МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №2

по курсу «Эффективное программирование современных микропроцессоров и мультипроцессоров»

(Вариант №3)

Выполнил: студент 3-го курса гр. 17208

Гафиятуллин А.Р

Новосибирск, 2020

1. **ЦЕЛИ РАБОТЫ:**

   Научиться векторизовать простые программы численного моделирования.

   Вариант №3: *решение уравнения Пуассона методом Якоби на float-ах.* Алгоритм моделирует установление стационарного распределение тепла в пластинке с заданным распределением источников и стоков тепла. В начальный момент времени значения искомой функции на сетке инициализируются нулями. На каждом шаге моделирования значения искомой функции пересчитываются по заданной формуле.

2. **ХОД РАБОТЫ:**

   2.1.Параметры тестирования:

   2.1.1. Тестирование происходило на процессоре **Intel(R) Core(TM) i7-9700F CPU @ 3.00GHz (CPU max MHz: 4700.0000 (Turbo Boost))**.

   2.1.2. Компилятор: **icc (ICC) 19.1.0.166 20191121;**

   2.1.3. Ключи компиляции: **-O2 -ip -xcoffeelake -axcoffeelake;**

   2.1.4. Параметры программы: $\mathbf{N_x = N_y = 9000}$, $\mathbf{N_t = 110}$.

   2.2.Программа, векторизованная с помощью компилятора**:**

   2.2.1. Текст программы: **(см. приложение 4.1)**;

   2.2.2. Отчет компилятора о векторизации:

   
   auto-vectorization-report.log

   2.2.3. Производительность:

   2.2.3.1. Число инструкций на такт: 1.17 (в 2.57 раз хуже скалярной версии). Но векторизованной программе это простительно, так как за раз обрабатывается сразу 8 значений типа *float;*

   2.2.3.2. Процент кэш-промахов для кэша 3 уровня: 22.26% (в 2 раза лучше скалярной версии). Но общее количество обращений на load в LLC увеличилось в 4.45 раза.

   2.2.3.3. Процент кэш-промахов для кэша-данных 1 уровня: 24.94% (в 4.5 раза хуже скалярной версии). Но общее количество обращений на load в L1-dcache уменьшилось в 4.49 раза.

   По всей видимости, сразу много 256-байтных векторов не помещается в L1-кэш, поэтому произошел перекос в сторону количества обращений к LLC-кэшу.

2.2.3.4. Процент неправильно предсказанных переходов: 0.13% (в 4.3 раза хуже скалярной версии). Но количество ветвлений уменьшилось в 7.06 раз. Не критично для векторизованной программы.

2.2.3.5. Время работы: 5.82 сек. (в 3.26 раз лучше скалярной версии).

2.3. Ручная векторизация программы:

2.3.1. Итоговый текст программы: **(см. приложение 4.2)**;

2.3.2. Этапы векторизации:

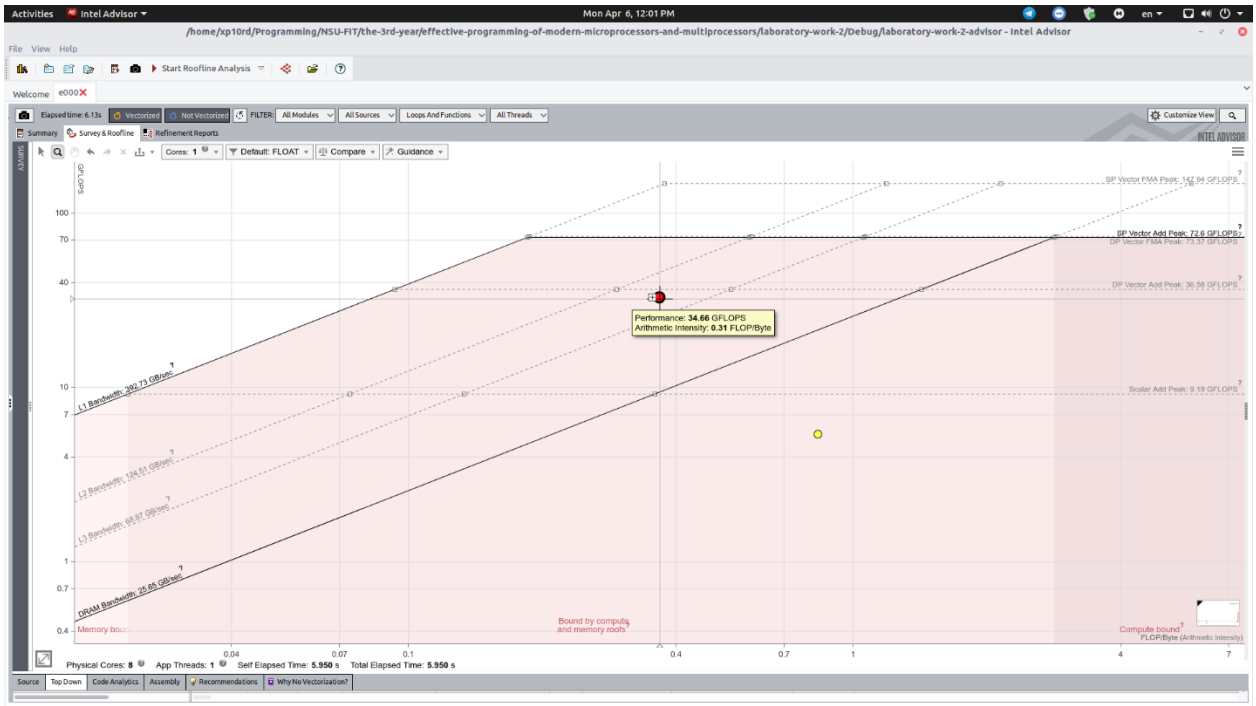
Этап, тип	**Время, сек.**
AVX2	6.41
AVX2 + FMA	6.66

