###### МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

###### ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

###### НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

###### Факультет информационных технологий

**Кафедра параллельных вычислений**

**ОТЧЕТ**

**О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ**

«Векторизация вычислений»

студента *2* курса, группы 22204

**Мустакаева Дмитрия Дмитриевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Кандидат технических наук

Власенко Андрей Юрьевич

Новосибирск 2022

# СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 2](#_gjdgxs)

[ЦЕЛЬ 3](#_30j0zll)

[ЗАДАНИЕ 3](#_1fob9te)

[АЛГОРИТМ ИЗ ЗАДАНИЯ 3](#_3znysh7)

[ХОД РАБОТЫ 4](#_2et92p0)

[ЗАКЛЮЧЕНИЕ 5](#_tyjcwt)

[Приложение 1. Программа на C++ с реализацией алгоритма без применения векторизации вычислений 6](#_1t3h5sf)

[Приложение 2. Программа на C++ с реализацией алгоритма с применением ручной векторизации с помощью ассемблерной вставки и встроенных SIMD-функций компилятора](#_4d34og8) 11

[Приложение 3. Программа на C++ с реализацией алгоритма при помощи библиотеки BLAS 1](#_17dp8vu)7

# ЦЕЛЬ

1. Изучение SIMD-расширений архитектуры x86/x86-64.
2. Изучение способов использования SIMD-расширений в программах на языке Си.
3. Получение навыков использования SIMD-расширений.

# ЗАДАНИЕ

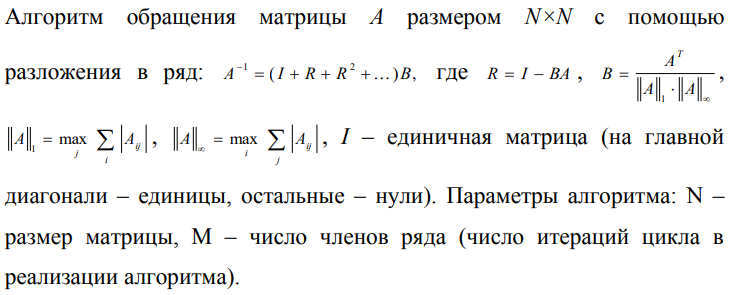
1. Написать три варианта программы, реализующей алгоритм из задания:

* вариант без ручной векторизации,
* вариант с ручной векторизацией (выбрать любой вариант из возможных трех: ассемблерная вставка, встроенные функции компилятора, расширение GCC),
* вариант с матричными операциями, выполненными с использованием оптимизированной библиотеки BLAS.

Для элементов матриц использовать тип данных float.

1. Проверить правильность работы программ на нескольких небольших тестовых наборах входных данных.
2. Каждый вариант программы оптимизировать по скорости, насколько это возможно.
3. Сравнить время работы трех вариантов программы для N=2048, M=10.

# АЛГОРИТМ ИЗ ЗАДАНИЯ



# ХОД РАБОТЫ

1. Была написана программа на языке C++ с реализацией алгоритма без применения векторизации (см. приложение).  
   
2. Затем была написана вторая версия программы, которая реализует тот же алгоритм, но с применением ручной векторизации встроенных функций компилятора. Ниже приведен снимок экрана с выводом времени работы программы.



1. Так же был написан третий вариант реализации алгоритма, с использованием библиотеки BLAS. Ниже приведен снимок экрана с выводом времени работы программы.



# ЗАКЛЮЧЕНИЕ

В ходе практической работы были изучены различные способы векторизации вычислений для оптимизации выполнения процессов на примере алгоритма по обращению матрицы.

Было написано три версии программы с различными способами использования векторизации вычислений. Ручная векторизация оказалась оказалась довольно эффективной, приблизившись к уровню библиотеки. Использование же библиотеки BLAS еще сильнее ускорило работу программы, тк в ней используется более высокооптимизированный код и параллельные вычисления.

