**ЦЕЛЬ**

1. Изучение SIMD-расширений архитектуры x86/x86-64.

2. Изучение способов использования SIMD-расширений в программах на языке Си.

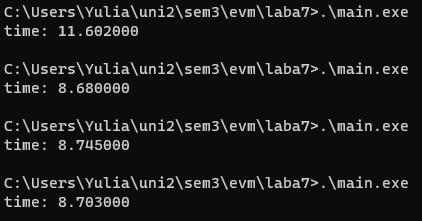
3. Получение навыков использования SIMD-расширений.

**ОПИСАНИЕ РАБОТЫ**

1. Без векторизации

Без векторизации как-то значительно оптимизировать программу сложно. Самая ёмкая операция — это перемножение матриц. Чтобы ее ускорить, можем улучшить кэш-локальность путем не перемножения матриц математическим способом (умножение строки на столбец), а проходить каждую матрицу построчно и постепенно накапливать суммы. Результат мы получим абсолютно тот же, однако, за счет того, что обе матрицы проходим последовательно, программа будет работать быстрее (причины этого лучше понятны после выполнения лабораторной работы номер 8).

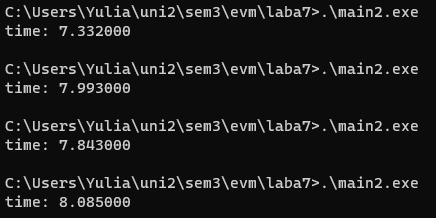
Время работы такой программы 8.68 секунд



1. Ручная векторизация

Сохраним принцип перемножения матриц таким же, в качестве векторов будем брать строки матрицы, перемножать их и постепенно накапливать сумму.

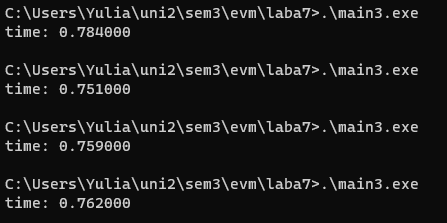
Время чуть ускорилось и теперь составляет 7.332 секунды



1. Библиотека BLAS

Для умножения матриц в данной библиотеке уже существует отдельная функция, поэтому нам остается только передать в нее аргументы.

Результат замера времени значительно отличается от предыдущих и составляет 0.751



***Приложение 1. Без векторизации***

#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
  
#define **N** 2048  
#define **M** 10  
  
float\* transpose\_matrix (float\* B) {  
 float\* transposed\_B = (float\*) calloc(**N**\***N**, sizeof(float));  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 transposed\_B[**N** \* j + i] = B[**N** \* i + j];  
 }  
 }  
 return transposed\_B;  
}  
  
void fill\_I(float\* I) {  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 if (i == j) {  
 I[**N** \* i + j] = 1;  
 }  
 }  
 }  
}  
  
void create\_matrix(float\* A) {  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 A[**N** \* i + j] = rand() % 2;  
 }  
 }  
}  
  
float column\_max(const float\* A) {  
 float max = 0;  
 for (int i = 0; i < **N**; i++) {  
 float cnt = 0;  
 for (int j = 0; j < **N**; j++) {  
 cnt += A[**N** \* j + i];  
 }  
 if (cnt > max) {  
 max = cnt;  
 }  
 }  
 return max;  
}  
  
float row\_max(const float\* A) {  
 float max = 0;  
 for (int i = 0; i < **N**; i++) {  
 float cnt = 0;  
 for (int j = 0; j < **N**; j++) {  
 cnt += A[**N** \* i + j];  
 }  
 if (cnt > max) {  
 max = cnt;  
 }  
 }  
 return max;  
}  
  
float find\_max(float\* A) {  
 return column\_max(A) \* row\_max(A);  
}  
  
void find\_B(float\* B, const float\* transposed\_A, float max) {  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 B[**N** \* i + j] = transposed\_A[**N** \* i + j] / max;  
 }  
 }  
}  
  
