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 22206

***Тропина Никиты Васильевича***

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

доцент, к.т.н.

А.Ю.Власенко

Новосибирск 2023

# СОДЕРЖАНИЕ

[ЦЕЛИ 3](#_Toc152230027)

[ЗАДАНИЕ 3](#_Toc152230028)

[ОПИСАНИЕ РАБОТЫ 4](#_Toc152230029)

[ЗАКЛЮЧЕНИЕ 5](#_Toc152230030)

[ПриложениЕ 1. Реализация алгоритма без векторизации. 6](#_Toc152230031)

[Приложение 2. Реализация алгоритма с помощью собственной векторизации. 8](#_Toc152230032)

[Приложение 3. Реализация программы с использованием библиотеки BLAS. 11](#_Toc152230033)

# ЦЕЛИ

1. Ознакомиться с методами векторизации вычислений.
2. Векторизовать вычисления самостоятельно и сравнить результат с использованием библиотеки BLAS.

# ЗАДАНИЕ

1. Написать три варианта программы, реализующей алгоритм обращения матрицы A размером с помощью разложения в ряд:  
   , где , , , I – единичная матрица. Параметры алгоритма N – размер матрицы, M – число членов ряда.

* Вариант без векторизации.
* Вариант с ручной векторизацией (выбрать любой вариант из возможных трех: ассемблерная вставка, встроенные функции компилятора, расширение GCC). При этом векторизованы должны быть как минимум операции перемножения двух матриц, сложения матриц, умножения матрицы на скаляр.
* Вариант с матричными операциями, выполненными с использованием оптимизированной библиотеки BLAS. Для элементов матриц использовать тип данных float. Использовать минимум 2 BLAS-функции.

1. Проверить правильность выполнения программы 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.
4. Составить отчет по лабораторной работе.

# ОПИСАНИЕ РАБОТЫ

Для заданного алгоритма было написано 3 варианта реализации: без векторизации (приложение 1), с ручной векторизацией с помощью встроенных функций компилятора (приложение 2) и с использованием оптимизированной библиотеки BLAS (приложение 3).

Все реализации были проверены на правильность и оптимизированы.

Сравнение скорости выполнения алгоритма производилось при одинаковых наборах входных данных: размер квадратной матрицы – 2048 × 2048 и вычислялось 10 членов ряда.

|  |  |  |
| --- | --- | --- |
| **Вариант реализации** | **Время выполнения на windows** | **Время выполнения на unix-системе** |
| Без векторизации | 270c | 268с |
| Собственная векторизация | 56c | 55с |
| С библиотекой BLAS | 84c | 24с |

Без векторизации:



Собственная векторизация:



Библиотека BLAS:



Как видно из результатов, собственная векторизация с применением расширения AVX оказала существенное влияние на скорость выполнения и ускорила вычисления в 4.8 раз. А программа с библиотекой BLAS разительно зависит от системы, на которой производились измерения. Так, на windows в 1.5 раза медленнее собственной векторизации, но на unix-системе в 2.8 раз быстрее ручной векторизации и в 11 раз быстрее программы без векторизации.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения практической работы были написаны 3 реализации алгоритма вычисления обратной матрицы: без векторизации (приложение 1), с ручной векторизацией с помощью встроенных функций компилятора (приложение 2) и с использованием оптимизированной библиотеки BLAS (приложение 3). Было измерено время выполнения каждого варианта на одинаковых входных данных. Размер матрицы 2048 × 2048 и количество членов ряда 10. Полученные результаты значительно различаются. Так, вариант без векторизации выполнил вычисления за 270 секунд, а с собственной векторизацией – 56 секунд, что на в 4.8 раз быстрее. При этом реализация с BLAS справилась за 84 секунды под windows и за 24.5 под unix-системой, что в 3.2 и 11 раз быстрее первого вариант, но в 1.5 раза медленнее собственной векторизации на windows, но на unix-системе в 2.2 раза быстрее ручной векторизации.

