Отчет по лабораторной работе №2. Мандарханов Данил Михайлович. Группа 22207. Вариант float-2

Задание

Постановка задачи

1. Векторизовать программу из практического задания 1, используя наибольшее доступное векторное расширение. Векторизацию выполнить двумя способами:
    a. С помощью компилятора. При необходимости использовать специальные ключи и директивы компилятора, OpenMP, незначительную модификацию кода.
    b. С помощью Intel vector intrinsics. Векторизацию проводить поэтапно, проверяя время и правильность работы на каждом этапе. Сравнить две версии кода:
        i. допускающую в критическом участке кода невыровненные обращения в память,
        ii. только с выровненными обращениями в память в критическом участке кода.
2. Проанализировать производительность наиболее быстрой векторизованной версии программы аналогично анализу в задании 1 (включая roofline-модель). Сравнить результаты анализа с результатами в задании 1.

Параметры программы

$N_x = N_y = 8000, N_t = 100$

Не векторизованная программа

```
1   void runJacobyMethod (float *rho) {
2       float *phi = (float*)calloc(2 * N_x * N_y, sizeof(float));
```

```
    float *phi_new = phi + N_x * N_y;

    float h_x = (X_b - X_a) / (N_x - 1);
    float h_y = (Y_b - Y_a) / (N_y - 1);

    float mainKoef = 0.2 / (1.0 / (h_x * h_x) + 1.0 / (h_y * h_y));
    float firstKoef = 2.5 / (h_x * h_x) - 0.5 / (h_y * h_y);
    float secondKoef = 2.5 / (h_y * h_y) - 0.5 / (h_x * h_x);
    float thirdKoef = 0.25 / (h_x * h_x) + 0.25 / (h_y * h_y);

    int index;
    float d, stepDelta;
    float globalDelta = 1.0;

    int iterNumber = 0;

        long long t1, t2;
        double tDiff;
        struct timespec curTime;
        clock_gettime(CLOCK_BOOTTIME, &curTime);
        t1 = curTime.tv_sec * 1000000000 + curTime.tv_nsec;

    while (iterNumber <= N_t) {
        stepDelta = -1.0;

        for (int i = 1; i < N_y - 1; i++) {
            index = i * N_x;

            for (int j = 1; j < N_x - 1; j++) {
                index++;

                phi_new[index] = mainKoef * (firstKoef * (phi[index - 1] + phi[index + 1]) +
                                             secondKoef * (phi[index - N_x] + phi[index + N_x]) +
```

```
                                        thirdKoef * (phi[index - N_x - 1] + phi[index - N_x + 1] + phi[index + N_x - 1] + phi[index + N_x + 1]) +
                                        2.0 * rho[index] +
                                        (rho[index - N_x] + rho[index + N_x] + rho[index - 1] + rho[index + 1]) * 0.25);

                d = fabs(phi[index] - phi_new[index]);
                if (d > stepDelta) stepDelta = d;
            }
        }

        if ((stepDelta - globalDelta) < 0.0000001) {
            globalDelta = stepDelta;
            swapFloatPointers(&phi, &phi_new);
            iterNumber++;
        }
        else {
            printf("Delta is growwing!\nJacoby method stopped\n");
            break;
        }
    }

        clock_gettime(CLOCK_BOOTTIME, &curTime);
        t2 = curTime.tv_sec * 1000000000 + curTime.tv_nsec;
        tDiff = (double) (t2 - t1) / 1000000000.0;
        printf("Time = %g s\n", tDiff);

    if(iterNumber % 2 == 1) swapFloatPointers(&phi, &phi_new);
    free(phi);
}
```

