**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 ЛАБОРАТОРНОЙ РАБОТЫ**

«ВЛИЯНИЕ КЭШ-ПАМЯТИ НА ВРЕМЯ ОБРАБОТКИ МАССИВОВ»

студента 2 курса, 21203 группы

Неретина Степана Ивановича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.Ю. Власенко

Новосибирск 2020

**СОДЕРЖАНИЕ**

ЦЕЛЬ…………………………………………………………………...……3

ЗАДАНИЕ……………………………………………………………………4

ОПИСАНИЕ РАБОТЫ……………...………..…………………………..…5

ЗАКЛЮЧЕНИЕ……………...…………………..………………………..…6

Приложение 1. Без векторизации…………...………………………….…..7

Приложение 2. SIMD …………………...………………………….…11

Приложение 3. BLAS ……...…………...………………………….….17

**ЦЕЛЬ**

Изучить принцип работы работа кэш-память процессора и на примере обхода элементов массива в разном порядке выяснить размер разных уровней кэш-памяти на конкретном ПК.

**ЗАДАНИЕ**

На основе разных способов обхода циклического массива: прямого, обратного и случайного, сделать среднюю оценку количества тактов для обращения к одному элементу массива. На основе полученных данных выяснить приблизительный размер кэша процессора.

**График роста тактов обращения**

asdasd

**ЗАКЛЮЧЕНИЕ**

В ходе работы были изучены SIMD-расширения архитектуры x86/x86-64 и их использование в коде. Получены навыки написания ассемблерных вставок в языке С++, а также изучена работу оптимизированной библиотеки линейной алгебры BLAS.  
Как было выяснено, лучше всего пользоваться готовой библиотекой, хотя ручная векторизация и может дать ощутимый прирост в производительности. Единственный недостаток библиотеки BLAS составляет большое количество разных параметров, из-за которых код становится трудно понимаемым.

**Приложение 1**

#include <iostream>

#include <random>

#include <cfloat>

#include <string.h>

using namespace std;

const inline size\_t M = 10;

const inline size\_t N = 2048;

void CopyMatrix(const float\* from, float\* to) {

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

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

to[i \* N + j] = from[i \* N + j];

}

}

}

void SumMatrix(const float \*M1, const float \*M2, float \*result) {

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

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

result[i \* N + j] = M1[i \* N + j] + M2[i \* N + j];

}

}

}

void showMatrix(float \*matrix) {

size\_t i, j;

for (i = 0; i < N; ++i) {

for (j = 0; j < N; ++j) {

cout << matrix[i \* N + j] << " ";

}

cout << endl;

}

cout << "\n\n\n";

}

inline float generateRandomInRange(float a, float b) {

random\_device rd;

uniform\_real\_distribution gen(a, b);

return gen(rd);

}

float\* GenerateMatrixI() {

auto \*I = new float[N \* N];

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

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

if (i == j) I[i \* N + j] = 1.0;

else I[i \* N + j] = 0.0;

}

}

return I;

}

float \*GenerateMatrix() {

auto \*A = new float[N \* N];

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

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

A[i \* N + j] = generateRandomInRange(1, 100000);

}

}

return A;

}

void TransposeMatrix(float \*B, const float \*A) {

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

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

B[i \* N + j] = A[j \* N + i];

}

}

float MaxLineCount(const float \*matrix) {

float maximum = FLT\_MIN;

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

float temp = 0;

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

temp += matrix[i \* N + j];

}

if (temp > maximum) {

maximum = temp;

}

}

return maximum;

}

float \*GenerateMatrixB(float \*A) {

auto \*B = new float[N \* N];

auto \*transposedA = new float[N \* N];

TransposeMatrix(transposedA, A);

float maxRowCount = MaxLineCount(A);

float maxColumnCount = MaxLineCount(transposedA);

float divider = 1 / (maxRowCount \* maxColumnCount);

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

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

B[i \* N + j] = transposedA[i \* N + j] \* divider;

}

}

delete[] transposedA;

return B;