# Приложение 1. Программа на C++ с реализацией алгоритма без применения векторизации вычислений

#include <cstdlib>

#include <iostream>

#include <math.h>

#include <string.h>

#include <time.h>

struct matrix {

float\* data;

size\_t size;

~matrix(){

delete[] data;

size = 0;

}

};

matrix\* create\_matrix(size\_t size) {

matrix\* mx = new matrix;

mx->data = new float[size \* size];

mx->size = size;

return mx;

}

void fill\_matrix(matrix\* mx) {

for(size\_t i = 0; i < mx->size; ++i){

for(size\_t j = 0; j < mx->size; ++j){

// mx->data[i \* mx->size + j] = static\_cast <float> (rand()) / (static\_cast <float> (RAND\_MAX/1000));

mx->data[i \* mx->size + j] = j + 1;

}

}

}

void print\_matrix(const matrix\* mx) {

std::cout << "\n";

for (size\_t i = 0; i < mx->size; ++i) {

for (size\_t j = 0; j < mx->size; ++j) {

std::cout << mx->data[i \* mx->size + j] << " ";

}

std::cout << "\n";

}

}

matrix\* identity\_matrix(size\_t size) {

matrix\* res = create\_matrix(size);

memset(res->data, 0, size \* size \* sizeof(float));

for (size\_t i = 0; i < size; ++i) {

res->data[i \* size + i] = 1;

}

return res;

}

matrix\* matrix\_transporition(const matrix\* mx) {

matrix\* res = create\_matrix(mx->size);

for (size\_t i = 0; i < mx->size; ++i) {

for (size\_t j = 0; j < mx->size; ++j) {

res->data[i \* mx->size + j] = mx->data[j \* mx->size + i];

}

}

return res;

}

matrix\* sub\_matrix(const matrix\* mx1, const matrix\* mx2) {

if (mx1->size != mx2->size)

throw std::runtime\_error("Different matrix sizes in sub\_matrix func\n");

matrix\* res = create\_matrix(mx1->size);

for (size\_t i = 0; i < mx1->size; ++i) {

for (size\_t j = 0; j < mx1->size; ++j) {

res->data[i \* mx1->size + j] = mx1->data[i \* mx1->size + j] - mx2->data[i \* mx2->size + j];

}

}

return res;

}

matrix\* sum\_matrix(const matrix\* mx1, const matrix\* mx2) {

if (mx1->size != mx2->size)

throw std::runtime\_error("Different matrix sizes in sub\_matrix func\n");

matrix\* res = create\_matrix(mx1->size);

for (size\_t i = 0; i < mx1->size; ++i) {

for (size\_t j = 0; j < mx1->size; ++j) {

res->data[i \* mx1->size + j] = mx1->data[i \* mx1->size + j] + mx2->data[i \* mx1->size + j];

}

}

return res;

}

matrix\* mul\_matrix(const matrix\* mx1, const matrix\* mx2) {

if (mx1->size != mx2->size)

throw std::runtime\_error("Different matrix sizes in sub\_matrix func\n");

matrix\* res = create\_matrix(mx1->size);

memset(res->data, 0, mx1->size \* mx1->size \* sizeof(float));

for (size\_t i = 0; i < mx1->size; i++) {

for (size\_t j = 0; j < mx1->size; j++) {

for (size\_t k = 0; k < mx1->size; k++) {

res->data[i \* mx1->size + k] += mx1->data[i \* mx1->size + j] \* mx2->data[j \* mx2->size + k];

}

}

}

return res;

}

void scalar\_mul\_matrix(const matrix\* mx, const float scalar) {

for (size\_t i = 0; i < mx->size; ++i) {

for (size\_t j = 0; j < mx->size; ++j) {

mx->data[i \* mx->size + j] \*= scalar;

