# ЭВМ и периферийные устройства

## Отчёт по лабораторной работе №6

## «Векторизация вычислений»

## Выполнил: Скопинцев Н.А. ФИТ НГУ 2 курс

## Преподаватель:

## Цель работы:

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.

## Листинг программы без векторизации

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

#include <math.h>

#include <time.h>

size\_t N = 2048; // размер матрицы

size\_t M = 10; //кол-во итераций в цикле

**void** printMatrix(**float**\* A) {

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

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

printf("%f ", A[i \* N + j]);

}

printf("\n");

}

printf("\n");

}

**float** maxLine(**float**\* A) {

**float** sum = 0;

**float** max = 0;

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

sum = 0;

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

sum += fabs(A[i \* N + j]);

}

**if** (sum > max) {

max = sum;

}

}

**return** max;

}

**float** maxColumn(**float**\* A) {

**float** sum = 0;

**float** max = 0;

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

sum = 0;

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

sum += fabs(A[i \* N + j]);

}

**if** (sum > max) {

max = sum;

}

}

**return** max;

}

**void** multMatrixs(**float**\* C, **float**\* A, **float**\* B) {

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

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

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

c[j] = 0;

}

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

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

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

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

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

}

}

}

}

**float**\* createMatrixB(**float**\* A) {

**float**\* B = (**float**\*)calloc(N \* N, **sizeof**(**float**));

**float** multOfMax = maxLine(A) \* maxColumn(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] / multOfMax;

}

}

**return** B;

}

**int** **main**() {

**struct** timespec start, end;

// созда�'м все нужные нам матрицы

**float**\* A = NULL; // исходная матрица

**float**\* B = NULL; // B = AT / |A|\_1 \* |A|\_inf

**float**\* R = NULL; // R = I - BA

**float**\* A\_rev = NULL; // Обратная матрица исходной матрицы

**float**\* C1 = NULL; // tmp1 - матрица

**float**\* C2 = NULL; // tmp2 - матрица

**float**\* C3 = NULL; // tmp3 - матрица

A = (**float**\*)calloc(N \* N, **sizeof**(**float**));

R = (**float**\*)calloc(N \* N, **sizeof**(**float**));

C1 = (**float**\*)calloc(N \* N, **sizeof**(**float**));

C2 = (**float**\*)calloc(N \* N, **sizeof**(**float**));

C3 = (**float**\*)calloc(N \* N, **sizeof**(**float**));

A\_rev = (**float**\*)calloc(N \* N, **sizeof**(**float**));

srand((**float**)time(NULL));

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

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

//scanf("%f", &A[i \* N + j]);

A[i \* N + j] = (**float**)rand();

}

}

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

//--�'ычисляем матрицу B----------

B = createMatrixB(A);

//-------------------------

//-- �'ычисляем матрицу R------------

multMatrixs(C1, B, A);

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

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

**if** (i == j) {

R[i \* N + j] = 1 - C1[i \* N + j];

}

**else** {

R[i \* N + j] = -C1[i \* N + j];

}

}

}

//---�'ычисляем обратную матрицу------

// сложили I + R

// C1 - сумма

// C2 - R в степени n

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

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

C1[i \* N + j] = R[i \* N + j];

C2[i \* N + j] = R[i \* N + j];

}

C1[i \* N + i] += 1;

}

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

multMatrixs(C3, R, C2);

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

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

C2[i \* N + j] = C3[i \* N + j];

C1[i \* N + j] += C3[i \* N + j];

}

}

}

// и умножаем результат на B

multMatrixs(A\_rev, C1, B);

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

printf("Time taken: %lf sec.\n", end.tv\_sec - start.tv\_sec

+ 0.000000001 \* (end.tv\_nsec - start.tv\_nsec));

//---------Проверка------------

multMatrixs(C1, A, A\_rev);

**float** check = maxLine(C1) \* maxColumn(C1);

printf("Проверка матрицы на единичную:\nMaxSumCol(A \* A^-1) \* MaxSumLine(A \* A^-1) = %f\n", check);

// освобождаем память от всех матриц

free(A);

free(B);

free(R);

free(A\_rev);

free(C1);

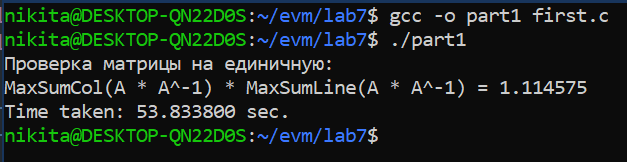
free(C2);

free(C3);