Время работы

```
epsmim@comrade:~/Desktop/Mandarkhanov/Lab_2$ gcc mainDefault.c -lm -O0
epsmim@comrade:~/Desktop/Mandarkhanov/Lab_2$ time ./a.out
Time = 170.215 s

real    2m51.386s
user    2m50.311s
sys     0m0.984s
```

```
epsmim@comrade:~/Desktop/Mandarkhanov/Lab_2$ gcc mainDefault.c -lm -Ofast -o default.out
epsmim@comrade:~/Desktop/Mandarkhanov/Lab_2$ time ./default.out
Time = 29.3247 s

real    0m29.785s
user    0m29.179s
sys     0m0.559s
```

Программа, векторизованная с помощью компилятора: текст программы, время работы

```
1   #pragma omp simd reduction(max: stepDelta)
2   for (int j = 1; j < N_x - 1; j++) {
```

```
root@DESKTOP-BUBJJQK:~/epsmim/lab_2# gcc mainOmpSimd.c -O2 -lm
root@DESKTOP-BUBJJQK:~/epsmim/lab_2# ./a.out
Time = 29.3974 s
root@DESKTOP-BUBJJQK:~/epsmim/lab_2# gcc mainOmpSimd.c -O2 -lm -fopenmp
root@DESKTOP-BUBJJQK:~/epsmim/lab_2# ./a.out
Time = 12.0328 s
```

Приложение 3. Программа, векторизованная с помощью Intel vector intrinsics: текст программы, описание различных этапов векторизации, время работы на различных этапах

