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, 23209 группы
Инокова Семёна Шухратовича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
А. Ю. Кудинов

Новосибирск 2024

СОДЕРЖАНИЕ

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.

ЗАДАНИЕ

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. Каждый вариант программы оптимизировать по скорости, насколько это возможно.

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

ОПИСАНИЕ РАБОТЫ

1. На языке Си написал 3 программы, реализующие алгоритм обращения матрицы с помощью разложения в ряд:

- вариант без ручной векторизации,
- вариант с ручной векторизацией (выбрал расширение AVX для векторизации, так как он содержит 256-битные регистры, подходящие для моих операций над элементами массива)
- вариант с матричными операциями, выполненными с использованием оптимизированной библиотеки BLAS.

2. Программы скомпилировал и измерил время работы при входных данных N=2048, M=10.

3. Сравнил время работы программ и проанализировал результат.

ЗАКЛЮЧЕНИЕ

В ходе данной работы я познакомился с SIMD-расширениями архитектуры x86/x86-64, научился использовать их в коде. Также я изучил оптимизированную библиотеку линейной алгебры BLAS.
В моём случае ручшая векторизация уменьшает время работы программы в 3,5 раза, а готовая библиотека BLAS в несколько десятков раз.

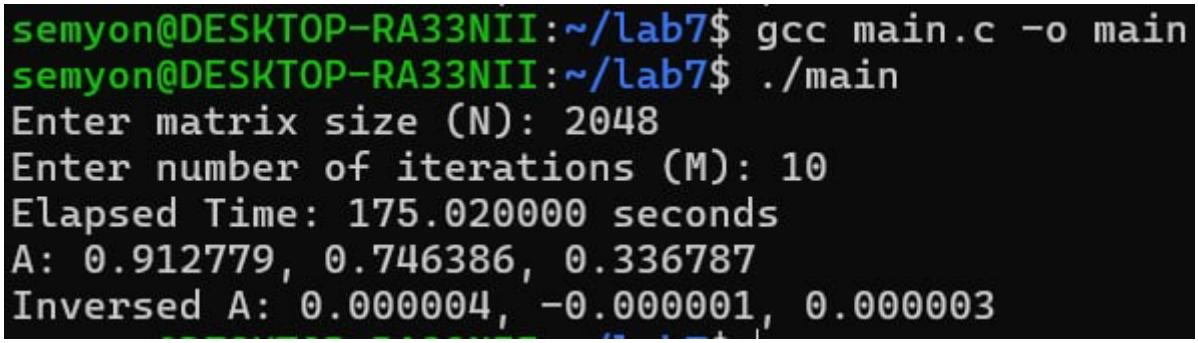
Приложение 1. *Код программы без векторизации и время работы программы*

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sys/times.h>
#include <unistd.h>
#include <string.h>

float *create_identity_matrix(size_t N)
{
   float *Im = calloc(N * N, sizeof(float));
   if (!Im) return NULL;

   for (size_t i = 0; i < N; ++i)
      Im[i * N + i] = 1.0f;

   return Im;
}

float *generate_B(const float *A, size_t N)
{
   float max_row_sum = __FLT_MIN__;
   float max_col_sum = __FLT_MIN__;

   float *row_sums = calloc(N, sizeof(float));
   float *col_sums = calloc(N, sizeof(float));
   if (!row_sums || !col_sums) {
      free(row_sums);
      free(col_sums);
      return NULL;
   }

   for (size_t i = 0; i < N; ++i) {
      for (size_t j = 0; j < N; ++j) {
         row_sums[i] += A[i * N + j];
         col_sums[j] += A[i * N + j];
      }
   }

   for (size_t i = 0; i < N; ++i) {
      if (row_sums[i] > max_row_sum) max_row_sum = row_sums[i];
      if (col_sums[i] > max_col_sum) max_col_sum = col_sums[i];
   }

   free(row_sums);
   free(col_sums);

   float *B = calloc(N * N, sizeof(float));
   if (!B) return NULL;
```

```
    float scaling_factor = max_row_sum * max_col_sum;
    if (scaling_factor == 0) return NULL;

    for (size_t i = 0; i < N; ++i)
        for (size_t j = 0; j < N; ++j)
            B[i * N + j] = A[j * N + i] / scaling_factor;

    return B;
}

float *create_random_matrix(size_t N)
{
    float *M = calloc(N * N, sizeof(float));
    if (!M) return NULL;

    for (size_t i = 0; i < N * N; ++i)
        M[i] = rand() / (float)RAND_MAX;

    return M;
}

void matrix_multiply(const float *A, const float *B, float *C, size_t N)
{
    for (size_t i = 0; i < N; ++i)
        for (size_t k = 0; k < N; ++k) {
            float a = A[i * N + k];
            for (size_t j = 0; j < N; ++j)
                C[i * N + j] += a * B[k * N + j];
        }
}

void matrix_subtract(const float *A, const float *B, float *C, size_t N)
{
    for (size_t i = 0; i < N * N; ++i)
        C[i] = A[i] - B[i];
}

void matrix_add(const float *A, const float *B, float *C, size_t N)
{
    for (size_t i = 0; i < N * N; ++i)
        C[i] = A[i] + B[i];
}

void matrix_invert(const float *A, float *result, size_t N, size_t M)
{
    float *Im = create_identity_matrix(N);
    float *B = generate_B(A, N);

    if (!Im || !B) return;

