут перемножаться так много времени, что ожидание может занять часы. Это обусловлено тем, что временная сложность выполнения 3-х вложенных циклов равна O(N3). Построим график зависимости времени перемножения от размера матриц на основе полученных данных и аппроксимируем его до значения 10000 (Рисунок 2.2).

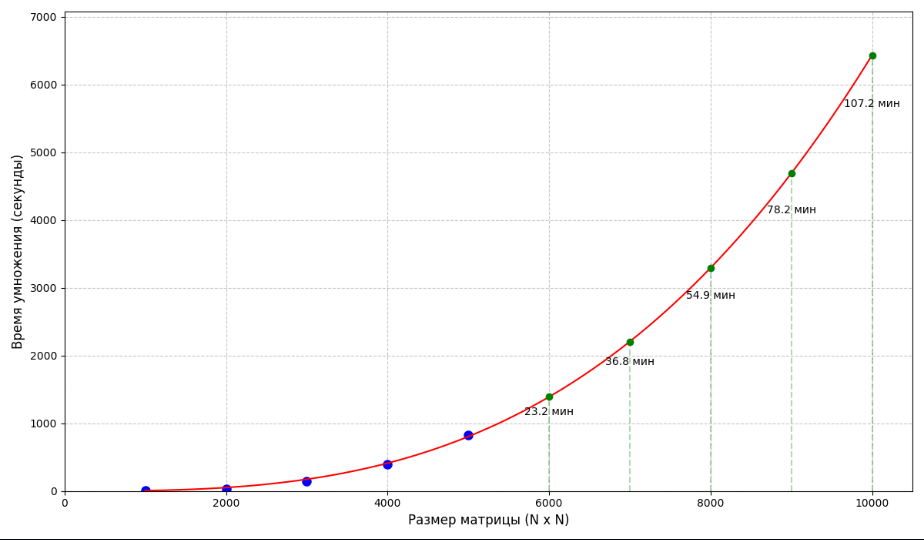


Рисунок 2.1 – Результат работы для различных входных данных

Анализируя график, можно сделать вывод, что рост времени происходит с кубической скоростью.

**Задание 3**

Для оценки предельного размера матриц, которые можно перемножить на моем вычислительном устройстве необходимо оценить размеры динамических массивов, которые можно создать на устройстве.

Общий объем оперативной памяти на устройстве – 32 Гб, под программы доступно примерно 30 Гб. Необходимый объем памяти для перемножения двух матриц вычисляется по формуле: 3 \* N2 \* 8 байт (размер переменной типа double). Подставив известный максимальный допустимый объем занятой памяти, получим N = 36600. Таким образом мы получили теоретическое значение максимального размера перемножаемых матриц.

**Задание 4**

Реализуем функцию, в которой будет выполняться оптимизация доступа к памяти, за счет построчного перебора элементов обеих матриц (Рисунок 4.1).

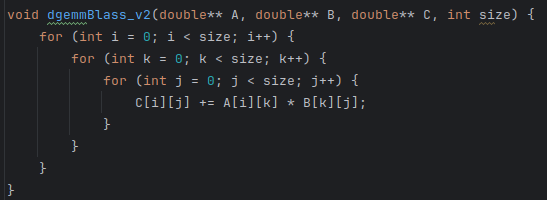


Рисунок 4.1 – Перемножение построчным перебором

Запустив программу, получим результат, представленный на рисунке 4.2.

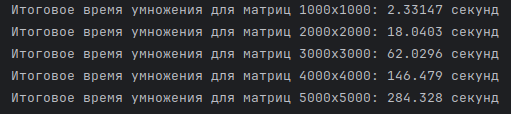


Рисунок 4.2 – Результат работы построчного перебора

Как можно заметить, результаты стали значительно лучше в сравнении с перемножением без оптимизации.

**Задание 5**

Реализуем функцию, в которой будет выполняться оптимизация доступа к памяти, за счет блочного перебора элементов обеих матриц (Рисунок 5.1).