}

}

}

unsigned int max\_col(const matrix\* mx) {

unsigned int max = 0;

for (size\_t j = 0; j < mx->size; ++j) {

unsigned int cur = 0;

for (size\_t i = 0; i < mx->size; ++i) {

cur += abs(mx->data[i + j \* mx->size]);

}

if (cur > max) {

max = cur;

}

}

return max;

}

unsigned int max\_row(const matrix\* mx) {

unsigned int max = 0;

for (size\_t i = 0; i < mx->size; ++i) {

unsigned int cur = 0;

for (size\_t j = 0; j < mx->size; ++j) {

cur += abs(mx->data[i \* mx->size + j]);

}

if (cur > max) {

max = cur;

}

}

return max;

}

matrix\* inverse\_matrix(const matrix\* mx, size\_t iterations) {

matrix\* res = identity\_matrix(mx->size);

matrix\* mx\_b = matrix\_transporition(mx);

scalar\_mul\_matrix(mx\_b, 1.0f / (max\_col(mx) \* max\_row(mx)));

matrix\* mx\_r = sub\_matrix(res, mul\_matrix(mx\_b, mx));

matrix\* mx\_r\_pow = sub\_matrix(res, mul\_matrix(mx\_b, mx));

for (size\_t i = 0; i < iterations; ++i) {

res = sum\_matrix(res, mx\_r\_pow);

mx\_r\_pow = mul\_matrix(mx\_r\_pow, mx\_r);

}

res = mul\_matrix(res, mx\_b);

return res;

}

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

if (argc != 3) {

throw std::runtime\_error("Wrong number of args.\nUsage: $./task4 N M\n");

}

srand(time(0));

size\_t N = atoll(argv[1]);

size\_t M = atoll(argv[2]);

matrix\* mx = create\_matrix(N);

fill\_matrix(mx);

//print\_matrix(mx);

struct timespec start, end;

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);

matrix\* res = inverse\_matrix(mx, M);

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);

std::cout << "Time taken: " << end.tv\_sec - start.tv\_sec + 0.000000001 \* (end.tv\_nsec - start.tv\_nsec) << " secs."

<< std::endl;

//print\_matrix(res);

return 0;

}

# Приложение 2. Программа на C++ с реализацией алгоритма с применением ручной векторизации с помощью встроенных SIMD-функций компилятора

#include <cstdlib>

#include <iostream>

#include <math.h>

#include <string.h>

#include <time.h>

#include <emmintrin.h>

#include <immintrin.h>

#include <iostream>

#include <unistd.h>

struct matrix {

float\* data;

size\_t size;

~matrix(){

delete[] data;

size = 0;

}

};

matrix\* create\_matrix(size\_t size) {

matrix\* mx = new matrix;

mx->data = new float[size \* size];

mx->size = size;

return mx;

}

void fill\_matrix(matrix\* mx) {

for(size\_t i = 0; i < mx->size; ++i){

for(size\_t j = 0; j < mx->size; ++j){

mx->data[i \* mx->size + j] = rand() % 3;

}

}

}

void print\_matrix(const matrix\* mx) {

std::cout << "\n";

for (size\_t i = 0; i < mx->size; ++i) {

for (size\_t j = 0; j < mx->size; ++j) {

std::cout << mx->data[i \* mx->size + j] << " ";

}

std::cout << "\n";

}

}

matrix\* identity\_matrix(size\_t size) {

matrix\* res = create\_matrix(size);

memset(res->data, 0, size \* size \* sizeof(float));

for (size\_t i = 0; i < size; ++i) {

res->data[i \* size + i] = 1;

}

return res;

}

matrix\* matrix\_transposition(const matrix\* mx) {

matrix\* res = create\_matrix(mx->size);

for (size\_t i = 0; i < mx->size; ++i) {

for (size\_t j = 0; j < mx->size; j += 4) {

\_\_m128 data = \_mm\_loadu\_ps(&mx->data[j \* mx->size + i]);

\_mm\_storeu\_ps(&res->data[i \* mx->size + j], data);