# Приложение 1. Реализация алгоритма без векторизации.

#include <iostream>

#include <ctime>

float\* multMatrix(float a[], float b[], int size)

{

float\* result = new float[size \* size] {};

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

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

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

result[i \* size + j] += a[i \* size + k] \* b[k \* size + j];

}

}

}

return result;

}

float findMaxLine(float matrix[], int size) {

float currentSum = 0;

float maxLine = 0;

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

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

currentSum += matrix[i \* size + j];

}

if (currentSum > maxLine) {

maxLine = currentSum;

}

currentSum = 0;

}

return maxLine;

}

float findMaxColumn(float matrix[], int size) {

float currentSum = 0;

float maxColumn = 0;

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

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

currentSum += matrix[j \* size + i];

}

if (currentSum > maxColumn) {

maxColumn = currentSum;

}

currentSum = 0;

}

return maxColumn;

}

float\* getMatrixB(float matrix[], int size) {

float divider = findMaxLine(matrix, size) \* findMaxColumn(matrix, size);

float\* matrixB = new float[size \* size] {};

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

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

matrixB[j \* size + i] = matrix[i \* size + j] / divider;

}

}

return matrixB;

}

float\* getOneMatrix(int size) {

float\* matrix = new float[size \* size] {};

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

matrix[i \* size + i] = 1;

}

return matrix;

}

float\* getMatrixR(float matrixA[], float matrixB[], int size) {

float\* matrixR = getOneMatrix(size);

float\* matrixMult = multMatrix(matrixB, matrixA, size);

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

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

matrixR[i \* size + j] -= matrixMult[i \* size + j];

}

}

return matrixR;

}

float\* inverseMatrix(float matrix[], int size, int nMembers) {

float\* matrixB = getMatrixB(matrix, size);

float\* matrixResult = getOneMatrix(size);

float\* matrixR = getMatrixR(matrix, matrixB, size);

float\* matrixDegreeR = matrixR;

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

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

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

matrixResult[i \* size + j] += matrixDegreeR[i \* size + j];

}

}

if (k != nMembers - 1) {

matrixDegreeR = multMatrix(matrixDegreeR, matrixR, size);

}

}

matrixResult = multMatrix(matrixResult, matrixB, size);

delete[] matrixB;

delete[] matrixR;

delete[] matrixDegreeR;

return matrixResult;

}

int main()

{

const int n = 2048;

float\* matrix = new float[n \* n] {};

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

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

matrix[i \* n + j] = i + j;

}

}

double startTime = clock();

float\* c = inverseMatrix(matrix, n, 10);

double finishTime = clock();

std::cout << "Время выполнения: " << (finishTime - startTime) / CLOCKS\_PER\_SEC << "c" << std::endl;

delete[] matrix;

delete[] c;

}

# Приложение 2. Реализация алгоритма с помощью собственной векторизации.

#include <iostream>

#include <ctime>

#include <immintrin.h>

float\* getOneMatrix(int size) {

float\* matrix = new float[size \* size] {};

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

matrix[i \* size + i] = 1;

}

return matrix;

}

float horizontalSum256(\_\_m256 x) {

\_\_m256 tmp = \_mm256\_setzero\_ps();

tmp = \_mm256\_permute2f128\_ps(x, x, 1);

x = \_mm256\_hadd\_ps(x, tmp);

x = \_mm256\_hadd\_ps(x, x);

x = \_mm256\_hadd\_ps(x, x);

float sum = x.m256\_f32[0];

return sum;

}

float findHorizontalMax256(\_\_m256 x) {

float maxNumber = 0;

for (int i = 0; i < 8; i++) {

float curNumber = x.m256\_f32[i];

if (curNumber > maxNumber) {

maxNumber = curNumber;

}

}

return maxNumber;

}

float findMaxColumnMy(float matrix[], int size) {

\_\_m256\* matrixVector = (\_\_m256\*)matrix;

\_\_m256 currentSum = \_mm256\_setzero\_ps();

float verticalSum = 0;

float maxColumn = 0;

for (int i = 0; i < size / 8; i++) {

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

currentSum = \_mm256\_add\_ps(matrixVector[j \* size / 8 + i], currentSum);

}

verticalSum = findHorizontalMax256(currentSum);

if (verticalSum > maxColumn) {

maxColumn = verticalSum;

}

currentSum = \_mm256\_setzero\_ps();

}

return maxColumn;

}

float findMaxLineMy(float matrix[], int size) {

\_\_m256\* matrixVector = (\_\_m256\*)matrix;