2.4. Производительность:

2.4.1. Все характеристики оказались примерно равными характеристикам авто-векторизованной версии, кроме бОльшего количества кэш-промахов LLC (39.42%).

2.4.2. Видимо из-за этих промахов программа с ручной векторизацией оказалась медленнее авто-векторизованной версии.

2.5. Roofline-модель с точкой, соответствующей основному циклу программы (красная точка посередине) для наиболее быстрого варианта векторизованной программы (авто-векторизованная версия):



Программа стала менее ограниченной по скорости памяти (точка сместилась выше), но осталась в такой же степени ограничена по вычислениям, как и скалярная версия. **Арифметическая интенсивность выросла.**

3. ВЫВОДЫ:

3.1. Научились векторизовать простые программы численного моделирования;

3.2. Авто-векторизация дает хороший прирост в производительности (в моем случае так вообще лучший) без приложения особых усилий.

4. ПРИЛОЖЕНИЕ:

4.1. Текст программы, векторизованной с помощью компилятора:

```
#define Nx  9000

#define Ny  9000
#define Nt  111

#define Xa  0.0f
#define Xb  4.0f
#define Ya  0.0f
#define Yb  4.0f

#define hx ((Xb - Xa) / (Nx - 1))
#define hy ((Yb - Ya) / (Ny - 1))
#define coeff1 (0.2f / ((1.0f / (hx * hx) + 1.0f / (hy * hy))))
#define coeff2 (0.5f * (5.0f / (hx * hx) - 1.0f / (hy * hy)))
#define coeff2b (0.5f * (5.0f / (hy * hy) - 1.0f / (hx * hx)))
#define coeff3 (0.25f * (1.0f / (hx * hx) + 1.0f / (hy * hy)))
```

```
#define X(j) (Xa + (j) * hx)
#define Y(i) (Ya + (i) * hy)

#define Xs1 (Xa + (Xb - Xa) / 3.0f)
#define Xs2 (Xa + (Xb - Xa) * 2.0f / 3.0f)
#define Ys1 (Ya + (Yb - Ya) * 2.0f / 3.0f)
#define Ys2 (Ya + (Yb - Ya) / 3.0f)
#define R (0.1f * ((Xb - Xa) > (Yb - Ya) ? (Yb - Ya) : (Xb - Xa)))

#define GRID_SIZE (Nx * Ny)
#define TIME_LAYERS 2

#define get(p, i, j) p[(i) * Nx + (j)]

#define max(a, b) ((a) > (b) ? (a) : (b))

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <math.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/times.h>

int main() {
    /* allocate memory */
    float *F_low = NULL, *F_high = NULL;
    F_low = malloc(GRID_SIZE * sizeof(float));
    F_high = malloc(GRID_SIZE * sizeof(float));
    float *p = malloc(GRID_SIZE * sizeof(float));
    if(!F_low || !F_high || !p) {
        perror("malloc");
        exit(errno);
    }

    /* init arrays */
    for(int i = 0; i < Ny; i++) {
        for(int j = 0; j < Nx; j++) {
            float xj = X(j);
            float yi = Y(i);
            if((xj - Xs1) * (xj - Xs1) + (yi - Ys1) * (yi - Ys1) < R * R) {
                get(p, i, j) = 0.1f;
            } else if((xj - Xs2) * (xj - Xs2) + (yi - Ys2) * (yi - Ys2) < R *
R) {
                get(p, i, j) = -0.1f;
            } else {
                get(p, i, j) = 0.0f;
            }
            get(F_low, i, j) = 0.0f;
        }
    }

    /* compute process */
    struct tms start, end;

    times(&start);
    float delta = 0.0f;
```

```
    for(int n = 0; n < Nt - 1; n++) {
        delta = 0.0f;
        float *F_curr = NULL, *F_next = NULL;
        if(n % 2 == 0) {
            F_curr = F_low;
            F_next = F_high;
        } else {
            F_next = F_low;
            F_curr = F_high;
        }
        for(int i = 1; i < Ny - 1; i++) {
            for(int j = 1; j < Nx - 1; j++) {
                float rez = coeff1 * (
                        coeff3 * (get(F_curr, i - 1, j - 1) + get(F_curr,  i
- 1, j + 1))   +
                        coeff2b * (get(F_curr, i - 1, j) + get(F_curr, i + 1,
j))           +
                        coeff2 * (get(F_curr, i, j - 1) + get(F_curr, i, j +
1))            +
                        coeff3 * (get(F_curr, i + 1, j - 1) + get(F_curr, i +
1, j + 1))    +
                        0.25f * (get(p, i - 1, j)   +
                        2.0f * get(p, i, j)         +
                        get(p, i, j - 1)            +
                        get(p, i, j + 1))           +
                        get(p, i + 1, j));
                delta = max(delta, fabs(get(F_curr, i, j) - rez));
                get(F_next, i, j) = rez;
            }
        }
    }
    times(&end);
    printf("n = %d, sigma = %.8f\n", Nt - 1, delta);
    printf("Total time: %lf sec.\n", (double)(end.tms_utime -
start.tms_utime) / sysconf(_SC_CLK_TCK));

    free(F_low);
    free(F_high);
    free(p);

    return 0;
}
```

