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

```
C:\Users\Yulia\uni2\sem3\evm\laba7>.\main.exe
time: 11.602000

C:\Users\Yulia\uni2\sem3\evm\laba7>.\main.exe
time: 8.680000

C:\Users\Yulia\uni2\sem3\evm\laba7>.\main.exe
time: 8.745000

C:\Users\Yulia\uni2\sem3\evm\laba7>.\main.exe
time: 8.703000
```

2. 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

```
C:\Users\Yulia\uni2\sem3\evm\laba7>.\main2.exe
time: 7.332000

C:\Users\Yulia\uni2\sem3\evm\laba7>.\main2.exe
time: 7.993000

C:\Users\Yulia\uni2\sem3\evm\laba7>.\main2.exe
time: 7.843000

C:\Users\Yulia\uni2\sem3\evm\laba7>.\main2.exe
time: 8.085000
```

3. 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

```
C:\Users\Yulia\uni2\sem3\evm\laba7>.\main3.exe
time: 0.784000

C:\Users\Yulia\uni2\sem3\evm\laba7>.\main3.exe
time: 0.751000

C:\Users\Yulia\uni2\sem3\evm\laba7>.\main3.exe
time: 0.759000

C:\Users\Yulia\uni2\sem3\evm\laba7>.\main3.exe
time: 0.762000
```

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);
}
```

```
#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);
}
```