}

}

return res;

}

matrix\* sub\_matrix(const matrix\* mx1, const matrix\* mx2) {

if (mx1->size != mx2->size)

throw std::runtime\_error("Different matrix sizes in sub\_matrix func\n");

matrix\* res = create\_matrix(mx1->size);

for (size\_t i = 0; i < mx1->size; ++i) {

for (size\_t j = 0; j < mx1->size; j += 4) {

\_\_m128 a = \_mm\_loadu\_ps(&mx1->data[i \* mx1->size + j]);

\_\_m128 b = \_mm\_loadu\_ps(&mx2->data[i \* mx1->size + j]);

\_\_m128 result = \_mm\_sub\_ps(a, b);

\_mm\_storeu\_ps(&res->data[i \* mx1->size + j], result);

}

}

return res;

}

matrix\* sum\_matrix(const matrix\* mx1, const matrix\* mx2) {

if (mx1->size != mx2->size)

throw std::runtime\_error("Different matrix sizes in sum\_matrix func\n");

matrix\* res = create\_matrix(mx1->size);

for (size\_t i = 0; i < mx1->size; ++i) {

for (size\_t j = 0; j < mx1->size; j += 4) {

\_\_m128 a = \_mm\_loadu\_ps(&mx1->data[i \* mx1->size + j]);

\_\_m128 b = \_mm\_loadu\_ps(&mx2->data[i \* mx1->size + j]);

\_\_m128 result = \_mm\_add\_ps(a, b);

\_mm\_storeu\_ps(&res->data[i \* mx1->size + j], result);

}

}

return res;

}

matrix\* transpose(const matrix\* mx) {

auto \*m\_a = (\_\_m128 \*) mx->data;

auto tmp\_m = m\_a;

matrix\* res = create\_matrix(mx->size);

for (size\_t i = 0; i < mx->size; ++i) {

for (int j = 0; j < mx->size / 4; ++j) {

\_\_m128 tmp = \_mm\_load\_ps((float \*) tmp\_m);

float four[4];

\_mm\_store\_ps(four, tmp);

res->data[(j \* 4 + 0) \* mx->size + i] = four[0];

res->data[(j \* 4 + 1) \* mx->size + i] = four[1];

res->data[(j \* 4 + 2) \* mx->size + i] = four[2];

res->data[(j \* 4 + 3) \* mx->size + i] = four[3];

tmp\_m++;

}

}

return res;

}

matrix\* mul\_matrix(const matrix\* mx1, const matrix\* mx2) {

matrix\* mx2\_tr = transpose(mx2);

matrix\* res = create\_matrix(mx1->size);

for (size\_t i = 0; i < mx1->size; ++i) {

for (int j = 0; j < mx1->size; ++j) {

\_\_m128 m\_sum = \_mm\_setzero\_ps();

for (int r = 0; r < mx1->size / 4; r++) {

\_\_m128 temp1 = \_mm\_load\_ps(&mx1->data[i \* mx1->size + r \* 4]);

\_\_m128 temp2 = \_mm\_load\_ps(&mx2\_tr->data[j \* mx1->size + r \* 4]);

m\_sum = \_mm\_add\_ps(m\_sum, \_mm\_mul\_ps(temp1, temp2));

}

float four[4];

\_mm\_store\_ps(four, m\_sum);

float sum = four[0] + four[1] + four[2] + four[3];

res->data[i \* mx1->size + j] = sum;

}

}

return res;

}

void scalar\_mul\_matrix(matrix\* mx, const float scalar) {

for (size\_t i = 0; i < mx->size; ++i) {

for (size\_t j = 0; j < mx->size; j += 4) {

\_\_m128 data = \_mm\_loadu\_ps(&mx->data[i \* mx->size + j]);

\_\_m128 result = \_mm\_mul\_ps(data, \_mm\_set1\_ps(scalar));