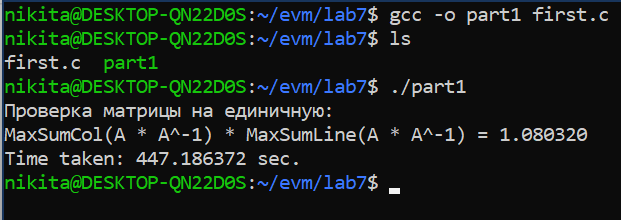
**return** 0;

}

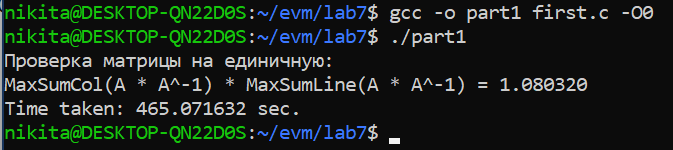
N = 1024



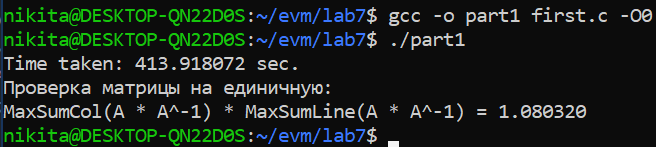
N = 2048



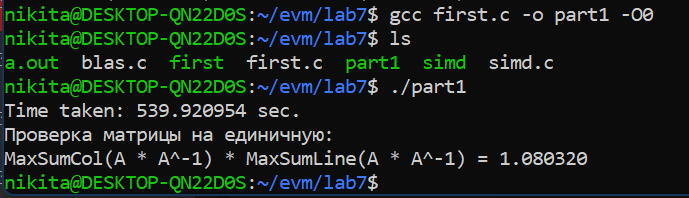
С оптимизацией -O0 (на всякий случай чтоб сам не оптимизировал)

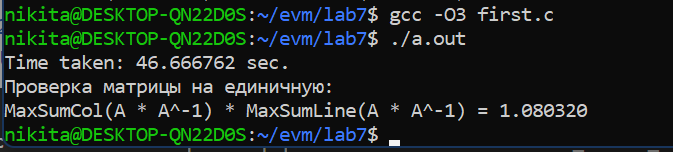


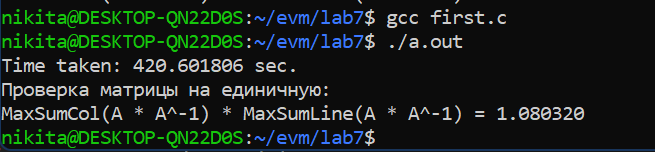
Переставил таймер до вычислений обратной матрицы для проверки



Последняя версия программы:







## Листинг с использованием ручной векторизацией

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

#include <time.h>

#include <math.h>

#include <immintrin.h>

size\_t N = 2048; // размер матрицы

size\_t M = 10; //кол-во итераций в цикле

**void** printMatrix(**float**\* A) {

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

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

printf("%f ", A[i \* N + j]);

}

printf("\n");

}

printf("\n");

}

**float** maxLine(**float**\* A) {

**float** sum = 0;

**float** max = 0;

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

sum = 0;

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

sum += fabs(A[i \* N + j]);

}

**if** (sum > max) {

max = sum;

}

}

**return** max;

}

**float** maxColumn(**float**\* A) {

**float** sum = 0;

**float** max = 0;

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

sum = 0;

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

sum += fabs(A[i \* N + j]);

}

**if** (sum > max) {

max = sum;

}

}

**return** max;

}

**void** multMatrixs(**float**\* C, **float**\* A, **float**\* B) {

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

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

**for** (**int** j = 0; j < N; j += 8) {

\_mm256\_store\_ps(c + j + 0, \_mm256\_setzero\_ps());

}

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

**float**\* b = B + k \* N;

\_\_m256 a = \_mm256\_set1\_ps(A[i\*N + k]);

**for** (**int** j = 0; j < N; j += 8) {

\_mm256\_store\_ps(c + j + 0, \_mm256\_fmadd\_ps(a, \_mm256\_load\_ps(b + j + 0), \_mm256\_load\_ps(c + j + 0)));