В критическом участке кода не выровненные обращения в память

```
void runJacobyMethod (float *rho) {
    float *phi;
    phi = (float*)malloc(2 * N_x * N_y * sizeof(float));
    float *phi_new = phi + N_x * N_y;

    for (int i = 0; i < N_y * 2; i++) {
        for (int j = 0; j < N_x; j++) {
            phi[i*N_y + j] = 0.0f;
        }
    }

    float h_x = (X_b - X_a) / (N_x - 1);
    float h_y = (Y_b - Y_a) / (N_y - 1);

    float mainKoef = 0.2 / (1.0 / (h_x * h_x) + 1.0 / (h_y * h_y));      __m128 mainKoef_m128  = _mm_set1_ps(mainKoef);
    float firstKoef = 2.5 / (h_x * h_x) - 0.5 / (h_y * h_y);             __m128 firstKoef_m128 = _mm_set1_ps(firstKoef);
    float secondKoef = 2.5 / (h_y * h_y) - 0.5 / (h_x * h_x);            __m128 secondKoef_m128= _mm_set1_ps(secondKoef);
    float thirdKoef = 0.25 / (h_x * h_x) + 0.25 / (h_y * h_y);           __m128 thirdKoef_m128 = _mm_set1_ps(thirdKoef);

    int index, vec_index;
    float d, stepDelta;
    float globalDelta = 1.0;
    int iterNumber = 0;

    int step1vector = 4;
    int countVecInLine = (N_x - 2) / 4;
    int remainsInLine = (N_x - 2) % 4;

        long long t1, t2;
        double tDiff;
        struct timespec curTime;
```

```
32          clock_gettime(CLOCK_BOOTTIME, &curTime);
33          t1 = curTime.tv_sec * 1000000000 + curTime.tv_nsec;
34
35      while (iterNumber <= N_t) {
36          stepDelta = -1.0;
37
38          for (int i = 1; i < N_y - 1; i++) {
39              index = i * N_x;
40
41              for (int j = 1; j < countVecInLine * step1vector + 1; j += step1vector) {
42                  vec_index = index  + j;
43
44                  __m128 v_phi_left     = _mm_loadu_ps(&phi[vec_index - 1]);
45                  __m128 v_phi_right    = _mm_loadu_ps(&phi[vec_index + 1]);
46                  __m128 v_phi_bottom   = _mm_loadu_ps(&phi[vec_index - N_x]);
47                  __m128 v_phi_top      = _mm_loadu_ps(&phi[vec_index + N_x]);
48                  __m128 v_phi_bot_left = _mm_loadu_ps(&phi[vec_index - N_x - 1]);
49                  __m128 v_phi_bot_right= _mm_loadu_ps(&phi[vec_index - N_x + 1]);
50                  __m128 v_phi_top_left = _mm_loadu_ps(&phi[vec_index + N_x - 1]);
51                  __m128 v_phi_top_right= _mm_loadu_ps(&phi[vec_index + N_x + 1]);
52
53                  __m128 v_rho_center   = _mm_loadu_ps(&rho[vec_index]);
54                  __m128 v_rho_bottom   = _mm_loadu_ps(&rho[vec_index - N_x]);
55                  __m128 v_rho_top      = _mm_loadu_ps(&rho[vec_index + N_x]);
56                  __m128 v_rho_left     = _mm_loadu_ps(&rho[vec_index - 1]);
57                  __m128 v_rho_right    = _mm_loadu_ps(&rho[vec_index + 1]);
58
59                  __m128 first_line    = _mm_add_ps(_mm_mul_ps(firstKoef_m128, _mm_add_ps(v_phi_left, v_phi_right)),
60                                                    _mm_mul_ps(secondKoef_m128, _mm_add_ps(v_phi_bottom, v_phi_top)));
61                  __m128 second_line   = _mm_mul_ps(thirdKoef_m128,
62                                                    _mm_add_ps(_mm_add_ps(v_phi_bot_left, v_phi_top_left),
63                                                               _mm_add_ps(v_phi_bot_right, v_phi_top_right)));
64                  __m128 third_line    = _mm_add_ps(_mm_mul_ps(_mm_set1_ps(0.25f),
65                                                               _mm_add_ps(_mm_add_ps(v_rho_bottom, v_rho_top),
```

```
                                                                    _mm_add_ps(v_rho_left, v_rho_right))),
                                                _mm_mul_ps(_mm_set1_ps(2.0f), v_rho_center));
                __m128 result        = _mm_mul_ps(mainKoef_m128,
                                                _mm_add_ps(first_line, _mm_add_ps(second_line, third_line)));

                _mm_storeu_ps(&phi_new[vec_index], result);

                for (int di = 0; di < step1vector; di++) {
                    d = fabs(phi[vec_index + di] - phi_new[vec_index + di]);
                    if (d > stepDelta) stepDelta = d;
                }
            }
        }

        int N_x_remains_index = countVecInLine * step1vector + 1;
        for (int i = 1; i < N_y - 1; i++) {
            index = i * N_x;
            for (int j = N_x_remains_index; j < N_x - 1; j++) {
                index++;

                phi_new[index] = mainKoef * (firstKoef * (phi[index - 1] + phi[index + 1]) +
                                             secondKoef * (phi[index - N_x] + phi[index + N_x]) +
                                             thirdKoef * (phi[index - N_x - 1] + phi[index - N_x + 1] + phi[index + N_x - 1] + phi[index + N_x + 1]) +
                                             2.0f * rho[index] +
                                             (rho[index - N_x] + rho[index + N_x] + rho[index - 1] + rho[index + 1]) * 0.25f);

                d = fabs(phi[index] - phi_new[index]);
                if (d > stepDelta) stepDelta = d;
            }
        }


```

```
98          if ((stepDelta - globalDelta) < 0.0000001) {
99              globalDelta = stepDelta;
100             swapFloatPointers(&phi, &phi_new);
101             iterNumber++;
102         }
103         else {
104             printf("Delta is growwing!\nJacoby method stopped\n");
105             break;
106         }
107     }
108
109         clock_gettime(CLOCK_BOOTTIME, &curTime);
110         t2 = curTime.tv_sec * 1000000000 + curTime.tv_nsec;
111         tDiff = (double) (t2 - t1) / 1000000000.0;
112         printf("Time = %g s\n", tDiff);
113
114     fillFile(phi, "phi_unalign.dat");
115
116     if(iterNumber % 2 == 1) swapFloatPointers(&phi, &phi_new);
117     free(phi);
118 }
```

```
epsmim@comrade:~/Desktop/Mandarkhanov/Lab_2$ gcc main128unalign.c -lm -Ofast -march=native -o unalign.out
epsmim@comrade:~/Desktop/Mandarkhanov/Lab_2$ time ./unalign.out
Time = 24.9616 s

real    0m25.651s
user    0m25.083s
sys     0m0.527s
```

