**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 № 7**

«ВЕКТОРИЗАЦИЯ ВЫЧИСЛЕНИЙ»

студента (ки) \_\_\_\_2\_\_\_\_\_ курса, \_\_21212\_\_ группы

**Олимпиева Юрия Юрьевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

**А. А. Артюхов**

2022 год

**Цели**

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 итерационного метода получения обратной матрицы через разложение в ряд, - без ручной векторизации, с ручной векторизацией (через расширения компилятора gcc), на основе библиотечных функций BLAS.  
2. Проверить корректность работы алгоритмов на нескольких тестовых данных.  
3. Оптимизировать каждую программу по скорости.

4.Определить время работы программ для N=2048, M=10.

5. Составить отчет по лабораторной работе.

**Описание работы**

* Написал на языке программирования Си++ программы, выполняющие подсчёт обратной матрицы разложением в ряд — версию без векторизации, с ручной векторизацией через intrinsics расширение gcc, на основании библиотек gsl и blas.
* Измерил с помощью утилиты time время работы программ.
* Построил графики зависимости времени вычисления в зависимости от размеров массива и уровня оптимизации.
* Составил отчёт о выполненной работе.

**Заключение**

В результате проделанной работы изучил способы приложений векторных расширений, получил навыки работы с векторными расширениями в языках Си/Си++, а также получил навыки работы с библиотеками gsl, BLAS.

**Приложение 1**

**/\* Copyright 2022 Olimpiev Y. \*/**

**#include <cassert>**

**#include <gsl/gsl\_blas.h>**

**#include <cstdlib>**

**#include <iostream>**

**static void PrintMatrix(const gsl\_matrix\_float\* M) {**

**for (size\_t i = 0; i < M->size1; i++) {**

**for (size\_t j = 0; j < M->size2; j++) {**

**std::cout << gsl\_matrix\_float\_get(M, i, j) << " ";**

**}**

**std::cout << std::endl;**

**}**

**}**

**void MatrixAdd(gsl\_matrix\_float\* A, gsl\_matrix\_float\* B) {**

**assert(A->size1 == B->size1 && A->size2 == B->size2);**

**for (size\_t i = 0; i < A->size1; i++) {**

**auto aRow = gsl\_matrix\_float\_row(A, i);**

**auto bRow = gsl\_matrix\_float\_row(B, i);**

**gsl\_vector\_float\_add(&aRow.vector, &bRow.vector);**

**}**

**}**

**gsl\_matrix\_float\* InverseBlas(gsl\_matrix\_float\* A, size\_t seriesMemberAmount) {**

**assert(A);**

**assert(A->size1 == A->size2);**

**gsl\_matrix\_float\* I = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**gsl\_matrix\_float\_set\_identity(I);**

**gsl\_matrix\_float\* res = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**gsl\_matrix\_float\_set\_zero(res);**

**float norm1 = gsl\_matrix\_float\_norm1(A);**

**gsl\_matrix\_float\_transpose(A);**

**float normInf = gsl\_matrix\_float\_norm1(A);**

**gsl\_matrix\_float\_transpose(A);**

**assert(norm1 != 0.0);**

**assert(normInf != 0.0);**

**gsl\_matrix\_float\* R = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**gsl\_matrix\_float\_set\_identity(R);**

**// Calculate R = I - BA.**

**gsl\_blas\_sgemm(CblasTrans, CblasNoTrans, -1.0f / (norm1 \* normInf), A, A, 1.0, R);**

**gsl\_matrix\_float\* currMember = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**gsl\_matrix\_float\_set\_identity(currMember);**