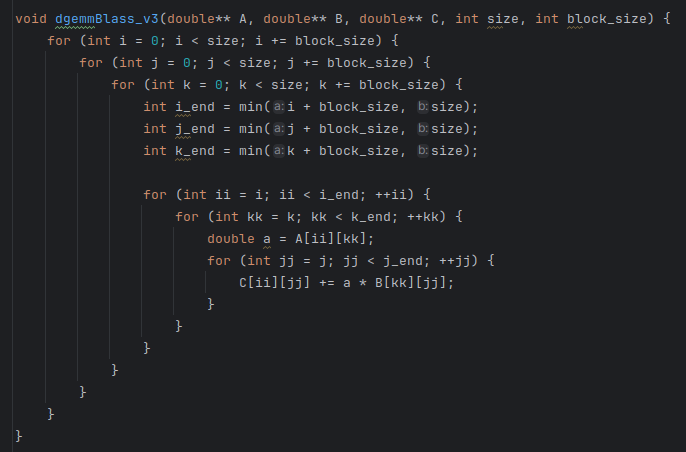


Рисунок 5.1 – Перемножение блочным перебором

Запустив программу и указав размер блока 20, получим результат, представленный на рисунке 5.2.

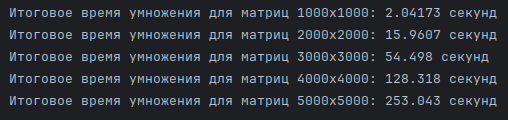


Рисунок 5.2 – Результат работы блочного перебора

Проанализировав полученные результаты, видим, что блочный способ перебора наиболее эффективный.

**Задание 6**

Построим графики зависимости времени перемножения от размера матриц для всех 3-х способов, рассмотренных ранее (Рисунок 6.1).

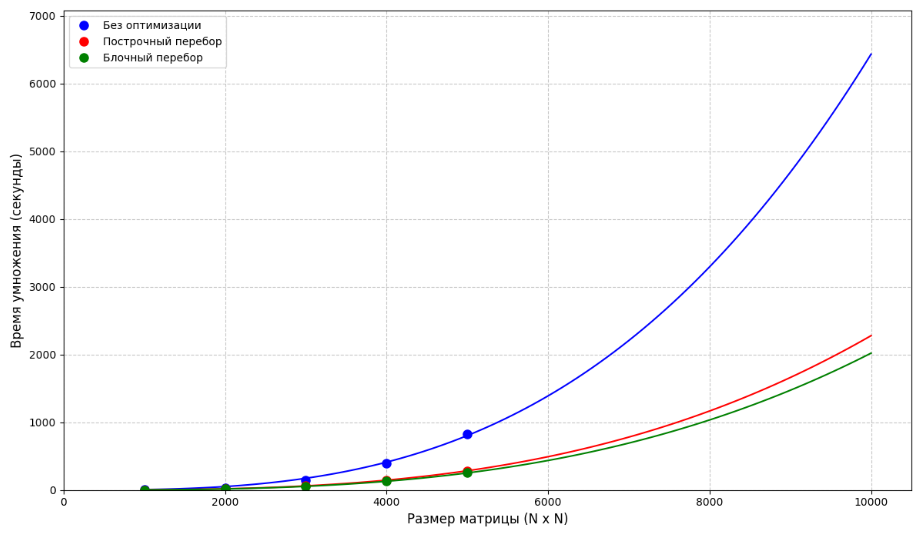


Рисунок 6.1 – Наглядное сравнение методов оптимизации

Далее, для блочного перебора определим размер блока, при котором достигается максимальное ускорение. Найдем размер такого блока для матриц размером 2048х2048 и в качестве проверяемых размеров будем использовать: 16, 32, 64, 128 и 256.

Для выполнения задания воспользуемся функцией testOptimalBlockSize (Рисунок 6.2).



Рисунок 6.2 – Функция для нахождения наиболее оптимального размера блока

Результат работы функции представлен на рисунке 6.3.

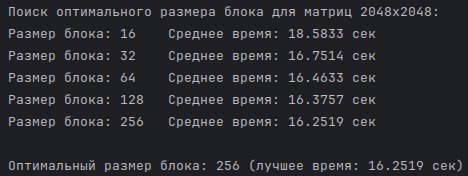


Рисунок 6.3 – Результат поиска наиболее оптимального размера блока

В рамках рассмотренных размеров матрицы и блоков делаем вывод, что наиболее оптимальный размер блока – 256.

