###### 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

«ВЕКТОРИЗАЦИЯ ВЫЧИСЛЕНИЙ»

студента 2 курса, группы 21209

Усольцева Антона Андреевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

М.А.Мичуров

В.А.Перепёлкин

Новосибирск 2022

**СОДЕРЖАНИЕ**

Цель 3

Задачи 3

Описание работы 4

Заключение 11

# 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.

# ЗАДАЧИ

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.

2. Проверить правильность работы программ на нескольких небольших тестовых наборах входных данных.

3. Сравнить время работы трех вариантов программы для N=2048, M=10.

# ОПИСАНИЕ РАБОТЫ

Все программы запускались на сервере.

Листинг 1

#include <stdio.h>  
#include <malloc.h>  
#include <math.h>  
#include <time.h>  
  
#define N 2048  
#define M 10  
  
void create\_matrix(float \*Matr, float value1, float value2) {  
 for (int i = 0; i < N; i++)  
 for (int j = 0; j < N; j++)  
 if (i == j)  
 Matr[i \* N + j] = value1;  
 else  
 Matr[i \* N + j] = value2;  
}  
  
float find\_max\_str(const float \*A) {  
 float a\_inf = -1;  
 float tmp = 0;  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++)  
 tmp += fabsf(A[i \* N + j]);  
 if (tmp > a\_inf)  
 a\_inf = tmp;  
 tmp = 0;  
 }  
 return a\_inf;  
}  
  
float find\_max\_column(const float \*A) {  
 float a\_1 = -1;  
 float tmp = 0;  
 for (int j = 0; j < N; j++) {  
 for (int i = 0; i < N; i++)  
 tmp += fabsf(A[j \* N + i]);  
 if (tmp > a\_1)  
 a\_1 = tmp;  
 tmp = 0;  
 }  
 return a\_1;  
}

void create\_B\_matrix(const float \*A, float \*B, float a\_1, float a\_inf) {  
 float koef = a\_1 \* a\_inf;  
 for (int i = 0; i < N; i++)  
 for (int j = 0; j < N; j++)  
 B[i \* N + j] = A[j \* N + i] / koef;  
}  
  
void transposition(const float \*A, float \*A\_t) {  
 for (int i = 0; i < N; i++)  
 for (int j = 0; j < N; j++)  
 A\_t[i \* N + j] = A[j \* N + i];  
}  
  
void mult(const float \*A, const float \*B\_t, float \*C) {  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++) {  
 float summ = 0;  
 for (int k = 0; k < N; k++)  
 summ += A[i \* N + k] \* B\_t[j \* N + k];  
 C[i \* N + j] = summ;  
 }  
 }  
}  
  
void summ(const float \*A, const float \*B, float \*C) {  
 for (int i = 0; i < N \* N; i++)  
 C[i] = A[i] + B[i];  
}  
  
void print\_matr(const float \*A) {  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++)  
 printf("%lf ", A[i \* N + j]);  
 printf("\n");  
 }  
 printf("\n");  
}  
  
float count\_trace(const float \*A) {  
 float trace;  
 for (int i = 0; i < N; i++)  
 for (int j = 0; j < N; j++)  
 if (i == j)  
 trace += A[i \* N + j];  
 return trace;  
}  
  
int main() {  
 float \*A = calloc(N \* N, sizeof(float));  
 float \*A\_1 = calloc(N \* N, sizeof(float));  
 float \*A\_1\_t = calloc(N \* N, sizeof(float));  
 float \*A\_t = calloc(N \* N, sizeof(float));  
 float \*R\_t = calloc(N \* N, sizeof(float));  
 float \*B\_t = calloc(N \* N, sizeof(float));  
 float \*I = calloc(N \* N, sizeof(float));  
 float \*B = calloc(N \* N, sizeof(float));  
 float \*R = calloc(N \* N, sizeof(float));  
 float \*cur = calloc(N \* N, sizeof(float));  
 float \*cur\_2 = calloc(N \* N, sizeof(float));  
  
 clock\_t beg = clock();  
  
 create\_matrix(A, 4.0f, 3.0f);  
 create\_matrix(I, 1.0f, 0.0f);  
  
 float a\_1 = find\_max\_column(A);  
 float a\_inf = find\_max\_str(A);  
  
 create\_B\_matrix(A, B, a\_1, a\_inf);  
  
 transposition(A, A\_t);  
 mult(B, A\_t, R);  
  
 for (int i = 0; i < N \* N; i++)  
 R[i] = I[i] - R[i];  
  
 for (int i = 0; i < N; i++)  
 for (int j = 0; j < N; j++)  
 cur\_2[i \* N + j] = R[i \* N + j];  
  
 transposition(R, R\_t);  
  