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

```
    float *current_power = calloc(N * N, sizeof(float));
    float *temp_result = calloc(N * N, sizeof(float));

    if (!R || !current_power || !temp_result) {
        free(Im); free(B); free(R); free(current_power); free(temp_result);
        return;
    }

    float *BA = calloc(N * N, sizeof(float));
    matrix_multiply(B, A, BA, N);
    matrix_subtract(Im, BA, R, N);

    memcpy(current_power, R, N * N * sizeof(float));
    matrix_add(Im, R, result, N);

    for (size_t i = 2; i <= M; ++i) {
        matrix_multiply(current_power, R, temp_result, N);
        matrix_add(result, temp_result, result, N);
        memcpy(current_power, temp_result, N * N * sizeof(float));
        memset(temp_result, 0, N * N * sizeof(float));
    }

    matrix_multiply(result, B, temp_result, N);
    memcpy(result, temp_result, N * N * sizeof(float));

    free(Im); free(B); free(R); free(BA); free(current_power); free(temp_result);
}

int main(void)
{
    size_t N = 0, M = 0;
    printf("Enter matrix size (N): ");
    if (scanf("%zu", &N) == 0) return 0;
    printf("Enter number of iterations (M): ");
    if (scanf("%zu", &M) == 0) return 0;

    srand(time(NULL));
    float *A = create_random_matrix(N);
    float *inverseA = calloc(N * N, sizeof(float));

    if (!A || !inverseA) return 1;

    struct tms start, end;
    clock_t clock_start = times(&start);

    matrix_invert(A, inverseA, N, M);

    clock_t clock_end = times(&end);

    double elapsed_time = (double)(end.tms_utime - start.tms_utime) / sysconf(_SC_CLK_TCK);

    printf("Elapsed Time: %lf seconds\n", elapsed_time);
```

```
    printf("A: %f, %f, %f\n", A[0], A[1], A[N]);

    printf("Inverse A: %f, %f, %f\n", inverseA[0], inverseA[1], inverseA[N]);

    free(A);
    free(inverseA);
    return 0;
}
```



Приложение 2. *Код программы с ручной векторизацией и время работы программы*

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sys/times.h>
#include <unistd.h>
#include <string.h>
#include <immintrin.h>

float *create_identity_matrix(size_t N)
{
   float *Im = calloc(N * N, sizeof(float));
   if (!Im) return NULL;

   for (size_t i = 0; i < N; ++i)
      Im[i * N + i] = 1.0f;

   return Im;
}

float *generate_B(const float *A, size_t N)
{
   float max_row_sum = __FLT_MIN__;
   float max_col_sum = __FLT_MIN__;

   float *row_sums = calloc(N, sizeof(float));
   float *col_sums = calloc(N, sizeof(float));
   if (!row_sums || !col_sums) {
      free(row_sums);
      free(col_sums);
      return NULL;
   }

   for (size_t i = 0; i < N; ++i) {
      for (size_t j = 0; j < N; ++j) {
         row_sums[i] += A[i * N + j];
         col_sums[j] += A[i * N + j];
      }
   }

   for (size_t i = 0; i < N; ++i) {
      if (row_sums[i] > max_row_sum) max_row_sum = row_sums[i];
      if (col_sums[i] > max_col_sum) max_col_sum = col_sums[i];
   }

   free(row_sums);
   free(col_sums);

   float *B = calloc(N * N, sizeof(float));
   if (!B) return NULL;
```

```
    float scaling_factor = max_row_sum * max_col_sum;
    if (scaling_factor == 0) return NULL;

    for (size_t i = 0; i < N; ++i)
        for (size_t j = 0; j < N; ++j)
            B[i * N + j] = A[j * N + i] / scaling_factor;

    return B;
}

float *create_random_matrix(size_t N)
{
    float *M = calloc(N * N, sizeof(float));
    if (!M) return NULL;

    for (size_t i = 0; i < N * N; ++i)
        M[i] = rand() / (float)RAND_MAX;

    return M;
}

void matrix_multiply(const float *A, const float *B, float *C, size_t N)
{
    float *B_T = calloc(N * N, sizeof(float));
    for (size_t i = 0; i < N; ++i)
        for (size_t j = 0; j < N; ++j)
            B_T[j * N + i] = B[i * N + j];