**Задание 7**

С помощью профилировщика для каждого способа оптимизации доступа к памяти оценим количество промахов при работе к КЭШ памятью. Поскольку на нашей виртуальной машине не включена поддержка аппаратных событий, то вместо утилиты perf, воспользуемся утилитой valgrind. Так как утилита valgrind эмулирует процесс выполнения программы для сохранения производительности, то время работы значительно увеличивается (в десятки и сотни раз), поэтому оценим количество промахов при работе с КЭШ памятью на примере матрицы 100х100. Результаты проверки представлены на рисунках 7.1 – 7.3.

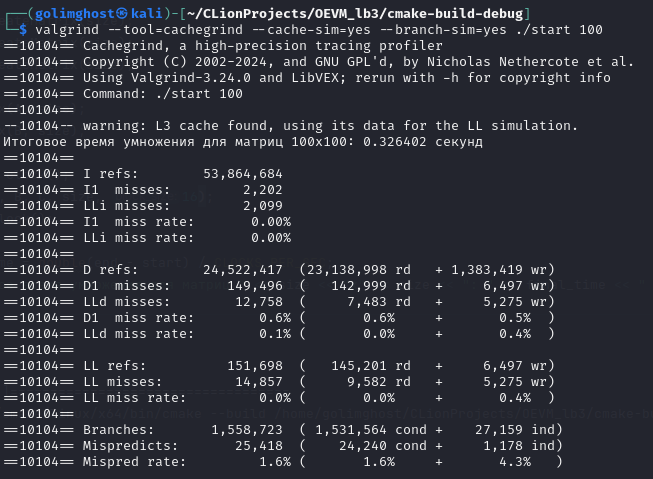


Рисунок 7.1 – Количество промахов при работе с КЭШ памятью без оптимизации

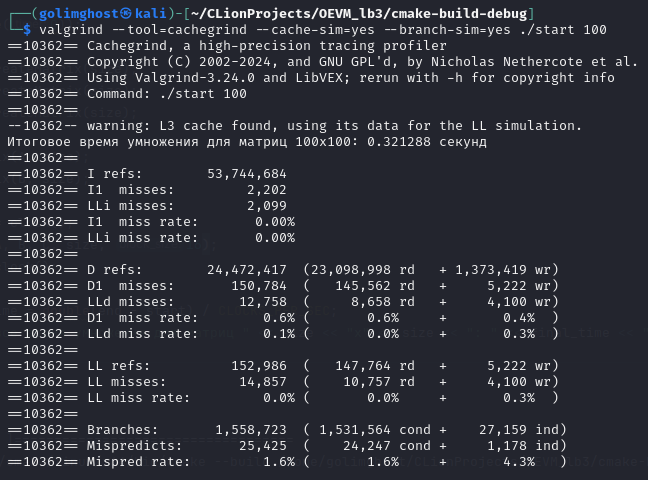


Рисунок 7.2 – Количество промахов при работе с КЭШ памятью с построчным перебором