  
void multiply\_matricies(const float\* A, const float\* B, float\* Result) {  
 for (int i = 0; i < **N**; i++) {  
 for (int k = 0; k < **N**; k++) {  
 for (int j = 0; j < **N**; j++) {  
 Result[**N** \* i + j] += A[**N** \* i + k] \* B[**N** \* k + j];  
 }  
 }  
 }  
}  
  
void find\_R(float\* R, const float\* A, const float\* B, const float\* I) {  
 float\* new\_mult = (float\*) calloc(**N**\***N**, sizeof(float));  
 multiply\_matricies(B, A, new\_mult);  
  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 R[**N** \* i + j] = I[**N** \* i + j] - new\_mult[**N** \* i + j];  
 }  
 }  
}  
  
void sum(float\* Result, float\* R, float\* previous) {  
 float\* new\_mult = (float\*) calloc(**N**\***N**, sizeof(float));  
 multiply\_matricies(R,previous, new\_mult);  
  
 for (int i = 0; i < **N**; ++i) {  
 for (int j = 0; j < **N**; ++j) {  
 previous[**N** \* i + j] = new\_mult[**N** \* i + j];  
 }  
 }  
  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 Result[**N** \* i + j] += new\_mult[**N** \* i + j];  
 }  
 }  
}  
  
int main() {  
 clock\_t start, end;  
  
 float\* I = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 float\* A = (float\*) calloc(**N**\***N**, sizeof(float));  
 float\* transposed\_A = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 float\* B = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 float\* R = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 float\* inverted\_A = (float\*) calloc(**N**\***N**, sizeof(float));  
  
  
 fill\_I(I);  
 create\_matrix(A);  
 transposed\_A = transpose\_matrix(A);  
  
 float max = find\_max(A);  
 find\_B(B, transposed\_A, max);  
 find\_R(R, A, B, I);  
  
 float\* Result = I;  
 float\* previous = (float\*) calloc(**N**\***N**, sizeof(float));  
 fill\_I(previous);  
 start = clock();  
 for (int i = 1; i < **M**; i++) {  
 sum(Result, R, previous);  
 }  
  
 multiply\_matricies(Result, B, inverted\_A);  
  
 end = clock();  
  
 float cpu\_time\_used = ((float) (end - start)) / **CLOCKS\_PER\_SEC**;  
 printf("time: %lf\n", cpu\_time\_used);  
  
 float\* help = (float\*) calloc(**N**\***N**, sizeof(float));  
 multiply\_matricies(A, inverted\_A, help);  
  
  
  
 free(A);  
 free(B);  
 free(transposed\_A);  
 free(R);  
 free(Result);  
}

***Приложение 2. Ручная векторизация***

#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
#include <immintrin.h>  
  
#define **N** 64  
#define **M** 12000  
  
void fill\_I(float\* I) {  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 if (i == j) {  
 I[**N** \* i + j] = 1;  
 }  
 }  
 }  
}  
  
void create\_matrix(float\* A, float\* transposed\_A) {  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 A[**N** \* i + j] = rand() % 10;  
 transposed\_A[**N** \* j + i] = A[**N** \* i + j];  
 }  
 }  
}  
  
float column\_max(const float\* A) {  
 float max = 0;  
 for (int i = 0; i < **N**; i++) {  
 float cnt = 0;  
 for (int j = 0; j < **N**; j++) {  
 cnt += A[**N** \* j + i];  
 }  
 if (cnt > max) {  
 max = cnt;  
 }  
 }  
 return max;  
}  
  
float row\_max(const float\* A) {  
 float max = 0;  
 for (int i = 0; i < **N**; i++) {  
 float cnt = 0;  
 for (int j = 0; j < **N**; j++) {  
 cnt += A[**N** \* i + j];  
 }  
 if (cnt > max) {  
 max = cnt;  
 }  
 }  
 return max;  
}  
  
float find\_max(float\* A) {  
 return column\_max(A) \* row\_max(A);  
}  
  
void find\_B(float\* B, const float\* transposed\_A, float max) {  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 B[**N** \* i + j] = transposed\_A[**N** \* i + j] / max;  
 }  
 }  
}  
  