Только с выровненными обращениями в память в критическом участке кода

```
1   void runJacobyMethod (float* rho) {
2       int countVecInLine = (N_x - 2) / VEC_SIZE;
3       int remainsInLine = (N_x - 2) % VEC_SIZE;
```

```
    int N_x_align = VEC_SIZE + (countVecInLine * VEC_SIZE) + VEC_SIZE; // left VEC_SIZE for border
                                                                       // right VEC_SIZE for remains and border

    float* phi;
    if (posix_memalign((void**)&phi, 16, 2*N_x_align*N_y*sizeof(float)) != 0) exit(1);
    float* phi_new = phi + N_x * N_y;

    for (int i = 0; i < N_y * 2; i++) {
        for (int j = 0; j < N_x_align; j++) {
            phi[i*N_y + j] = 0.0f;
        }
    }

    float h_x = (X_b - X_a) / (N_x - 1);
    float h_y = (Y_b - Y_a) / (N_y - 1);

    float mainKoef = 0.2 / (1.0 / (h_x * h_x) + 1.0 / (h_y * h_y));      __m128 mainKoef_m128  = _mm_set1_ps(mainKoef);
    float firstKoef = 2.5 / (h_x * h_x) - 0.5 / (h_y * h_y);             __m128 firstKoef_m128 = _mm_set1_ps(firstKoef);
    float secondKoef = 2.5 / (h_y * h_y) - 0.5 / (h_x * h_x);            __m128 secondKoef_m128= _mm_set1_ps(secondKoef);
    float thirdKoef = 0.25 / (h_x * h_x) + 0.25 / (h_y * h_y);           __m128 thirdKoef_m128 = _mm_set1_ps(thirdKoef);

    int index, vec_index;
    float d, stepDelta;
    float globalDelta = 1.0;
    int iterNumber = 0;

        long long t1, t2;
        double tDiff;
        struct timespec curTime;
        clock_gettime(CLOCK_BOOTTIME, &curTime);
        t1 = curTime.tv_sec * 1000000000 + curTime.tv_nsec;

    while (iterNumber <= N_t) {
        stepDelta = -1.0;
```

```
38
39          for (int i = 1; i < N_y - 1; i++) {
40              index = i * N_x + 4;
41
42              for (int j = 0; j < countVecInLine * VEC_SIZE + 1; j += VEC_SIZE) {
43                  vec_index = index + j;
44                  __m128 av_phi_left      = _mm_load_ps(&phi[vec_index - 4]);
45                  __m128 av_phi_left_bot  = _mm_load_ps(&phi[vec_index - 4 - N_x]);
46                  __m128 av_phi_left_top  = _mm_load_ps(&phi[vec_index - 4 + N_x]);
47
48                  __m128 av_phi_right     = _mm_load_ps(&phi[vec_index + 4]);
49                  __m128 av_phi_right_bot = _mm_load_ps(&phi[vec_index + 4 - N_x]);
50                  __m128 av_phi_right_top = _mm_load_ps(&phi[vec_index + 4 + N_x]);
51
52                  __m128 av_phi_cnt       = _mm_load_ps(&phi[vec_index]);
53                  __m128 av_phi_bot       = _mm_load_ps(&phi[vec_index - N_x]);
54                  __m128 av_phi_top       = _mm_load_ps(&phi[vec_index + N_x]);
55
56                  __m128 v_phi_left_top  = _mm_blend_ps(_mm_shuffle_ps(av_phi_left_top , av_phi_left_top , _MM_SHUFFLE(ANY,
    ANY, ANY, 3  )),
57                                                        _mm_shuffle_ps(av_phi_top      , av_phi_top      , _MM_SHUFFLE(2  ,
    1  , 0  , ANY)),
58                                                        MASK_ABBB);