}

}

}

}

**float**\* createMatrixB(**float**\* A, **float**\* memory) {

**float**\* B = memory + 6 \* N \* N;

**float** multOfMax = maxLine(A) \* maxColumn(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] / multOfMax;

}

}

**return** B;

}

**int** **main**() {

**struct** timespec start, end;

// созда�'м все нужные нам матрицы

**float**\* A = NULL; // исходная матрица

**float**\* B = NULL; // B = AT / |A|\_1 \* |A|\_inf

**float**\* R = NULL; // R = I - BA

**float**\* A\_rev = NULL; // Обратная матрица исходной матрицы

**float**\* C1 = NULL; // tmp1 - матрица

**float**\* C2 = NULL; // tmp2 - матрица

**float**\* C3 = NULL; // tmp3 - матрица

**float**\* memory = \_mm\_malloc(7 \* N \* N \* **sizeof**(**float**), 32);

A = memory;

R = memory + N \* N;

C1 = memory + 2 \* N \* N;

C2 = memory + 3 \* N \* N;

C3 = memory + 4 \* N \* N;

A\_rev = memory + 5 \* N \* N;

//srand((float)time(NULL));

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

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

//scanf("%f", &A[i \* N + j]);

A[i \* N + j] = (**float**)rand();

}

}

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

//--�'ычисляем матрицу B----------

B = createMatrixB(A, memory);

//-------------------------

//-- �'ычисляем матрицу R------------

multMatrixs(C1, B, A);

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

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

**if** (i == j) {

R[i \* N + j] = 1 - C1[i \* N + j];

}

**else** {

R[i \* N + j] = -C1[i \* N + j];

}

}

}

//---�'ычисляем обратную матрицу------

// сложили I + R

// C1 - сумма

// C2 - R в степени n

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

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

C1[i \* N + j] = R[i \* N + j];

C2[i \* N + j] = R[i \* N + j];

}

C1[i \* N + i] += 1;

}

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

multMatrixs(C3, R, C2);

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

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

C2[i \* N + j] = C3[i \* N + j];

C1[i \* N + j] += C3[i \* N + j];

}

}

}

// и умножаем результат на B

multMatrixs(A\_rev, C1, B);

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

printf("Time taken: %lf sec.\n", end.tv\_sec - start.tv\_sec

+ 0.000000001 \* (end.tv\_nsec - start.tv\_nsec));

//---------Проверка------------

multMatrixs(C1, A, A\_rev);

**float** check = maxLine(C1) \* maxColumn(C1);

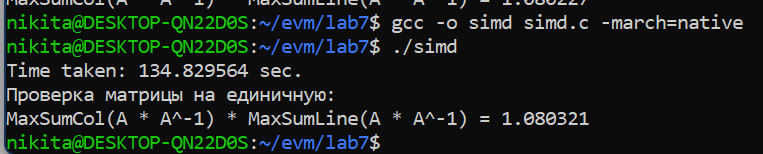
printf("Проверка матрицы на единичную:\nMaxSumCol(A \* A^-1) \* MaxSumLine(A \* A^-1) = %f\n", check);

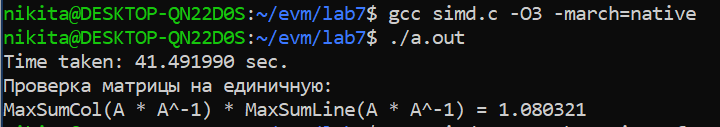
// освобождаем память от всех матриц

\_mm\_free(memory);

**return** 0;

}





## Листинг с использованием библиотеки CBLAS

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

#include <math.h>

#include <time.h>

#include <cblas.h>

size\_t N = 2048; // размер матрицы

size\_t M = 10; //кол-во итераций в цикле

**void** printMatrix(**float**\* A) {

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

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

printf("%f ", A[i \* N + j]);

}

printf("\n");

}

printf("\n");

}

**float** maxLine(**float**\* A) {

**float** sum = 0;

**float** max = 0;

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

sum = 0;

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

sum += fabs(A[i \* N + j]);

}

**if** (sum > max) {

max = sum;

}

}

**return** max;

}

**float** maxColumn(**float**\* A) {

**float** sum = 0;

**float** max = 0;

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

sum = 0;

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

sum += fabs(A[i \* N + j]);

}

**if** (sum > max) {

max = sum;

}

}

**return** max;

}

**float**\* createMatrixB(**float**\* A, **float**\* memory) {

**float**\* B = memory + 6 \* N \* N;

**float** multOfMax = maxLine(A) \* maxColumn(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] / multOfMax;

}

}

**return** B;

}

**int** **main**() {

**struct** timespec start, end;

// созда�'м все нужные нам матрицы

**float**\* A = NULL; // исходная матрица

**float**\* B = NULL; // B = AT / |A|\_1 \* |A|\_inf

**float**\* R = NULL; // R = I - BA

**float**\* A\_rev = NULL; // Обратная матрица исходной матрицы

**float**\* C1 = NULL; // tmp1 - матрица

**float**\* C2 = NULL; // tmp2 - матрица

**float**\* C3 = NULL; // tmp3 - матрица

**float**\* memory = malloc(7 \* N \* N \* **sizeof**(**float**));

A = memory;