 // cur == R^(m-1), cur\_2 = R^m  
 for (int m = 0; m < M; m++) {  
 for (int i = 0; i < N; i++)  
 for (int j = 0; j < N; j++)  
 cur[i \* N + j] = cur\_2[i \* N + j];  
 summ(I, cur, I);  
 mult(cur, R\_t, cur\_2);  
 }  
  
 transposition(B, B\_t);  
 mult(I, B\_t, A\_1);  
  
 clock\_t end = clock();  
  
 transposition(A\_1, A\_1\_t);  
 mult(A, A\_1\_t, cur);  
  
 float trace = count\_trace(cur);  
  
 double time\_used = ((double) (end - beg)) / CLOCKS\_PER\_SEC;  
 printf("Time: %lf\n", time\_used);  
 printf("Trace: %f\n", trace);  
  
 free(A); free(A\_1); free(A\_t);  
 free(A\_1\_t); free(B); free(B\_t);  
 free(I); free(R); free(R\_t);  
 free(cur); free(cur\_2);  
 return 0;  
}

1. Листинг 1 - программа, вычисляющая обратную матрицу без использования какой-либо векторизации.

Проверим корректность вычислений на примере:

N=4,

M=10000,

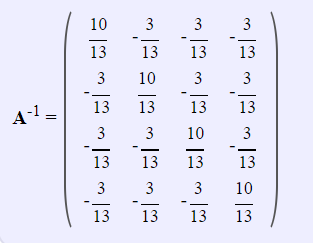
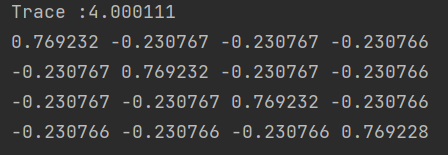
A=4333

3433

3343

3334

Тогда обратная матрица:



Также в конце программы считается след матрицы A\*A-1, который должен быть примерно равен N. В проверке след как раз получился примерно 4.

Для N=2048, M=10 и уровня оптимизации O3 время работы ~ 103 сек.

(так как шагов всего 10, то подсчет следа будет не очень полезным с точки зрения получения числа N, но с помощью следа можно будет проследить, верно ли считается матрица в остальных способах).



2. Добавим к Листингу 1 подключение заголовочного файла с описаниями SIMD intrinsics для расширения SSE и заменим функцию mult и sum

Листинг 2

#include <stdio.h>  
#include <malloc.h>  
#include <math.h>  
#include <xmmintrin.h>  
#include <time.h>  
  
#define N 2048  
#define M 10  
  
void create\_matrix(float \*Matr, float value1, float value2) {  
 for (int i = 0; i < N; i++)  
 for (int j = 0; j < N; j++)  
 if (i == j)  
 Matr[i \* N + j] = value1;  
 else  
 Matr[i \* N + j] = value2;  
}

float find\_max\_str(const float \*A) {  
 float a\_inf = -1;  
 float tmp = 0;  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++)  
 tmp += fabsf(A[i \* N + j]);  
 if (tmp > a\_inf)  
 a\_inf = tmp;  
 tmp = 0;  
 }  
 return a\_inf;  
}

float find\_max\_column(const float \*A) {  
 float a\_1 = -1;  
 float tmp = 0;  
 for (int j = 0; j < N; j++) {  
 for (int i = 0; i < N; i++)  
 tmp += fabsf(A[j \* N + i]);  
 if (tmp > a\_1)  
 a\_1 = tmp;  
 tmp = 0;  
 }  
 return a\_1;  
}  
  