**gsl\_matrix\_float\* tmp = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

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

**MatrixAdd(res, currMember);**

**gsl\_blas\_sgemm(CblasNoTrans, CblasNoTrans, 1.0, R, currMember, 0.0, tmp);**

**gsl\_matrix\_float\_memcpy(currMember, tmp);**

**}**

**gsl\_blas\_sgemm(CblasNoTrans, CblasTrans, 1.0f / (norm1 \* normInf), res, A, 0.0, tmp);**

**gsl\_matrix\_float\_free(I);**

**gsl\_matrix\_float\_free(R);**

**gsl\_matrix\_float\_free(currMember);**

**gsl\_matrix\_float\_free(res);**

**return tmp;**

**}**

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

**size\_t matrixSize = 2048;**

**gsl\_matrix\_float\* m = gsl\_matrix\_float\_alloc(matrixSize, matrixSize);**

**assert(m);**

**gsl\_matrix\_float\_set\_identity(m);**

**auto res = InverseBlas(m, 10);**

**PrintMatrix(res);**

**gsl\_matrix\_float\_free(m);**

**gsl\_matrix\_float\_free(res);**

**return EXIT\_SUCCESS;**

**}**

**Build on Linux:**

g++ main.cpp -o lab7blas[0 | 3] -O[0 | 3] -lgsl -lblas

**Приложение 2**

**/\* Copyright 2022 Olimpiev Y. \*/**

**#include <cassert>**

**#include <cstdlib>**

**#include <gsl/gsl\_blas.h>**

**#include <iostream>**

**#include <limits>**

**static void PrintMatrix(const gsl\_matrix\_float\* M) {**

**assert(M);**

**for (size\_t i = 0; i < M->size1; i++) {**

**for (size\_t j = 0; j < M->size2; j++) {**

**std::cout << gsl\_matrix\_float\_get(M, i, j) << " ";**

**}**

**std::cout << std::endl;**

**}**

**}**

**static gsl\_matrix\_float\* MatrixMultiplication(const gsl\_matrix\_float\* A, const gsl\_matrix\_float\* B, gsl\_matrix\_float\* res) {**

**assert(A && B);**

**assert(A->size2 == B->size1);**

**for (size\_t i = 0; i < A->size1; i++) {**

**for (size\_t k = 0; k < B->size2; k++) {**

**for (size\_t j = 0; j < A->size2; j++ ) {**

**res->data[i \* res->size1 + k] += A->data[i \* A->size1 + j] \* B->data[j \* B->size1 + k];**

**}**

**}**

**}**

**return res;**

**}**

**static gsl\_matrix\_float\* MatrixSub(gsl\_matrix\_float\* A, gsl\_matrix\_float\* B, gsl\_matrix\_float\* res) {**

**assert(A && B);**

**assert(A->size1 == B->size1 && A->size2 == B->size2);**

**for (size\_t row = 0; row < res->size1; row++) {**

**for (size\_t column = 0; column < res->size2; column++) {**

**res->data[row \* res->size1 + column] = A->data[row \* res->size1 + column] - B->data[row \* res->size1 + column];**

**}**

**}**

**return res;**

**}**

**static gsl\_matrix\_float\* MatrixSum(gsl\_matrix\_float\* A, gsl\_matrix\_float\* B, gsl\_matrix\_float\* res) {**

**assert(A && B);**

**assert(A->size1 == B->size1 && A->size2 == B->size2);**

**assert(res->size1 == A->size1 && res->size2 == A->size1);**

**for (size\_t row = 0; row < res->size1; row++) {**

**for (size\_t column = 0; column < res->size2; column++) {**

**res->data[row \* res->size1 + column] = A->data[row \* res->size1 + column] + B->data[row \* res->size1 + column];**

**}**

**}**

**return res;**

**}**

**static void ScalarMultiplication(gsl\_matrix\_float\* A, float scalar) {**

**assert(A);**

**for (size\_t row = 0; row < A->size1; row++) {**

**for (size\_t column = 0; column < A->size2; column++) {**

**A->data[row \* A->size1 + column] \*= scalar;**

**}**

**}**

**}**

**static void TransposeMatrix(gsl\_matrix\_float\* A) {**

**assert(A);**

**assert(A->size1 == A->size2);**

**for (size\_t row = 0; row < A->size1; row++) {**

**for (size\_t column = row; column < A->size2; column++) {**

**std::swap(A->data[row \* A->size1 + column], A->data[column \* A->size2 + row]);**

**}**

**}**

**}**

**static float MatrixNorm1(const gsl\_matrix\_float\* A) {**

**assert(A);**

**float norm = std::numeric\_limits<float>::min();**

**for (size\_t column = 0; column < A->size2; column++) {**

**float sum = 0.0f;**

**for (size\_t row = 0; row < A->size1; row++) {**

**sum += A->data[row \* A->size1 + column];**

**}**

**norm = std::max(sum, norm);**

**}**

**return norm;**

**}**

**static float MatrixNormInf(const gsl\_matrix\_float\* A) {**

**assert(A);**

**float norm = std::numeric\_limits<float>::min();**

**for (size\_t row = 0; row < A->size1; row++) {**

**float sum = 0.0f;**

**for (size\_t column = 0; column < A->size2; column++) {**

**sum += A->data[row \* A->size1 + column];**

**}**

**norm = std::max(norm, sum);**

**}**

**return norm;**

**}**

**static gsl\_matrix\_float\* InverseCustom(const gsl\_matrix\_float\* A, size\_t seriesMemberAmount) {**

**assert(A);**

**assert(A->size1 == A->size2);**

**gsl\_matrix\_float\* I = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**gsl\_matrix\_float\_set\_identity(I);**

**gsl\_matrix\_float\* B = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**gsl\_matrix\_float\_memcpy(B, A);**

**TransposeMatrix(B);**

**ScalarMultiplication(B, 1.0f / (MatrixNorm1(A) \* MatrixNormInf(A)));**

**gsl\_matrix\_float\* res = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**gsl\_matrix\_float\* tmp = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**MatrixMultiplication(B, A, tmp);**