R = memory + N \* N;

C1 = memory + 2 \* N \* N;

C2 = memory + 3 \* N \* N;

C3 = memory + 4 \* N \* N;

A\_rev = memory + 5 \* N \* N;

srand((**float**)time(NULL));

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

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

//scanf("%f", &A[i \* N + j]);

A[i \* N + j] = (**float**)rand();

}

}

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

//--�'ычисляем матрицу B----------

B = createMatrixB(A, memory);

//-------------------------

//-- �'ычисляем матрицу R------------

cblas\_sgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans, N, N, N, 1.0 , B, N, A, N, 0.0, C1, N);

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

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

**if** (i == j) {

R[i \* N + j] = 1 - C1[i \* N + j];

}

**else** {

R[i \* N + j] = -C1[i \* N + j];

}

}

}

//---�'ычисляем обратную матрицу------

// сложили I + R

// C1 - сумма

// C2 - R в степени n

cblas\_scopy(N\*N, R, 1, C1, 1);

cblas\_scopy(N\*N, R, 1, C2, 1);

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

C1[i \* N + i] += 1;

}

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

/\*multMatrixs(C3, R, C2);\*/

cblas\_sgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans, N, N, N, 1.0 , R, N, C2, N, 0.0, C3, N);

cblas\_scopy(N\*N, C3, 1, C2, 1);

cblas\_saxpy(N\*N, 1.0, C3, 1, C1, 1);

}

// и умножаем результат на B

cblas\_sgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans, N, N, N, 1.0 , C1, N, B, N, 0.0, A\_rev, N);

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

printf("Time taken: %lf sec.\n", end.tv\_sec - start.tv\_sec

+ 0.000000001 \* (end.tv\_nsec - start.tv\_nsec));

//---------Проверка------------

cblas\_sgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans, N, N, N, 1.0 , A, N, A\_rev, N, 0.0, C1, N);

**float** check = maxLine(C1) \* maxColumn(C1);

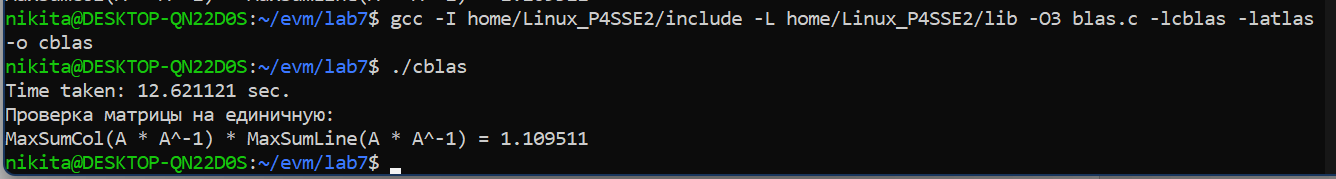
printf("Проверка матрицы на единичную:\nMaxSumCol(A \* A^-1) \* MaxSumLine(A \* A^-1) = %f\n", check);

// освобождаем память от всех матриц

free(memory);

**return** 0;

}



**Ключи компиляции:**

gcc firsct.c -o part1 -O0

./part1

gcc simd.c -o simd -march=native

./simd

gcc -I home/Linux\_P4SSE2/include -L home/Linux\_P4SSE2/lib -O3 blas.c -lcblas -latlas -o cblas

./cblas

|  |  |  |  |
| --- | --- | --- | --- |
|  | Без ручной векторизации | С ручной векторизацией | С использование  BLAS |
| Без оптимизиаций | 420 секунд | 134 секунды |  |
| -O3 | 46 секунд | 41,5 секунды | 12,5 секунды |

**Время**

## Вывод

В процессе выполнения работы была получена информация о SIMD-расширениях архитектуры x86/x86-64. SIMD-расширения предназначены для ускорения вычислений однотипных данных.

* Без ручной векторизации в среднем программа работала 500 секунд
* С ручной векторизацией 2 минуты
* С библиотекой BLAS от 12 до 13 секунд

Также было проведено сравнение по времени работы одного и того же алгоритма, реализованного без векторизации на SIMD-расширениях, с ручной векторизацией и с векторизацией средствами библиотеки BLAS. По результатам нетрудно заметить, что самым оптимальным по времени является способ с использованием библиотеки BLAS, а самым читаемым для постороннего человека - без векторизации (с учётом всех подпрограмм).