void create\_B\_matrix(const float \*A, float \*B, float a\_1, float a\_inf) {  
 float koef = a\_1 \* a\_inf;  
 for (int i = 0; i < N; i++)  
 for (int j = 0; j < N; j++)  
 B[i \* N + j] = A[j \* N + i] / koef;  
}  
  
void transposition(const float \*A, float \*A\_t) {  
 for (int i = 0; i < N; i++)  
 for (int j = 0; j < N; j++)  
 A\_t[i \* N + j] = A[j \* N + i];  
}  
  
float scalar\_mult(const float \*A, const float \*B, int n) {  
 \_\_m128 \*xx, \*yy;  
 \_\_m128 p, s;  
 xx = (\_\_m128 \*) A;  
 yy = (\_\_m128 \*) B;  
 s = \_mm\_setzero\_ps();  
 for (int i = 0; i < n / 4; ++i) {  
 p = \_mm\_mul\_ps(xx[i], yy[i]);  
 s = \_mm\_add\_ps(s, p);  
 }  
 p = \_mm\_movehl\_ps(p, s);  
 s = \_mm\_add\_ps(s, p);  
 p = \_mm\_shuffle\_ps(s, s, 1);  
 s = \_mm\_add\_ss(s, p);  
 float sum;  
 \_mm\_store\_ss(&sum, s);  
 return sum;  
}  
  
void mult\_vectorize(const float \*A, const float \*B\_t, float \*C) {  
 for (int i = 0; i < N; i++)   
 for (int j = 0; j < N; j++)   
 C[i \* N + j] = scalar\_mult(&A[i \* N], &B\_t[j \* N], N);  
}  
  
void summ\_vectorize(const float \*A, const float \*B, float \*C) {  
 \_\_m128 \*m128\_result = (\_\_m128 \*) C;  
 \_\_m128 \*m128\_left = (\_\_m128 \*) A;  
 \_\_m128 \*m128\_right = (\_\_m128 \*) B;  
 for (int i = 0; i < N \* N / 4; ++i)  
 m128\_result[i] = \_mm\_add\_ps(m128\_left[i], m128\_right[i]);  
}  
void print\_matr(const float \*A) {  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++)  
 printf("%lf ", A[i \* N + j]);  
 printf("\n");  
 }  
 printf("\n");  
}  
float count\_trace(const float \*A) {  
 float trace;  
 for (int i = 0; i < N; i++)  
 for (int j = 0; j < N; j++)  
 if (i == j)  
 trace += A[i \* N + j];  
 return trace;  
}  
  
int main() {  
 float \*A = calloc(N \* N, sizeof(float));  
 float \*A\_1 = calloc(N \* N, sizeof(float));  
 float \*A\_1\_t = calloc(N \* N, sizeof(float));  
 float \*A\_t = calloc(N \* N, sizeof(float));  
 float \*R\_t = calloc(N \* N, sizeof(float));  
 float \*B\_t = calloc(N \* N, sizeof(float));  
 float \*I = calloc(N \* N, sizeof(float));  
 float \*B = calloc(N \* N, sizeof(float));  
 float \*R = calloc(N \* N, sizeof(float));  
 float \*cur = calloc(N \* N, sizeof(float));  
 float \*cur\_2 = calloc(N \* N, sizeof(float));  
  
 clock\_t beg = clock();  
  
 create\_matrix(A, 4.0f, 3.0f);  
 create\_matrix(I, 1.0f, 0.0f);  
  
 float a\_1 = find\_max\_column(A);  
 float a\_inf = find\_max\_str(A);  
  
 create\_B\_matrix(A, B, a\_1, a\_inf);  
  
 transposition(A, A\_t);  
 mult\_vectorize(B, A\_t, R);  
  
 for (int i = 0; i < N \* N; i++)  
 R[i] = I[i] - R[i];  
  
 for (int i = 0; i < N; i++)  
 for (int j = 0; j < N; j++)  
 cur\_2[i \* N + j] = R[i \* N + j];  
  
 transposition(R, R\_t);  
  