\_mm\_storeu\_ps(&mx->data[i \* mx->size + j], result);

}

}

}

unsigned int max\_col(const matrix\* mx) {

unsigned int max = 0;

for (size\_t j = 0; j < mx->size; ++j) {

unsigned int cur = 0;

for (size\_t i = 0; i < mx->size; ++i) {

cur += abs(mx->data[i + j \* mx->size]);

}

if (cur > max) {

max = cur;

}

}

return max;

}

unsigned int max\_row(const matrix\* mx) {

unsigned int max = 0;

for (size\_t i = 0; i < mx->size; ++i) {

unsigned int cur = 0;

for (size\_t j = 0; j < mx->size; ++j) {

cur += abs(mx->data[i \* mx->size + j]);

}

if (cur > max) {

max = cur;

}

}

return max;

}

matrix\* inverse\_matrix(const matrix\* mx, size\_t iterations) {

matrix\* res = identity\_matrix(mx->size);

matrix\* mx\_b = matrix\_transposition(mx);

scalar\_mul\_matrix(mx\_b, 1.0f / (max\_col(mx) \* max\_row(mx)));

matrix\* mx\_r = sub\_matrix(res, mul\_matrix(mx\_b, mx));

matrix\* mx\_r\_pow = sub\_matrix(res, mul\_matrix(mx\_b, mx));

for (size\_t i = 0; i < iterations; ++i) {

res = sum\_matrix(res, mx\_r\_pow);

mx\_r\_pow = mul\_matrix(mx\_r\_pow, mx\_r);

}

res = mul\_matrix(res, mx\_b);

return res;

}

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

if (argc != 3) {

throw std::runtime\_error("Wrong number of args.\nUsage: $./task4 N M\n");

}

srand(time(0));

size\_t N = atoll(argv[1]);

if(N % 4 != 0) {

throw std::runtime\_error("N should be % 4 == 0");

}

size\_t M = atoll(argv[2]);

matrix\* mx = create\_matrix(N);

fill\_matrix(mx);

//print\_matrix(mx);

struct timespec start, end;

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);

matrix\* res = inverse\_matrix(mx, M);

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);

std::cout << "Time taken: " << end.tv\_sec - start.tv\_sec + 0.000000001 \* (end.tv\_nsec - start.tv\_nsec) << " secs."

<< std::endl;

//print\_matrix(res);

return 0;

}

# Приложение 3. Программа на C++ с реализацией алгоритма при помощи библиотеки BLAS

#include <cstdlib>

#include <iostream>

#include <cmath>

#include <ctime>

#include <cblas.h>

struct matrix {

float\* data;

size\_t size;

~matrix(){

delete[] data;

size = 0;

}

};

matrix\* create\_matrix(size\_t size) {

matrix\* mx = new matrix;

mx->data = new float[size \* size];

mx->size = size;

return mx;

}

void fill\_matrix(matrix\* mx) {

for (size\_t i = 0; i < mx->size; ++i) {

for (size\_t j = 0; j < mx->size; ++j) {

mx->data[i \* mx->size + j] = j;

}

}

}

void print\_matrix(const matrix\* mx) {

std::cout << "\n";

for (size\_t i = 0; i < mx->size; ++i) {

for (size\_t j = 0; j < mx->size; ++j) {

std::cout << mx->data[i \* mx->size + j] << " ";

}

std::cout << "\n";

}

}

matrix\* identity\_matrix(size\_t size) {

matrix\* res = create\_matrix(size);

std::fill(res->data, res->data + size \* size, 0.0f);

for (size\_t i = 0; i < size; ++i) {

res->data[i \* size + i] = 1.0f;

}

return res;

}

matrix\* matrix\_transposition(const matrix\* mx) {

matrix\* res = create\_matrix(mx->size);

for (size\_t i = 0; i < mx->size; ++i) {

for (size\_t j = 0; j < mx->size; ++j) {

res->data[i \* mx->size + j] = mx->data[j \* mx->size + i];