4.2. Текст последнего варианта программы с ручной векторизацией (AVX2 + FMA):

```
#define ALIGN 32


#define VECTOR_SIZE 8
#define SHIFT1 1
#define SHIFT2 2

#define Nx  9000
#define REAL_Nx (Nx + SHIFT2)
#define Ny  9000
#define Nt  111
```

```
#define Xa  0.0f
#define Xb  4.0f
#define Ya  0.0f
#define Yb  4.0f

#define hx ((Xb - Xa) / (Nx - 1))
#define hy ((Yb - Ya) / (Ny - 1))
#define coeff1 (0.2f / ((1.0f / (hx * hx) + 1.0f / (hy * hy))))
#define coeff2 (0.5f * (5.0f / (hx * hx) - 1.0f / (hy * hy)))
#define coeff2b (0.5f * (5.0f / (hy * hy) - 1.0f / (hx * hx)))
#define coeff3 (0.25f * (1.0f / (hx * hx) + 1.0f / (hy * hy)))
#define X(j) (Xa + (j) * hx)
#define Y(i) (Ya + (i) * hy)

#define Xs1 (Xa + (Xb - Xa) / 3.0f)
#define Xs2 (Xa + (Xb - Xa) * 2.0f / 3.0f)
#define Ys1 (Ya + (Yb - Ya) * 2.0f / 3.0f)
#define Ys2 (Ya + (Yb - Ya) / 3.0f)
#define R (0.1f * ((Xb - Xa) > (Yb - Ya) ? (Yb - Ya) : (Xb - Xa)))

#define GRID_SIZE (REAL_Nx * Ny)

#define get(p, i, j) p[(i) * REAL_Nx + (j)]

#define max(a, b) ((a) > (b) ? (a) : (b))

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <math.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/times.h>
#include <immintrin.h>

int main() {
    /* allocate memory */
    float *F0 = _mm_malloc(GRID_SIZE * sizeof(float), ALIGN);
    float *F1 = _mm_malloc(GRID_SIZE * sizeof(float), ALIGN);
    float *p = _mm_malloc(GRID_SIZE * sizeof(float), ALIGN);
    if(!F0 || !F1 || !p) {
        perror("malloc");
        exit(errno);
    }

    /* init arrays */
    for(int i = 0; i < Ny; i++) {
        for(int j = 0; j < Nx; j++) {
            float xj = X(j);
            float yi = Y(i);
            if((xj - Xs1) * (xj - Xs1) + (yi - Ys1) * (yi - Ys1) < R * R) {
                get(p, i, j) = 0.1f;
            } else if((xj - Xs2) * (xj - Xs2) + (yi - Ys2) * (yi - Ys2) < R *
R) {
                get(p, i, j) = -0.1f;
            } else {
```

```
                get(p, i, j) = 0.0f;
            }
            get(F0, i, j) = 0.0f;
        }
    }

    /* prepare for vectorization */
    __m256 v_coeff1 = _mm256_set1_ps(coeff1);
    __m256 v_coeff2 = _mm256_set1_ps(coeff2);
    __m256 v_coeff2b = _mm256_set1_ps(coeff2b);
    __m256 v_coeff3 = _mm256_set1_ps(coeff3);
    __m256 v_delta = _mm256_setzero_ps();
    __m256 v_coeff4 = _mm256_set1_ps(0.25f);
    __m256 v_coeff5 = _mm256_set1_ps(2.0f);

    float *curr_F0; // current time layer
    float *curr_F1; // next time layer

    /* compute process */
    struct tms start, end;

    times(&start);
    for(int n = 0; n < Nt - 1; n++) {
        v_delta = _mm256_setzero_ps();

        if(n % 2 == 0) {
            curr_F0 = F0;
            curr_F1 = F1;
        } else {