}

float \*MultiplyMatrices(const float \*M1, const float \*M2) {

auto\* M3 = new float[N \* N];

memset(M3, 0, sizeof(float) \* N \* N);

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

float\* c = M3 + i \* N;

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

const float\* b = M2 + k \* N;

float a = M1[i \* N + k];

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

c[j] += a \* b[j];

}

}

}

return M3;

}

void SubtractMatrix(const float \*I, const float \*multed, float \*R) {

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

for (size\_t j = 0; j < N; j++)

R[i \* N + j] = I[i \* N + j] - multed[i \* N + j];

}

float \*GenerateMatrixR(const float \*A, float \*I, const float \*B) {

auto \*R = new float[N \* N];

auto \*multed = MultiplyMatrices(B, A);

// R - BA

SubtractMatrix(I, multed, R);

delete[] multed;

return R;

}

float \*GetInversedMatrix(float \*A) {

float \*I = GenerateMatrixI();

auto \*copyI = new float[N\*N];

CopyMatrix(I, copyI);

float \*B = GenerateMatrixB(A);

// std::cout << "B matrix" << std::endl;

// showMatrix(B);

float \*R = GenerateMatrixR(A, copyI, B);

// std::cout << "R matrix" << std::endl;

// showMatrix(R);

float \*Rn = R;

float \*t;

SumMatrix(I, Rn, I);

// std::cout << "I matrix" << std::endl;

// showMatrix(I);

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

t = MultiplyMatrices(Rn, R);

// std::cout << "T matrix " << i << " " << std::endl;

// showMatrix(t);

if (i != 0) delete[] Rn;

Rn = t;

SumMatrix(I, Rn, I);

// std::cout << "I matrix" << std::endl;

// showMatrix(I);

}

auto new\_I = MultiplyMatrices(I, B);

delete[] I;

delete[] B;

delete[] R;

delete[] t;

delete [] copyI;

return new\_I;

}

int main()

{

float\* A = GenerateMatrix();

float\* InversedA = GetInversedMatrix(A);

// std::cout << "Matrix A" << std::endl;

// showMatrix(A);

// std::cout << "Inversed Matrix" << std::endl;

// showMatrix(InversedA);

// std::cout << "A\*A^-1" << std::endl;

// showMatrix(MultiplyMatrices(A, InversedA));

delete[] A;

delete[] InversedA;

return 0;

}

**Приложение 2**

#include <iostream>

#include <xmmintrin.h>

#include <cstdlib>

#include <cfloat>

#include <random>

#include <string.h>

using namespace std;

const inline size\_t M = 10;

const inline size\_t N = 2048;

void CopyMatrix(const float\* from, float\* to) {

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

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

to[i \* N + j] = from[i \* N + j];

}

}

}

void SumMatrix(const float \*M1, const float \*M2, float \*result) {

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

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

result[i \* N + j] = M1[i \* N + j] + M2[i \* N + j];

}

}

}

void showMatrix(const float \*matrix) {

size\_t i, j;

for (i = 0; i < N; ++i) {

for (j = 0; j < N; ++j) {

cout << matrix[i \* N + j] << " ";

}

cout << endl;

}

cout << "\n\n\n";

}

inline float generateRandomInRange(float a, float b) {

random\_device rd;

uniform\_real\_distribution gen(a, b);

return gen(rd);

}

float \*GenerateMatrixI() {

auto \*I = new float[N \* N];

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

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

if (i == j) I[i \* N + j] = 1.0;

else I[i \* N + j] = 0.0;

}

}

return I;

}

float \*GenerateMatrix() {

auto \*A = new float[N \* N];

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

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

A[i \* N + j] = generateRandomInRange(1, 100);

}

}

return A;

}

void TransposeMatrix(float \*B, const float \*A) {

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

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

B[i \* N + j] = A[j \* N + i];

}

}

float MaxLineCount(const float \*matrix) {

float maximum = FLT\_MIN;

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

float temp = 0;