 // cur == R^(m-1), cur\_2 = R^m  
 for (int m = 0; m < M; m++) {  
 for (int i = 0; i < N; i++)  
 for (int j = 0; j < N; j++)  
 cur[i \* N + j] = cur\_2[i \* N + j];  
 summ\_vectorize(I, cur, I);  
 mult\_vectorize(cur, R\_t, cur\_2);  
 }  
  
 transposition(B, B\_t);  
 mult\_vectorize(I, B\_t, A\_1);  
  
 clock\_t end = clock();  
  
 transposition(A\_1, A\_1\_t);  
 mult\_vectorize(A, A\_1\_t, cur);  
  
 float trace = count\_trace(cur);  
  
 double time\_used = ((double) (end - beg)) / CLOCKS\_PER\_SEC;  
 printf("Time: %lf\n", time\_used);  
 printf("Trace: %f\n", trace);  
  
 //print\_matr(A\_1);  
  
 free(A); free(A\_1); free(A\_t);  
 free(A\_1\_t); free(B); free(B\_t);  
 free(I); free(R); free(R\_t);  
 free(cur); free(cur\_2);  
 return 0;  
}

Флаги Cmake

set(CMAKE\_C\_FLAGS "-g -O3 -msse3")

В данном способе, используя встроенные SIMD-функции компилятора, векторизовал умножения (с использованием скалярного произведения двух строк матрицы), а также сумму.

Для N=2048, M=10 и уровня оптимизации O3 время работы ~ 36 cек.



3. Использование BLAS

Добавим к Листингу 1 подключение заголовочного файла и заменим функцию mult

Листинг 3

#include "mkl.h"

void mult (const float \*A, const float \*B, float \*C) {  
 cblas\_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,  
 N, N, N, 1.0, A, N, B, N, 0.0, C, N);  
}

Функция компиляции icx matrix.c -qmkl -o matrix.out

Для N=2048, M=10: время работы ~ 2 cек, trace = 1,000606.

Получили, что t1(без векторизации)>t2(SIMD intrinsics)>t3(BLAS)

# ЗАКЛЮЧЕНИЕ

В ходе работы были изучены различные типы векторизации: ручная векторизация: встроенные функции компилятора, а так же использование оптимизированной библиотеки BLAS. Для каждого вида векторизации было измерено время вычисления обратной матрицы размером 2048х2048.

Вопросы:

*1) Какие части программы вы векторизовали? Как именно?*

1. Векторизовал сложение и умножение: сложение векторизовал с помощью функции \_mm\_add\_ps(), которая в цикле

for(int i = 0; i < N \* N / 4; ++i)  
 m128\_result[i] = \_mm\_add\_ps(m128\_left[i], m128\_right[i]);

выполняет сложение двух векторов длины N\*N, но за каждую итерацию цикла складывает сразу 4 ячейки.

Умножение векторизовал, используя скалярное умножение двух векторов: для этого перед умножением A\*B матрица B сначала транспонируется (чтобы умножать строку из А на строку из В) и затем i-ая строка из А скалярно умножается с j-ой строкой B и полученное значение записывается в result[i\*N+j]. Скалярное умножение взял из задания и подаю ему на вход две позиции в матрице (два начала нужных строк): A[i\*N] и B[j\*N]

*2) Какое ускорение теоретически может дать векторизация? Как оно соотносится с реальным? Почему?*

2) Векторизация ускоряет сложение в 4 раза, а также умножение в 4 раза (так как каждые две строки умножаются в 4 раза быстрее). Получается что суммарное ускорение работы программы не должно ускориться более чем в 4 раза. t1(без векторизации) = 103с., t2(SIMD intrinsics) = 36с; 103/36 ~2.87

Ускорение в 4 раза не получили, так как векторизованы только некоторые (наиболее часто встречающиеся) функции. Остальные части программы выполняются за такое же время, что и без векторизации.

*3) В каком порядке вы выполняли обход элементов матриц (последовательно/непоследовательно)? Одинаковый ли алгоритм умножения матриц в первой и второй программах? На что влияет порядок обхода элементов матрицы?*

3) Обход элементов выполнял последовательно. В ходе работы было замечено, что классическое умножение матрицы “строка на столбец” выполняется сильно дольше, чем умножение “строка на строку” (первая программа не закончила за 8 минут вычисления при классическом умножении). Поэтому перед умножением А\*В во обеих программах матрица В сначала транспонируется, чтобы умножать “строка на строку”.