    for (size_t i = 0; i < N; ++i) {
        for (size_t j = 0; j < N; ++j) {
            __m256 c_sum = _mm256_setzero_ps();

            size_t k = 0;
            for (; k + 7 < N; k += 8) {
                __m256 a_vec = _mm256_loadu_ps(&A[i * N + k]);
                __m256 b_vec = _mm256_loadu_ps(&B_T[j * N + k]);
                c_sum = _mm256_fmadd_ps(a_vec, b_vec, c_sum);
            }

            float temp[8];
            _mm256_storeu_ps(temp, c_sum);
            __m256 shuf = _mm256_permute2f128_ps(c_sum, c_sum, 1);
            __m256 sums = _mm256_add_ps(c_sum, shuf);
            sums = _mm256_hadd_ps(sums, sums);
            sums = _mm256_hadd_ps(sums, sums);
            float total = _mm_cvtss_f32(_mm256_castps256_ps128(sums));

            for (; k < N; ++k) {
                total += A[i * N + k] * B_T[j * N + k];
            }
```

```
            C[i * N + j] = total;
        }
    }

    free(B_T);
}

void matrix_subtract(const float *A, const float *B, float *C, size_t N)
{
    for (size_t i = 0; i < N * N; ++i)
        C[i] = A[i] - B[i];
}

void matrix_add(const float *A, const float *B, float *C, size_t N)
{
    for (size_t i = 0; i < N * N; ++i)
        C[i] = A[i] + B[i];
}

void matrix_invert(const float *A, float *result, size_t N, size_t M)
{
    float *Im = create_identity_matrix(N);
    float *B = generate_B(A, N);

    if (!Im || !B) return;

    float *R = calloc(N * N, sizeof(float));
    float *current_power = calloc(N * N, sizeof(float));
    float *temp_result = calloc(N * N, sizeof(float));

    if (!R || !current_power || !temp_result) {
        free(Im); free(B); free(R); free(current_power); free(temp_result);
        return;
    }

    float *BA = calloc(N * N, sizeof(float));
    matrix_multiply(B, A, BA, N);
    matrix_subtract(Im, BA, R, N);

    memcpy(current_power, R, N * N * sizeof(float));
    matrix_add(Im, R, result, N);

    for (size_t i = 2; i <= M; ++i) {
        matrix_multiply(current_power, R, temp_result, N);
        matrix_add(result, temp_result, result, N);
        memcpy(current_power, temp_result, N * N * sizeof(float));
        memset(temp_result, 0, N * N * sizeof(float));
    }

    matrix_multiply(result, B, temp_result, N);
    memcpy(result, temp_result, N * N * sizeof(float));
```

```
    free(Im); free(B); free(R); free(BA); free(current_power); free(temp_result);
}

int main(void)
{
    size_t N = 0, M = 0;
    printf("Enter matrix size (N): ");
    if (scanf("%zu", &N) == 0) return 0;
    printf("Enter number of iterations (M): ");
    if (scanf("%zu", &M) == 0) return 0;

    srand(time(NULL));
    float *A = create_random_matrix(N);
    float *inverseA = calloc(N * N, sizeof(float));

    if (!A || !inverseA) return 1;

    struct tms start, end;
    clock_t clock_start = times(&start);

    matrix_invert(A, inverseA, N, M);

    clock_t clock_end = times(&end);
    double elapsed_time = (double)(end.tms_utime - start.tms_utime) / sysconf(_SC_CLK_TCK);

    printf("Elapsed Time: %lf seconds\n", elapsed_time);

    printf("A: %f, %f, %f\n", A[0], A[1], A[N]);

    printf("Inverse A: %f, %f, %f\n", inverseA[0], inverseA[1], inverseA[N]);