**gsl\_matrix\_float\* R = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**MatrixSub(I, tmp, R);**

**gsl\_matrix\_float\* seriesMember = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**gsl\_matrix\_float\_set\_identity(seriesMember);**

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

**gsl\_matrix\_float\_set\_zero(tmp);**

**// res += R^i. seriesMember + res -> res.**

**MatrixSum(res, seriesMember, res);**

**// Calc R^i. seriesMember \* R -> tmp.**

**MatrixMultiplication(seriesMember, R, tmp);**

**// Swap tmp and curr series member.**

**std::swap(tmp, seriesMember);**

**}**

**MatrixMultiplication(res, B, tmp);**

**gsl\_matrix\_float\_free(I);**

**gsl\_matrix\_float\_free(R);**

**gsl\_matrix\_float\_free(B);**

**gsl\_matrix\_float\_free(res);**

**gsl\_matrix\_float\_free(seriesMember);**

**return tmp;**

**}**

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

**size\_t matrixSize = 2048;**

**gsl\_matrix\_float\* m = gsl\_matrix\_float\_alloc(matrixSize, matrixSize);**

**assert(m);**

**gsl\_matrix\_float\_set\_identity(m);**

**auto res = InverseCustom(m, 10);**

**PrintMatrix(res);**

**gsl\_matrix\_float\_free(res);**

**gsl\_matrix\_float\_free(m);**

**return EXIT\_SUCCESS;**

**}**

**Build on Linux:**

g++ main.cpp -o lab7custom[0 | 3] -O[0 | 3] -lgsl -lblas

**Приложение 3**

**/\* Copyright 2022 Olimpiev Y. \*/**

**#include <cassert>**

**#include <cstdlib>**

**#include <gsl/gsl\_blas.h>**

**#include <iostream>**

**#include <limits>**

**typedef float v4f \_\_attribute\_\_ ((vector\_size (16)));**

**const size\_t vecLen = 4;**

**static void PrintMatrix(const gsl\_matrix\_float\* M) {**

**assert(M);**

**for (size\_t i = 0; i < M->size1; i++) {**

**for (size\_t j = 0; j < M->size2; j++) {**

**std::cout << gsl\_matrix\_float\_get(M, i, j) << " ";**

**}**

**std::cout << std::endl;**

**}**

**}**

**static void TransposeMatrix(gsl\_matrix\_float\* A) {**

**assert(A);**

**assert(A->size1 == A->size2);**

**for (size\_t row = 0; row < A->size1; row++) {**

**for (size\_t column = row; column < A->size2; column++) {**

**std::swap(A->data[row \* A->size1 + column], A->data[column \* A->size2 + row]);**

**}**

**}**

**}**

**static float dot(float\* a, float\* b, size\_t n) {**

**assert(a);**

**assert(b);**

**v4f res = {0.0f, 0.0f, 0.0f, 0.0f};**

**v4f\* aa = (v4f\*)a;**

**v4f\* bb = (v4f\*)b;**

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

**res += aa[i] \* bb[i];**

**}**

**return res[0] + res[1] + res[2] + res[3];**

**}**

**static gsl\_matrix\_float\* MatrixMultiplication(const gsl\_matrix\_float\* A, gsl\_matrix\_float\* B, gsl\_matrix\_float\* res) {**

**assert(A && B);**

**assert(A->size2 == B->size1);**

**v4f\* resVec = (v4f\*)res->data;**

**v4f\* bVec = (v4f\*)B->data;**

**TransposeMatrix(B);**

**for (size\_t i = 0; i < A->size1 ; i++) {**

**for (size\_t j = 0; j < A->size2 ; j++ ) {**

**res->data[i \* res->size1 + j] = dot(&A->data[i \* A->size1], &B->data[j \* B->size1], A->size2 / 4);**

**}**

**}**

**TransposeMatrix(B);**

**return res;**

**}**

**static gsl\_matrix\_float\* MatrixSub(const gsl\_matrix\_float\* A, const gsl\_matrix\_float\* B, gsl\_matrix\_float\* res) {**

**assert(A && B);**

**assert(A->size1 == B->size1 && A->size2 == B->size2);**

**v4f\* aVec = (v4f\*)A->data;**

**v4f\* bVec = (v4f\*)B->data;**

**v4f\* resVec = (v4f\*)res->data;**

**size\_t size = A->size1 \* A->size2;**

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

**resVec[i] = aVec[i] - bVec[i];**

**}**

**return res;**

**}**

**static gsl\_matrix\_float\* MatrixSum(const gsl\_matrix\_float\* A, const gsl\_matrix\_float\* B, gsl\_matrix\_float\* res) {**