}

}

return res;

}

matrix\* sub\_matrix(const matrix\* mx1, const matrix\* mx2) {

if (mx1->size != mx2->size)

throw std::runtime\_error("Different matrix sizes in sub\_matrix func\n");

matrix\* res = create\_matrix(mx1->size);

cblas\_scopy(mx1->size \* mx1->size, mx1->data, 1, res->data, 1);

cblas\_saxpy(mx1->size \* mx1->size, -1.0f, mx2->data, 1, res->data, 1); //ax + y

return res;

}

matrix\* sum\_matrix(const matrix\* mx1, const matrix\* mx2) {

if (mx1->size != mx2->size)

throw std::runtime\_error("Different matrix sizes in sum\_matrix func\n");

matrix\* res = create\_matrix(mx1->size);

cblas\_scopy(mx1->size \* mx1->size, mx1->data, 1, res->data, 1);

cblas\_saxpy(mx1->size \* mx1->size, 1.0f, mx2->data, 1, res->data, 1);

return res;

}

matrix\* mul\_matrix(const matrix\* mx1, const matrix\* mx2) {

if (mx1->size != mx2->size)

throw std::runtime\_error("Different matrix sizes in mul\_matrix func\n");

matrix\* res = create\_matrix(mx1->size);

cblas\_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, mx1->size, mx1->size, mx1->size, 1.0f, mx1->data, mx1->size,

mx2->data, mx2->size, 0.0f, res->data, mx1->size);

return res;

}

void scalar\_mul\_matrix(matrix\* mx, const float scalar) {

cblas\_sscal(mx->size \* mx->size, scalar, mx->data, 1);

}

unsigned int max\_col(const matrix\* mx) {

unsigned int max = 0;

for (size\_t j = 0; j < mx->size; ++j) {

unsigned int cur = 0;

for (size\_t i = 0; i < mx->size; ++i) {

cur += std::abs(mx->data[i + j \* mx->size]);

}

if (cur > max) {

max = cur;

}

}

return max;

}

unsigned int max\_row(const matrix\* mx) {

unsigned int max = 0;

for (size\_t i = 0; i < mx->size; ++i) {

unsigned int cur = 0;

for (size\_t j = 0; j < mx->size; ++j) {

cur += std::abs(mx->data[i \* mx->size + j]);

}

if (cur > max) {

max = cur;

}

}

return max;

}

matrix\* inverse\_matrix(const matrix\* mx, size\_t iterations) {

matrix\* res = identity\_matrix(mx->size);

matrix\* mx\_b = matrix\_transposition(mx);

scalar\_mul\_matrix(mx\_b, 1.0f / (max\_col(mx) \* max\_row(mx)));

matrix\* mx\_r = sub\_matrix(res, mul\_matrix(mx\_b, mx));

matrix\* mx\_r\_pow = sub\_matrix(res, mul\_matrix(mx\_b, mx));

for (size\_t i = 0; i < iterations; ++i) {

res = sum\_matrix(res, mx\_r\_pow);

mx\_r\_pow = mul\_matrix(mx\_r\_pow, mx\_r);

}

res = mul\_matrix(res, mx\_b);

return res;

}

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

if (argc != 3) {

throw std::runtime\_error("Wrong number of args.\nUsage: $./task4 N M\n");

}

srand(time(0));

size\_t N = atoll(argv[1]);

size\_t M = atoll(argv[2]);

matrix\* mx = create\_matrix(N);

fill\_matrix(mx);

//print\_matrix(mx);

struct timespec start, end;

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);

matrix\* res = inverse\_matrix(mx, M);

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);

std::cout << "Time taken: " << end.tv\_sec - start.tv\_sec + 0.000000001 \* (end.tv\_nsec - start.tv\_nsec) << " secs."

<< std::endl;

//print\_matrix(res);

return 0;

}