    free(A);
    free(inverseA);
    return 0;
}
```

```
semyon@DESKTOP-RA33NII:~/lab7$ gcc -mfma -mavx mainexp.c -o mainexp
semyon@DESKTOP-RA33NII:~/lab7$ ./mainexp
Enter matrix size (N): 2048
Enter number of iterations (M): 10
Elapsed Time: 53.370000 seconds
A: 0.240374, 0.417725, 0.466910
Inverse A: -0.000002, -0.000000, -0.000000
```

Приложение 3. *Код программы с использованием BLAS и время работы программы*

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include <sys/times.h>
#include <cblas.h>
#include <stddef.h>

float *create_identity_matrix(size_t N);
void matrix_invert(const float *A, float *result, size_t N, size_t M);
void print_matrix(const float *matrix, size_t N);

float *create_random_matrix(size_t N)
{
   float *M = calloc(N * N, sizeof(float));
   if (!M) return NULL;

   for (size_t i = 0; i < N * N; ++i)
      M[i] = rand() / (float)RAND_MAX;

   return M;
}

float *create_identity_matrix(size_t N) {
   float *Imatrix = (float *)calloc(N * N, sizeof(float));
   if (!Imatrix) return NULL;

   for (size_t i = 0; i < N; ++i)
      Imatrix[i * N + i] = 1.0f;

   return Imatrix;
}

float *generate_B(const float *A, size_t N)
{
   float max_row_sum = __FLT_MIN__;
   float max_col_sum = __FLT_MIN__;

   float *row_sums = calloc(N, sizeof(float));
   float *col_sums = calloc(N, sizeof(float));
   if (!row_sums || !col_sums) {
      free(row_sums);
      free(col_sums);
      return NULL;
   }

   for (size_t i = 0; i < N; ++i) {
      for (size_t j = 0; j < N; ++j) {
```

```
            row_sums[i] += A[i * N + j];
            col_sums[j] += A[i * N + j];
        }
    }

    for (size_t i = 0; i < N; ++i) {
        if (row_sums[i] > max_row_sum) max_row_sum = row_sums[i];
        if (col_sums[i] > max_col_sum) max_col_sum = col_sums[i];
    }

    free(row_sums);
    free(col_sums);

    float *B = calloc(N * N, sizeof(float));
    if (!B) return NULL;

    float scaling_factor = max_row_sum * max_col_sum;
    if (scaling_factor == 0) return NULL;

    for (size_t i = 0; i < N; ++i)
        for (size_t j = 0; j < N; ++j)
            B[i * N + j] = A[j * N + i] / scaling_factor;

    return B;
}

void matrix_multiply(const float *A, const float *B, float *C, size_t N) {
    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1.0f, A, N, B, N, 0.0f,
C, N);
}

void matrix_subtract(const float *A, const float *B, float *C, size_t N) {
    for (size_t i = 0; i < N * N; ++i)
        C[i] = A[i] - B[i];
}

void matrix_add(const float *A, const float *B, float *C, size_t N) {
    for (size_t i = 0; i < N * N; ++i)
        C[i] = A[i] + B[i];
}

void matrix_invert(const float *A, float *result, size_t N, size_t M)
{
    float *Im = create_identity_matrix(N);
    float *B = generate_B(A, N);

    if (!Im || !B) return;

    float *R = calloc(N * N, sizeof(float));
    float *current_power = calloc(N * N, sizeof(float));
    float *temp_result = calloc(N * N, sizeof(float));
```

```
    if (!R || !current_power || !temp_result) {
        free(Im); free(B); free(R); free(current_power); free(temp_result);
        return;
    }

    float *BA = calloc(N * N, sizeof(float));
    matrix_multiply(B, A, BA, N);
    matrix_subtract(Im, BA, R, N);

    memcpy(current_power, R, N * N * sizeof(float));
    matrix_add(Im, R, result, N);

    for (size_t i = 2; i <= M; ++i) {
        matrix_multiply(current_power, R, temp_result, N);
        matrix_add(result, temp_result, result, N);
        memcpy(current_power, temp_result, N * N * sizeof(float));
        memset(temp_result, 0, N * N * sizeof(float));
    }

    matrix_multiply(result, B, temp_result, N);
    memcpy(result, temp_result, N * N * sizeof(float));

    free(Im); free(B); free(R); free(BA); free(current_power); free(temp_result);
}


int main(void) {
    size_t N = 0, M = 0;
    printf("Enter matrix size (N): ");
    if (scanf("%zu", &N) == 0) return 0;
    printf("Enter number of iterations (M): ");
    if (scanf("%zu", &M) == 0) return 0;

    srand(time(NULL));
    float *A = create_random_matrix(N);
    float *inverseA = (float *)calloc(N * N, sizeof(float));

    if (!A || !inverseA) return 1;

    struct tms start, end;
    clock_t clock_start = times(&start);

    matrix_invert(A, inverseA, N, M);

    clock_t clock_end = times(&end);

    double elapsed_time = (double)(end.tms_utime - start.tms_utime) / sysconf(_SC_CLK_TCK);

    printf("Elapsed Time: %lf seconds\n", elapsed_time);

    printf("A: %f, %f, %f\n", A[0], A[1], A[N]);
```

```
    printf("Inverse A: %f, %f, %f\n", inverseA[0], inverseA[1], inverseA[N]);

    free(A);
    free(inverseA);
    return 0;
}
```

```
semyon@DESKTOP-RA33NII:~/lab7$ gcc mainBLAS2.c -o mainBLAS2 -lopenblas -lm
semyon@DESKTOP-RA33NII:~/lab7$ ./mainBLAS2
Enter matrix size (N): 2048
Enter number of iterations (M): 10
Elapsed Time: 4.780000 seconds
A: 0.432950, 0.777437, 0.556080
Inverse A: -0.000000, 0.000001, 0.000003
```