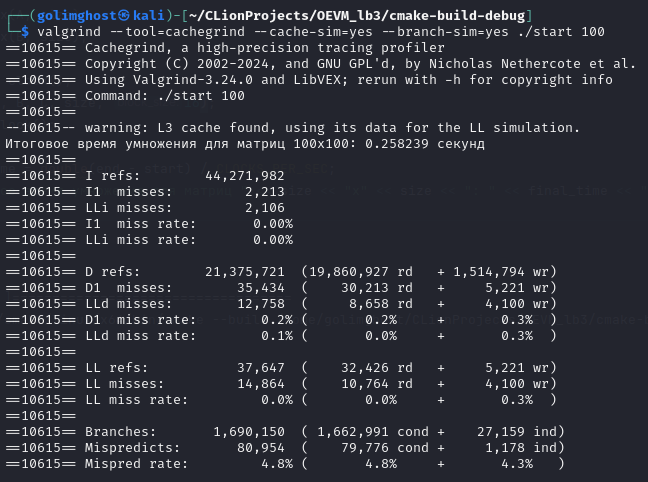


Рисунок 7.3 – Количество промахов при работе с КЭШ памятью с блочным перебором

Сравнив результаты трёх тестов, можно сделать вывод, что третий способ (с результатом 0.258 сек) оказался наиболее эффективным, так как показал наименьшее время выполнения (на ~20% быстрее первых двух вариантов) и снижение количества промахов кэша данных (D1 misses) в 4 раза (с 0.6% до 0.2%). При этом промахи последнего уровня кэша (LLd) остались на прежнем низком уровне (~0.1%), что указывает на улучшенную локальность данных. Однако третий вариант увеличил количество ошибочных предсказаний ветвлений (mispred rate) до 4.8%, что может быть связано с более сложной логикой циклов. В целом, оптимизация доступа к памяти дала больший выигрыш, чем потери на branch mispredictions, поэтому третий подход можно считать оптимальным.

**Вывод**

В ходе выполнения данной лабораторной работы было реализовано последовательное перемножение двух матриц, а также оптимизация взаимодействия с памятью, за счет построчного и блочного переборов.

Тестирование с рядом входных данных показало, что наиболее оптимальным вариантов исполнения будет блочный перебор при перемножении. По результатам тестирования были построены графики зависимости времени вычислений от размера матриц, что позволило наглядно показать эффективность предложенных методов оптимизации.

Также был вычислен экспериментально наиболее оптимальный размер блока, который потребовал наименьшее количество времени.

Помимо вышеперечисленного, была произведена оценка количества промахов при работе с КЭШ памятью, что также убедило нас в наибольшей оптимальности использования способа блочного перебора.

**Приложение А**

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <cstring>

#include <climits>

using namespace std;

double\*\* createMatrix(int size) {

double\*\* matrix = new double\*[size];

for (int i = 0; i < size; i++) {

matrix[i] = new double[size];

}

return matrix;

}

void freeMatrix(double\*\* matrix, int size) {

for (int i = 0; i < size; i++) {

delete[] matrix[i];

}

delete[] matrix;

}

void dgemmBlass\_v1(double\*\* A, double\*\* B, double\*\* C, int size) {

for (int i = 0; i < size; i++) {

for (int j = 0; j < size; j++) {

C[i][j] = 0.0;

for (int k = 0; k < size; k++) {

C[i][j] += A[i][k] \* B[k][j];

}

}

}

}

void dgemmBlass\_v2(double\*\* A, double\*\* B, double\*\* C, int size) {

for (int i = 0; i < size; i++) {

for (int k = 0; k < size; k++) {

for (int j = 0; j < size; j++) {

C[i][j] += A[i][k] \* B[k][j];

}

}

}

}

void dgemmBlass\_v3(double\*\* A, double\*\* B, double\*\* C, int size, int block\_size) {

for (int i = 0; i < size; i += block\_size) {

for (int j = 0; j < size; j += block\_size) {

for (int k = 0; k < size; k += block\_size) {

int i\_end = min(i + block\_size, size);

int j\_end = min(j + block\_size, size);

int k\_end = min(k + block\_size, size);

for (int ii = i; ii < i\_end; ++ii) {

for (int kk = k; kk < k\_end; ++kk) {

double a = A[ii][kk];

for (int jj = j; jj < j\_end; ++jj) {

C[ii][jj] += a \* B[kk][jj];

}

}

}

}

}

}

}

void fillRandomMatrix(double\*\* matrix, int size) {

for (int i = 0; i < size; i++) {

for (int j = 0; j < size; j++) {

matrix[i][j] = (rand() % 100000) / 100.0;

}

}

}

void testOptimalBlockSize() {

const int test\_size = 2048;

const int trials = 3;

double\*\* A = createMatrix(test\_size);

double\*\* B = createMatrix(test\_size);

double\*\* C = createMatrix(test\_size);

fillRandomMatrix(A, test\_size);

fillRandomMatrix(B, test\_size);

cout << "Поиск оптимального размера блока для матриц " << test\_size << "x" << test\_size << ":\n";

int best\_block = 0;

double best\_time = INT\_MAX;

for (int block\_size = 16; block\_size <= 256; block\_size \*= 2) {

double total\_time = 0;

for (int t = 0; t < trials; t++) {

for (int i = 0; i < test\_size; i++) {

memset(C[i], 0, test\_size \* sizeof(double));