            curr_F0 = F1;
            curr_F1 = F0;
        }

        for(int i = 1; i < Ny - 1; i++) {

            __m256 *v_F_prev = (__m256*)(curr_F0 + (i - 1) * REAL_Nx);
            __m256 *v_F_prev_shifted = (__m256*)(curr_F0 + (i - 1) * REAL_Nx
+ SHIFT2);
            __m256 *v_F_prev_vertical = (__m256*)(curr_F0 + (i - 1) * REAL_Nx
+ SHIFT1);

            __m256 *v_F_curr = (__m256*)(curr_F0 + i * REAL_Nx);
            __m256 *v_F_curr_shifted = (__m256*)(curr_F0 + i * REAL_Nx +
SHIFT2);

            __m256 *v_F_next = (__m256*)(curr_F0 + (i + 1) * REAL_Nx);
            __m256 *v_F_next_shifted = (__m256*)(curr_F0 + (i + 1) * REAL_Nx
+ SHIFT2);
            __m256 *v_F_next_vertical = (__m256*)(curr_F0 + (i + 1) * REAL_Nx
+ SHIFT1);

            __m256 *v_F_rez = (__m256*)(curr_F1 + i * REAL_Nx + SHIFT1);

            __m256 *v_p_curr = (__m256*)(p + i * REAL_Nx);
            __m256 *v_p_curr_shifted = (__m256*)(p + i * REAL_Nx + SHIFT2);
```

```
            __m256 *v_p_prev_vertical = (__m256*)(p + (i - 1) * REAL_Nx +
SHIFT1);
            __m256 *v_p_curr_vertical = (__m256*)(p + i * REAL_Nx + SHIFT1);
            __m256 *v_p_next_vertical = (__m256*)(p + (i + 1) * REAL_Nx +
SHIFT1);

            // main cycle
            for(int j = 0; j < Nx / VECTOR_SIZE; j++) {

                __m256 rez = v_coeff1 * (
                        _mm256_fmadd_ps(v_coeff3, (v_F_prev[j] +
v_F_prev_shifted[j]),
                        _mm256_fmadd_ps(v_coeff2b, (v_F_prev_vertical[j] +
v_F_next_vertical[j]),
                        _mm256_fmadd_ps(v_coeff2, (v_F_curr[j] +
v_F_curr_shifted[j]),
                        _mm256_fmadd_ps(v_coeff3, (v_F_next[j] +
v_F_next_shifted[j]),

                        _mm256_fmadd_ps(v_coeff5, v_p_curr_vertical[j],
                        v_coeff4 * (v_p_prev_vertical[j]    +
                                v_p_curr[j]                 +
                                v_p_curr_shifted[j]         +
                                v_p_next_vertical[j])))))));

                __m256 local_delta = _mm256_max_ps(v_F_rez[j] - rez, rez -
v_F_rez[j]);
                v_delta = _mm256_max_ps(v_delta, local_delta);
                v_F_rez[j] = rez;
            }
        }
    }
    times(&end);

    /* print max delta */
    float max_delta = 0.0f;
    float *vec_delta = (float*)(&v_delta);
    for(int i = 0; i < VECTOR_SIZE; i++) {
        max_delta = max(max_delta, vec_delta[i]);
    }
    printf("n = %d, sigma = %.8f\n", Nt - 1, max_delta);
    printf("Total time: %lf sec.\n", (double)(end.tms_utime -
start.tms_utime) / sysconf(_SC_CLK_TCK));

    _mm_free(F0);
    _mm_free(F1);
    _mm_free(p);

    return 0;
}
```