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

temp += matrix[i \* N + j];

}

if (temp > maximum) {

maximum = temp;

}

}

return maximum;

}

float \*GenerateMatrixB(float \*A) {

auto \*B = new float[N \* N];

auto \*transposedA = new float[N \* N];

TransposeMatrix(transposedA, A);

float maxRowCount = MaxLineCount(A);

float maxColumnCount = MaxLineCount(transposedA);

float divider = 1 / (maxRowCount \* maxColumnCount);

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

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

B[i \* N + j] = transposedA[i \* N + j] \* divider;

}

}

delete[] transposedA;

return B;

}

float\* MultiplyMatrices(const float\* matrix\_1, const float\* matrix\_2)

{

\_\_m128 vector\_2;//переменная для хранения значения второй матрицы

\_\_m128 result\_vector;//переменная для записи итоговой марицы

auto\* result\_matrix = (float\*)\_mm\_malloc(N \* N \* sizeof(float), 16);//выделение памяти с выравниванием

memset(result\_matrix, 0, N\*N\*sizeof(float));

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

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

\_\_m128 element\_vector = \_mm\_set1\_ps(matrix\_1[i \* N + j]);//4 позиции в одно значение

for (int k = 0; k < N; k += 4) {

vector\_2 = \_mm\_load\_ps(&matrix\_2[j \* N + k]);//4 значение по адресу

result\_vector = \_mm\_load\_ps(&result\_matrix[i \* N + k]);

result\_vector = \_mm\_add\_ps(result\_vector, \_mm\_mul\_ps(element\_vector, vector\_2));

\_mm\_store\_ps(&result\_matrix[i \* N + k], result\_vector);

}

}

}

return result\_matrix;

}

void SubtractMatrix(const float \*I, const float \*multed, float \*R) {

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

for (size\_t j = 0; j < N; j++)

R[i \* N + j] = I[i \* N + j] - multed[i \* N + j];

}

float \*GenerateMatrixR(const float \*A, float \*I, const float \*B) {

auto \*R = new float[N \* N];

auto \*multed = MultiplyMatrices(B, A);

// R - BA

SubtractMatrix(I, multed, R);

\_mm\_free(multed);

return R;

}

float \*GetInversedMatrix(float \*A) {

float \*I = GenerateMatrixI();

auto \*copyI = new float[N\*N];

CopyMatrix(I, copyI);

float \*B = GenerateMatrixB(A);

// std::cout << "B matrix" << std::endl;

// showMatrix(B);

float \*R = GenerateMatrixR(A, copyI, B);

// std::cout << "R matrix" << std::endl;

// showMatrix(R);

float \*Rn = R;

float \*t;

SumMatrix(I, Rn, I);

// std::cout << "I matrix" << std::endl;

// showMatrix(I);

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

t = MultiplyMatrices(Rn, R);

// std::cout << "T matrix " << i << " " << std::endl;

// showMatrix(t);

if (i != 0) delete[] Rn;

Rn = t;

SumMatrix(I, Rn, I);

// std::cout << "I matrix" << std::endl;

// showMatrix(I);

}

auto new\_I = MultiplyMatrices(I, B);

delete[] I;

delete[] B;

delete[] R;

delete[] t;

delete [] copyI;

return new\_I;

}

int main() {

float \*A = GenerateMatrix();

float \*InversedA = GetInversedMatrix(A);

// std::cout << "Matrix A" << std::endl;

// showMatrix(A);

// std::cout << "Inversed Matrix" << std::endl;

// showMatrix(InversedA);

// std::cout << "A\*A^-1" << std::endl;

// showMatrix(MultiplyMatrices(A,InversedA));

delete[] A;

delete[] InversedA;

return 0;

}

Приложение 3

#include <iostream>

#include <random>

#include <cfloat>

#include <iomanip>

#include "cblas.h"

using namespace std;

const inline size\_t M = 100000;

const inline size\_t N = 16;

void CopyMatrix(const float\* from, float\* to) {

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

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

to[i \* N + j] = from[i \* N + j];