void transpose\_matrix (float\* B) {  
 float\* transposed\_B = (float\*) calloc(**N**\***N**, sizeof(float));  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 transposed\_B[**N** \* j + i] = B[**N** \* i + j];  
 }  
 }  
 B = transposed\_B;  
}  
  
float inner2(float\* x, float\* y) {\_\_m256 p, s;  
  
 s = \_mm256\_set1\_ps(0);  
  
 for (int i = 0; i <= (**N** - 1) / 8; i++) {  
 \_\_m256 xx = \_mm256\_loadu\_ps(x + 8 \* i);  
 \_\_m256 yy = \_mm256\_loadu\_ps(y + 8 \* i);  
 p = \_mm256\_mul\_ps(xx, yy);  
 s = \_mm256\_add\_ps(s, p);  
 }  
  
 p = **\_mm256\_permute2f128\_ps**(s, p, 1);  
 s = \_mm256\_add\_ps(s, p);  
 p = **\_mm256\_shuffle\_ps**(s, s, 14);  
 s = \_mm256\_add\_ps(s, p);  
 p = **\_mm256\_shuffle\_ps**(s, s, 1);  
 s = \_mm256\_add\_ps(s, p);  
  
 float sum;  
 \_mm256\_storeu\_ps(&sum, s);  
 return sum;  
}  
  
void multiply\_matrices(float\* A, float\* B, float\* Result) {  
 transpose\_matrix(B);  
  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 Result[**N** \* i + j] += inner2(A + (i \* **N**), B + (j \* **N**));  
 }  
 }  
  
 transpose\_matrix(B);  
}  
  
void find\_R(float\* R, float\* A, float\* B, float\* I) {  
 float\* new\_mult = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 multiply\_matrices(B, A, new\_mult); *//BA* for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 R[**N** \* i + j] = I[**N** \* i + j] - new\_mult[**N** \* i + j]; *//I - BA* }  
 }  
}  
  
void sum(float\* Result, float\* R, float\* previous) {  
 float\* new\_mult = (float\*) calloc(**N**\***N**, sizeof(float));  
 multiply\_matrices(R,previous, new\_mult);  
  
 for (int i = 0; i < **N**; ++i) {  
 for (int j = 0; j < **N**; ++j) {  
 previous[**N** \* i + j] = new\_mult[**N** \* i + j];  
 }  
 }  
  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 Result[**N** \* i + j] += new\_mult[**N** \* i + j];  
 }  
 }  
}  
  
int main() {  
 clock\_t start, end;  
  
 float\* I = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 float\* A = (float\*) calloc(**N**\***N**, sizeof(float));  
 float\* transposed\_A = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 float\* B = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 float\* R = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 float\* inverted\_A = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 fill\_I(I);  
 create\_matrix(A, transposed\_A);  
  
 float max = find\_max(A);  
 find\_B(B, transposed\_A, max);  
 find\_R(R, A, B, I);  
  
 float\* Result = I;  
 float\* previous = (float\*) calloc(**N**\***N**, sizeof(float));  
 fill\_I(previous);  
  
 start = clock();  
  
 for (int i = 1; i < **M**; i++) {  
 sum(Result, R, previous);  
 }  
 multiply\_matrices(Result, B, inverted\_A);  
  
 end = clock();  
  
 float cpu\_time\_used = ((float) (end - start)) / **CLOCKS\_PER\_SEC**;  
 printf("time: %lf\n", cpu\_time\_used);  
  
 multiply\_matrices(A, inverted\_A, Result);  
 printf("%f ", Result[0]);  
  
 free(A);  
 free(B);  
 free(transposed\_A);  
 free(R);  
 free(Result);  
}

***Приложение 3. BLAS***

#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
#include <cblas.h>  
  
#define **N** 2048  
#define **M** 10  
  
void fill\_I(float\* IdentityMatrix) {  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 if (i == j) {  
 IdentityMatrix[**N** \* i + j] = 1;  
 }  
 }  
 }  
}  
  