}

clock\_t start = clock();

dgemmBlass\_v3(A, B, C, test\_size, block\_size);

clock\_t end = clock();

total\_time += double(end - start) / CLOCKS\_PER\_SEC;

}

double avg\_time = total\_time / trials;

cout << "Размер блока: " << block\_size << "\tСреднее время: " << avg\_time << " сек" << endl;

if (avg\_time < best\_time) {

best\_time = avg\_time;

best\_block = block\_size;

}

}

cout << "\nОптимальный размер блока: " << best\_block << " (лучшее время: " << best\_time << " сек)" << endl;

freeMatrix(A, test\_size);

freeMatrix(B, test\_size);

freeMatrix(C, test\_size);

}

int main(int argc, char\* argv[]) {

int size;

if (argc > 1)

{

size = atoi(argv[1]);

srand(time(0));

double\*\* A = createMatrix(size);

double\*\* B = createMatrix(size);

double\*\* C = createMatrix(size);

fillRandomMatrix(A, size);

fillRandomMatrix(B, size);

clock\_t start = clock();

dgemmBlass\_v1(A, B, C, size);

clock\_t end = clock();

double final\_time = double(end - start) / CLOCKS\_PER\_SEC;

cout << "Итоговое время умножения для матриц " << size << "x" << size << ": " << final\_time << " секунд\n";

freeMatrix(A, size);

freeMatrix(B, size);

freeMatrix(C, size);

} else

{

cout << "Введите номер способа оптимизации: 1 - без оптимизации, 2 - построчный перебор, 3 - блочный перебор" << endl;

int chose;

cin >> chose;

if (chose == 1)

{

cout << "Время перемножения без оптимизации" << endl;

for (int i = 1000; i < 3000; i += 1000)

{

double\*\* A = createMatrix(i);

double\*\* B = createMatrix(i);

double\*\* C = createMatrix(i);

fillRandomMatrix(A, i);

fillRandomMatrix(B, i);

clock\_t start = clock();

dgemmBlass\_v1(A, B, C, i);

clock\_t end = clock();

double final\_time = double(end - start) / CLOCKS\_PER\_SEC;

cout << "Итоговое время умножения для матриц " << i << "x" << i << ": " << final\_time << " секунд\n";

freeMatrix(A, i);

freeMatrix(B, i);

freeMatrix(C, i);

}

}

else if (chose == 2)

{

cout << "Построчный перебор:" << endl;

for (int i = 1000; i < 3000; i += 1000)

{

double\*\* A = createMatrix(i);

double\*\* B = createMatrix(i);

double\*\* C = createMatrix(i);

fillRandomMatrix(A, i);

fillRandomMatrix(B, i);

clock\_t start = clock();

dgemmBlass\_v2(A, B, C, i);

clock\_t end = clock();

double final\_time = double(end - start) / CLOCKS\_PER\_SEC;

cout << "Итоговое время умножения для матриц " << i << "x" << i << ": " << final\_time << " секунд\n";

freeMatrix(A, i);

freeMatrix(B, i);

freeMatrix(C, i);

}

}

else

{

cout << "Введите размер блока: ";

int block\_size;

cin >> block\_size;

cout << "Блочный перебор:" << endl;

for (int i = 1000; i < 3000; i += 1000)

{

double\*\* A = createMatrix(i);

double\*\* B = createMatrix(i);

double\*\* C = createMatrix(i);

fillRandomMatrix(A, i);

fillRandomMatrix(B, i);

clock\_t start = clock();

dgemmBlass\_v3(A, B, C, i, block\_size);

clock\_t end = clock();

double final\_time = double(end - start) / CLOCKS\_PER\_SEC;

cout << "Итоговое время умножения для матриц " << i << "x" << i << ": " << final\_time << " секунд\n";

freeMatrix(A, i);

freeMatrix(B, i);

freeMatrix(C, i);

}

}

}

return 0;

}