}

}

}

//void SumMatrix(const float \*M1, const float \*M2, float \*result) {

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

// for (size\_t j = 0; j < N; j++) {

// result[i \* N + j] = M1[i \* N + j] + M2[i \* N + j];

// }

// }

//}

void showMatrix(const float \*matrix) {

size\_t i, j;

for (i = 0; i < N; ++i) {

for (j = 0; j < N; ++j) {

const auto\* x = &matrix[i \* N + j];

std::cout.setf(std::ios::fixed);

std::cout << std::setprecision(3);

std::cout << std::setw(6);

// if (\*x < 0.0001){

// std::cout << 0;

// }

// else{

// cout << \*x;

// }

std::cout << \*x;

std::cout << ' ';

}

cout << endl;

}

cout << "\n\n\n";

}

void SumMatrix(const float \*M1, float \*M2) {

cblas\_saxpy(N\*N, 1, M1, 1, M2, 1);

}

inline float generateRandomInRange(float a, float b) {

random\_device rd;

uniform\_real\_distribution gen(a, b);

return gen(rd);

}

float\* GenerateMatrixI() {

auto \*I = new float[N \* N];

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

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

if (i == j) I[i \* N + j] = 1.0;

else I[i \* N + j] = 0.0;

}

}

return I;

}

float \*GenerateMatrix() {

auto \*A = new float[N \* N];

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

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

A[i \* N + j] = generateRandomInRange(1, 100000);

}

}

return A;

}

void TransposeMatrix(float \*B, const float \*A) {

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

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

B[i \* N + j] = A[j \* N + i];

}

}

float MaxLineCount(const float \*matrix) {

float maximum = FLT\_MIN;

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

float temp = 0;

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

temp += matrix[i \* N + j];

}

if (temp > maximum) {

maximum = temp;

}

}

return maximum;

}

float \*GenerateMatrixB(float \*A) {

auto \*B = new float[N \* N];

auto \*transposedA = new float[N \* N];

TransposeMatrix(transposedA, A);

float maxRowCount = MaxLineCount(A);

float maxColumnCount = MaxLineCount(transposedA);

float divider = 1 / (maxRowCount \* maxColumnCount);

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

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

B[i \* N + j] = transposedA[i \* N + j] \* divider;

}

}

delete[] transposedA;

return B;

}

float \*MultiplyMatrices(const float \*M1, const float \*M2) {

auto\* M3 = new float[N \* N];

cblas\_sgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans, N,N,N,1.0,M1,N,M2,N,0.0,M3,N);

return M3;

}

void SubtractMatrix(const float \*I, const float \*multed, float \*R) {

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

for (size\_t j = 0; j < N; j++)

R[i \* N + j] = I[i \* N + j] - multed[i \* N + j];

}

float \*GenerateMatrixR(const float \*A, float \*I, const float \*B) {

auto \*R = new float[N \* N];

auto \*multed = MultiplyMatrices(B, A);

// R - BA

SubtractMatrix(I, multed, R);

delete[] multed;

return R;

}

float \*GetInversedMatrix(float \*A) {

float \*I = GenerateMatrixI();

auto \*copyI = new float[N\*N];

CopyMatrix(I, copyI);

float \*B = GenerateMatrixB(A);

float \*R = GenerateMatrixR(A, copyI, B);

float \*Rn = R;

float \*t;

SumMatrix(Rn, I);

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

t = MultiplyMatrices(Rn, R);

if (i != 0) delete[] Rn;

Rn = t;

SumMatrix(Rn, I);

}

auto new\_I = MultiplyMatrices(I, B);

delete[] I;

delete[] B;

delete[] R;

delete[] t;

delete [] copyI;

return new\_I;

}

int main() {

float \*A = GenerateMatrix();

float \*InversedA = GetInversedMatrix(A);

showMatrix(MultiplyMatrices(A,InversedA));

delete[] A;

delete[] InversedA;

return 0;

}