void create\_matrix(float\* A, float\* transposed\_A) {  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 A[**N** \* i + j] = rand() % 10;  
 transposed\_A[**N** \* j + i] = A[**N** \* i + j];  
 }  
 }  
}  
  
float column\_max(const float\* A) {  
 float max = 0;  
 for (int i = 0; i < **N**; i++) {  
 float cnt = 0;  
 for (int j = 0; j < **N**; j++) {  
 cnt += A[**N** \* j + i];  
 }  
 if (cnt > max) {  
 max = cnt;  
 }  
 }  
 return max;  
}  
  
float row\_max(const float\* A) {  
 float max = 0;  
 for (int i = 0; i < **N**; i++) {  
 float cnt = 0;  
 for (int j = 0; j < **N**; j++) {  
 cnt += A[**N** \* i + j];  
 }  
 if (cnt > max) {  
 max = cnt;  
 }  
 }  
 return max;  
}  
  
float find\_max(float\* A) {  
 return column\_max(A) \* row\_max(A);  
}  
  
void find\_B(float\* B, const float\* transposed\_A, float max) {  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 B[**N** \* i + j] = transposed\_A[**N** \* i + j] / max;  
 }  
 }  
}  
  
void transpose\_matrix (float\* B) {  
 float\* transposed\_B = (float\*) calloc(**N**\***N**, sizeof(float));  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 transposed\_B[**N** \* j + i] = B[**N** \* i + j];  
 }  
 }  
 B = transposed\_B;  
}  
  
void multiply\_matricies(float\* A, float\* B, float\* Result) {  
 cblas\_sgemm(*CblasRowMajor*,*CblasNoTrans*,*CblasNoTrans*, **N**, **N**, **N**, 1.0, A, **N**, B, **N**, 0.0, Result, **N**);  
}  
  
void find\_R(float\* R, float\* A, float\* B, float\* IdentityMatrix) {  
 float\* new\_mult = (float\*) calloc(**N**\***N**, sizeof(float));  
 multiply\_matricies(B, A, new\_mult);  
  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 R[**N** \* i + j] = IdentityMatrix[**N** \* i + j] - new\_mult[**N** \* i + j];  
 }  
 }  
}  
  
void sum(float\* Result, float\* R, float\* previous) {  
 float\* new\_mult = (float\*) calloc(**N**\***N**, sizeof(float));  
 multiply\_matricies(R,previous, new\_mult);  
  
 for (int i = 0; i < **N**; ++i) {  
 for (int j = 0; j < **N**; ++j) {  
 previous[**N** \* i + j] = new\_mult[**N** \* i + j];  
 }  
 }  
  
 for (int i = 0; i < **N**; i++) {  
 for (int j = 0; j < **N**; j++) {  
 Result[**N** \* i + j] += new\_mult[**N** \* i + j];  
 }  
 }  
}  
  
int main() {  
 clock\_t start, end;  
  
 float\* IdentityMatrix = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 float\* A = (float\*) calloc(**N**\***N**, sizeof(float));  
 float\* transposed\_A = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 float\* B = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 float\* R = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 float\* inverted\_A = (float\*) calloc(**N**\***N**, sizeof(float));  
  
 fill\_I(IdentityMatrix);  
 create\_matrix(A, transposed\_A);  
  
 float max = find\_max(A);  
 find\_B(B, transposed\_A, max);  
 find\_R(R, A, B, IdentityMatrix);  
  
 float\* Result = IdentityMatrix;  
 float\* previous = (float\*) calloc(**N**\***N**, sizeof(float));  
 fill\_I(previous);  
  
 start = clock();  
 for (int i = 1; i < **M**; i++) {  
 sum(Result, R, previous);  
 }  
 multiply\_matricies(Result, B, inverted\_A);  
  
 end = clock();  
  
 float cpu\_time\_used = ((float) (end - start)) / **CLOCKS\_PER\_SEC**;  
 printf("time: %lf\n", cpu\_time\_used);  
  
  
 free(A);  
 free(B);  
 free(transposed\_A);  
 free(R);  
 free(Result);  
}