**assert(A && B);**

**assert(A->size1 == B->size1 && A->size2 == B->size2);**

**v4f\* aVec = (v4f\*)A->data;**

**v4f\* bVec = (v4f\*)B->data;**

**v4f\* resVec = (v4f\*)res->data;**

**size\_t size = A->size1 \* A->size2;**

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

**resVec[i] = aVec[i] + bVec[i];**

**}**

**return res;**

**}**

**static void ScalarMultiplication(gsl\_matrix\_float\* A, float scalar) {**

**assert(A);**

**v4f\* aVec = (v4f\*)A->data;**

**size\_t size = A->size1 \* A->size2;**

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

**aVec[i] \*= scalar;**

**}**

**}**

**static float MatrixNorm1(const gsl\_matrix\_float\* A) {**

**assert(A);**

**float norm = std::numeric\_limits<float>::min();**

**for (size\_t column = 0; column < A->size2; column++) {**

**float sum = 0.0f;**

**for (size\_t row = 0; row < A->size1; row++) {**

**sum += A->data[row \* A->size1 + column];**

**}**

**norm = std::max(sum, norm);**

**}**

**return norm;**

**}**

**static float MatrixNormInf(const gsl\_matrix\_float\* A) {**

**assert(A);**

**float norm = std::numeric\_limits<float>::min();**

**for (size\_t row = 0; row < A->size1; row++) {**

**float sum = 0.0f;**

**for (size\_t column = 0; column < A->size2; column++) {**

**sum += A->data[row \* A->size1 + column];**

**}**

**norm = std::max(norm, sum);**

**}**

**return norm;**

**}**

**static void InverseCustom(gsl\_matrix\_float\* A, gsl\_matrix\_float\* res, size\_t seriesMemberAmount) {**

**assert(A);**

**assert(res);**

**assert(A->size1 == A->size2);**

**assert(res->size1 == A->size1 && res->size2 == A->size2);**

**// Set I matrix for calc.**

**gsl\_matrix\_float\* I = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**gsl\_matrix\_float\_set\_identity(I);**

**// Calc B = A^T / (norm1(A) \* normInf(A)).**

**gsl\_matrix\_float\* B = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**gsl\_matrix\_float\_memcpy(B, A);**

**TransposeMatrix(B);**

**ScalarMultiplication(B, 1.0f / (MatrixNorm1(A) \* MatrixNormInf(A)));**

**// Reserve tmp matrix for buffering.**

**gsl\_matrix\_float\* tmp = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**// tmp = B\*A.**

**MatrixMultiplication(B, A, tmp);**

**// Calc R = I - B\*A.**

**gsl\_matrix\_float\* R = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**MatrixSub(I, tmp, R);**

**// Reserve memory for**

**gsl\_matrix\_float\* seriesMember = gsl\_matrix\_float\_calloc(A->size1, A->size2);**

**gsl\_matrix\_float\_set\_identity(seriesMember);**

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

**gsl\_matrix\_float\_set\_zero(tmp);**

**// res += R^i. seriesMember + res -> res.**

**MatrixSum(seriesMember, res, res);**

**// Calc R^i. seriesMember \* R -> tmp.**

**MatrixMultiplication(seriesMember, R, tmp);**

**std::swap(seriesMember, tmp);**

**}**

**// Calc (I + R + R^2 + ...) \* B.**

**gsl\_matrix\_float\_set\_zero(tmp);**

**MatrixMultiplication(res, B, tmp);**

**gsl\_matrix\_float\_memcpy(res, tmp);**

**gsl\_matrix\_float\_free(I);**

**gsl\_matrix\_float\_free(R);**

**gsl\_matrix\_float\_free(B);**

**gsl\_matrix\_float\_free(tmp);**

**gsl\_matrix\_float\_free(seriesMember);**

**}**

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

**size\_t matrixSize = 2048;**

**gsl\_matrix\_float\* m = gsl\_matrix\_float\_alloc(matrixSize, matrixSize);**

**assert(m);**

**gsl\_matrix\_float\_set\_identity(m);**

**gsl\_matrix\_float\* res = gsl\_matrix\_float\_calloc(matrixSize, matrixSize);**

**assert(res);**

**InverseCustom(m, res, 10);**

**PrintMatrix(res);**

**gsl\_matrix\_float\_free(m);**

**gsl\_matrix\_float\_free(res);**

**return EXIT\_SUCCESS;**

**}**

**Build on Linux:**

g++ main.cpp -o lab7intrinsics[0 | 3] -O[0 | 3] -lgsl -lblas

**Приложение 4**

График зависимости времени вычисления обратной матрицы в зависимости от размера матрицы.