\_\_m256 currentSum = \_mm256\_setzero\_ps();

float horizontalSum = 0;

float maxLine = 0;

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

for (int j = 0; j < size / 8; j++) {

currentSum = \_mm256\_add\_ps(matrixVector[i \* size / 8 + j], currentSum);

}

horizontalSum = horizontalSum256(currentSum);

if (maxLine < horizontalSum) {

maxLine = horizontalSum;

}

horizontalSum = 0;

currentSum = \_mm256\_setzero\_ps();

}

return maxLine;

}

void copyMatrix(float dst[], float src[], int size) {

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

dst[i] = src[i];

}

}

void addMatrixMy(float A[], float B[], float result[], int size) {

float\* resTmp = new float[size \* size] {};

\_\_m256\* m256ResTmp = (\_\_m256\*)resTmp;

\_\_m256\* m256A = (\_\_m256\*)A;

\_\_m256\* m256B = (\_\_m256\*)B;

for (int i = 0; i < size \* size / 8; i++) {

m256ResTmp[i] = \_mm256\_add\_ps(m256A[i], m256B[i]);

}

copyMatrix(result, resTmp, size);

delete[] resTmp;

}

void subMatrixMy(float A[], float B[], float result[], int size) {

float\* resTmp = new float[size \* size] {};

\_\_m256\* m256ResTmp = (\_\_m256\*)resTmp;

\_\_m256\* m256A = (\_\_m256\*)A;

\_\_m256\* m256B = (\_\_m256\*)B;

for (int i = 0; i < size \* size / 8; i++) {

m256ResTmp[i] = \_mm256\_sub\_ps(m256A[i], m256B[i]);

}

copyMatrix(result, resTmp, size);

delete[] resTmp;

}

void multMatrixMy(float a[], float b[], float result[], int size) {

\_\_m256\* m256Result = (\_\_m256\*)result;

const \_\_m256\* m256B = (const \_\_m256\*)b;

\_\_m256 m256A{};

\_\_m256 tmp;

float\* resTmp = new float[size \* size] {};

\_\_m256\* m256ResTmp = (\_\_m256\*)resTmp;

for (int i = 0; i < size \* size / 8; ++i)

m256ResTmp[i] = \_mm256\_setzero\_ps();

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

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

{

m256A = \_mm256\_set1\_ps(a[size \* i + j]);

for (int k = 0; k < size / 8; k++)

{

tmp = \_mm256\_mul\_ps(m256A, m256B[size \* j / 8 + k]);

m256ResTmp[size \* i / 8 + k] = \_mm256\_add\_ps(m256ResTmp[size \* i / 8 + k], tmp);

}

}

copyMatrix(result, resTmp, size);

delete[] resTmp;

}

void sumMatrixMy(float a[], float b[], float result[], int size) {

\_\_m256\* sumRes = (\_\_m256\*)result;

\_\_m256\* matrixA = (\_\_m256\*)a;

\_\_m256\* matrixB = (\_\_m256\*)b;

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

for (int j = 0; j < size / 8; j++) {

sumRes[i \* size / 8 + j] = \_mm256\_add\_ps(matrixA[i \* size / 8 + j], matrixB[i \* size / 8 + j]);

}

}

}

void multScalarMy(float A[], float scalar, int size) {

\_\_m256\* m256A = (\_\_m256\*)A;

\_\_m256 m256Scalar = \_mm256\_set1\_ps(scalar);

for (int i = 0; i < size \* size / 8; ++i)

m256A[i] = \_mm256\_mul\_ps(m256A[i], m256Scalar);

}

float\* getMatrixBMy(float matrix[], int size) {

float divider = findMaxColumnMy(matrix, size) \* findMaxLineMy(matrix, size);

float\* matrixB = new float[size \* size] {};

float a = 0;

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

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

matrixB[j \* size + i] = matrix[i \* size + j] / divider;

}

}

return matrixB;

}

float\* getMatrixRMy(float matrixA[], float matrixB[], int size) {

float\* matrixR = getOneMatrix(size);

float\* matrixMult = new float[size \* size] {};

multMatrixMy(matrixB, matrixA, matrixMult, size);

subMatrixMy(matrixR, matrixMult, matrixR, size);

delete[] matrixMult;

return matrixR;