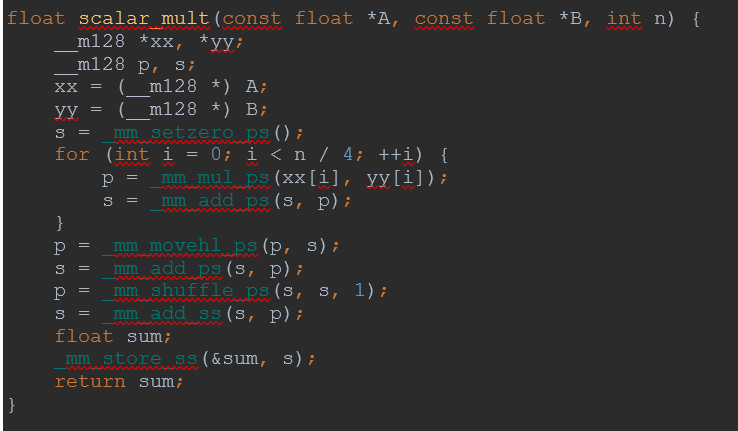
*Почему вариант умножения "строка на столбец" такой долгий?*

При умножении строка на столбец идет умножение A[i\*N+k]\*B[k\*N+j] (i - номер строки, j - номер столбца, k-индекс перебора по строке/столбцу). На следующей итерации будет A[i\*N+k+1]\*B[(k+1)\*N+j] то есть по массиву А идем последовательно, а по массивы В скачками в N элементов. При обращении к i\*N+k позиции массива, значение i\*N+k+1 может записаться в кэш, то есть ~~компилятор(не он~~) предугадывает, что возможно с последующей ячейкой тоже будут производиться какие-то операции. Но N элементов очевидно в кэш грузиться не будут и поэтому обращение к B[(k+1)\*N+j](при строка на столбец) будет сильно дольше, чем к B[j\*N+k+1] (при строка на строку)

P.S. : Во-первых, данные в кэш загружаются не по одному элементу, а сразу по несколько (если точнее - по размеру кэш-строки, что-то около нескольких десятков байтов)

Во-вторых, кэш-контроллер анализирует паттерны обращения в память, и выполняет предвыборку данных. Последовательный обход массива он распознает очень хорошо.

*Объясните подробно, как именно работает векторизованное скалярное произведение, что делает каждая из \_mm\* функций*



Сначала два поданных указателя на начало нужной строки приводятся к типу данных \_m128. Создается переменная p для хранения четырех произведений элементов векторов и s для накопления четырех частичных сумм произведений. С помощью \_mm\_setzero\_ps все четыре частичные суммы в s обнуляются. Затем каждая функция в цикле обрабатывает сразу 4 значения изначального вектора, а вектор берем длины N => цикл исполняется N/4 раз. \_mm\_mul\_ps выполняет четыре умножения и записывает в нужные ячейки p, т.е. p[0] = xx[4\*i+0]\*yy[4\*i+0], p[1] = xx[4\*i+1]\*yy[4\*i+1], p[2] = xx[4\*i+2]\*yy[4\*i+2], p[3] = xx[4\*i+3]\*yy[4\*i+3].

Затем полученные значения с помощью \_mm\_add\_ps прибавляются к соответствующим ячейкам s : s[0]+= p[0], s[1]+= p[1], s[2]+= p[2], s[3]+= p[3].

После цикла:

\_mm\_movehl\_ps(s,p) делает следующее : p[0]=s[2], p[1]=s[3], p[2]=p[2], p[3]=p[3].

Затем \_mm\_add\_ps(s,p) делает: s[0]+=s[2], s[1]+=s[3], s[2]+=p[2], s[3]+=p[3].

Затем \_mm\_shuffle\_ps(s,s,1) перемещает в r[0] значение s[1]

Затем \_mm\_add\_ss(s,p) делает: s[0] = s[0](==(s[0]+s[2]) из начального s) **+** r[0] (==(s[1]+s[3]) из начального s), s[1]=s[1], s[2]=s[2], s[3]=s[3].

Затем \_mm\_store\_ss записывает в summ значение s[0] (которое равно s[0]+s[1]+s[2]+s[3] из начального s).

Полученная сумма и будет скалярным произведением.