}

float\* inverseMatrixMy(float matrix[], int size, int nSteps) {

float\* matrixB = getMatrixBMy(matrix, size);

float\* matrixR = getMatrixRMy(matrix, matrixB, size);

float\* matrixResult = getOneMatrix(size);

float\* matrixDegreeR = new float[size \* size] {};

copyMatrix(matrixDegreeR, matrixR, size);

addMatrixMy(matrixResult, matrixDegreeR, matrixResult, size);

for (int i = 1; i < nSteps; i++) {

multMatrixMy(matrixDegreeR, matrixR, matrixDegreeR, size);

addMatrixMy(matrixResult, matrixDegreeR, matrixResult, size);

}

multMatrixMy(matrixResult, matrixB, matrixResult, size);

delete[] matrixB;

delete[] matrixR;

delete[] matrixDegreeR;

return matrixResult;

}

int main()

{

const int n = 2048;

float\* matrix = new float[n \* n] {};

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

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

matrix[i \* n + j] = i + j;

}

}

double startTime = clock();

float\* c = inverseMatrixMy(matrix, n, 10);

double finishTime = clock();

std::cout << "Время выполнения: " << (finishTime - startTime) / CLOCKS\_PER\_SEC << "c" << std::endl;

delete[] matrix;

delete[] c;

}

# Приложение 3. Реализация программы с использованием библиотеки BLAS.

#include <iostream>

#include <openblas/cblas.h>

#include <ctime>

float findMaxLine(float matrix[], int size) {

float currentSum = 0;

float maxLine = 0;

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

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

currentSum += matrix[i \* size + j];

}

if (currentSum > maxLine) {

maxLine = currentSum;

}

currentSum = 0;

}

return maxLine;

}

float findMaxColumn(float matrix[], int size) {

float currentSum = 0;

float maxColumn = 0;

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

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

currentSum += matrix[j \* size + i];

}

if (currentSum > maxColumn) {

maxColumn = currentSum;

}

currentSum = 0;

}

return maxColumn;

}

float\* getMatrixB(float matrix[], int size) {

float divider = findMaxLine(matrix, size) \* findMaxColumn(matrix, size);

float\* matrixB = new float[size \* size] {};

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

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

matrixB[j \* size + i] = matrix[i \* size + j] / divider;

}

}

return matrixB;

}

float\* getOneMatrix(int size) {

float\* matrix = new float[size \* size] {};

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

matrix[i \* size + i] = 1;

}

return matrix;

}

void multMatrixBlas(float a[], float b[], float result[], int size) {

float\* resultTmp = new float[size \* size] {};

cblas\_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, size, size, size, 1.0, a, size, b, size, 0.0, resultTmp, size);

cblas\_scopy(size \* size, resultTmp, 1, result, 1);

delete[] resultTmp;

}

float\* inverseMatrixBlas(float matrix[], int size, int nSteps) {

float\* matrixB = getMatrixB(matrix, size);

float\* matrixR = getOneMatrix(size);

float\* matrixResult = getOneMatrix(size);

cblas\_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, size, size, size, -1.0, matrixB, size, matrix, size, 1.0, matrixR, size);

cblas\_saxpy(size \* size, 1.0, matrixR, 1, matrixResult, 1);

float\* matrixDegreeR = new float[size \* size] {};

cblas\_scopy(size \* size, matrixR, 1, matrixDegreeR, 1);

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

multMatrixBlas(matrixDegreeR, matrixR, matrixDegreeR, size);

cblas\_saxpy(size \* size, 1.0, matrixDegreeR, 1, matrixResult, 1);

}

multMatrixBlas(matrixResult, matrixB, matrixResult, size);

delete[] matrixB;

delete[] matrixR;

delete[] matrixDegreeR;

return matrixResult;

}

int main()

{

const int n = 2048;

float\* matrix = new float[n \* n] {};

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

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

matrix[i \* n + j] = i + j;

}

}

double startTime = clock();

float\* c = inverseMatrixBlas(matrix, n, 10);

double finishTime = clock();

std::cout << "Время выполнения: " << (finishTime - startTime) / CLOCKS\_PER\_SEC << "c" << std::endl;

delete[] matrix;

delete[] c;

}