59                  __m128 v_phi_left      = _mm_blend_ps(_mm_shuffle_ps(av_phi_left     , av_phi_left     , _MM_SHUFFLE(ANY,
    ANY, ANY, 3  )),
60                                                        _mm_shuffle_ps(av_phi_cnt      , av_phi_cnt      , _MM_SHUFFLE(2  ,
    1  , 0  , ANY)),
61                                                        MASK_ABBB);
62                  __m128 v_phi_left_bot  = _mm_blend_ps(_mm_shuffle_ps(av_phi_left_bot , av_phi_left_bot , _MM_SHUFFLE(ANY,
    ANY, ANY, 3  )),
63                                                        _mm_shuffle_ps(av_phi_bot      , av_phi_bot      , _MM_SHUFFLE(2  ,
    1  , 0  , ANY)),
64                                                        MASK_ABBB);
65
```

```
66                  __m128 v_phi_right_top = _mm_blend_ps(_mm_shuffle_ps(av_phi_top      , av_phi_top      , _MM_SHUFFLE(ANY, 3  , 2  , 1  )),
67                                                        _mm_shuffle_ps(av_phi_right_top, av_phi_right_top, _MM_SHUFFLE(0  , ANY, ANY, ANY)),
68                                                        MASK_AAAB);
69                  __m128 v_phi_right     = _mm_blend_ps(_mm_shuffle_ps(av_phi_cnt      , av_phi_cnt      , _MM_SHUFFLE(ANY, 3  , 2  , 1  )),
70                                                        _mm_shuffle_ps(av_phi_right    , av_phi_right    , _MM_SHUFFLE(0  , ANY, ANY, ANY)),
71                                                        MASK_AAAB);
72                  __m128 v_phi_right_bot = _mm_blend_ps(_mm_shuffle_ps(av_phi_bot      , av_phi_bot      , _MM_SHUFFLE(ANY, 3  , 2  , 1  )),
73                                                        _mm_shuffle_ps(av_phi_right_bot, av_phi_right_bot, _MM_SHUFFLE(0  , ANY, ANY, ANY)),
74                                                        MASK_AAAB);
75
76                  __m128 av_rho_cnt      = _mm_load_ps(&rho[vec_index]);
77                  __m128 av_rho_bot      = _mm_load_ps(&rho[vec_index - N_x]);
78                  __m128 av_rho_top      = _mm_load_ps(&rho[vec_index + N_x]);
79                  __m128 av_rho_left     = _mm_load_ps(&rho[vec_index - 4]);
80                  __m128 av_rho_right    = _mm_load_ps(&rho[vec_index + 4]);
81
82                  __m128 v_rho_left      = _mm_blend_ps(_mm_shuffle_ps(av_rho_left     , av_rho_left     , _MM_SHUFFLE(ANY, ANY, ANY, 3  )),
83                                                        _mm_shuffle_ps(av_rho_cnt      , av_rho_cnt      , _MM_SHUFFLE(2  , 1  , 0  , ANY)),
84                                                        MASK_ABBB);
85                  __m128 v_rho_right     = _mm_blend_ps(_mm_shuffle_ps(av_rho_cnt      , av_rho_cnt      , _MM_SHUFFLE(ANY, 3  , 2  , 1  )),
86                                                        _mm_shuffle_ps(av_rho_right    , av_rho_right    , _MM_SHUFFLE(0  , ANY, ANY, ANY)),
87                                                        MASK_AAAB);
88
89                  __m128 first_line    = _mm_add_ps(_mm_mul_ps(firstKoef_m128, _mm_add_ps(v_phi_left, v_phi_right)),
```

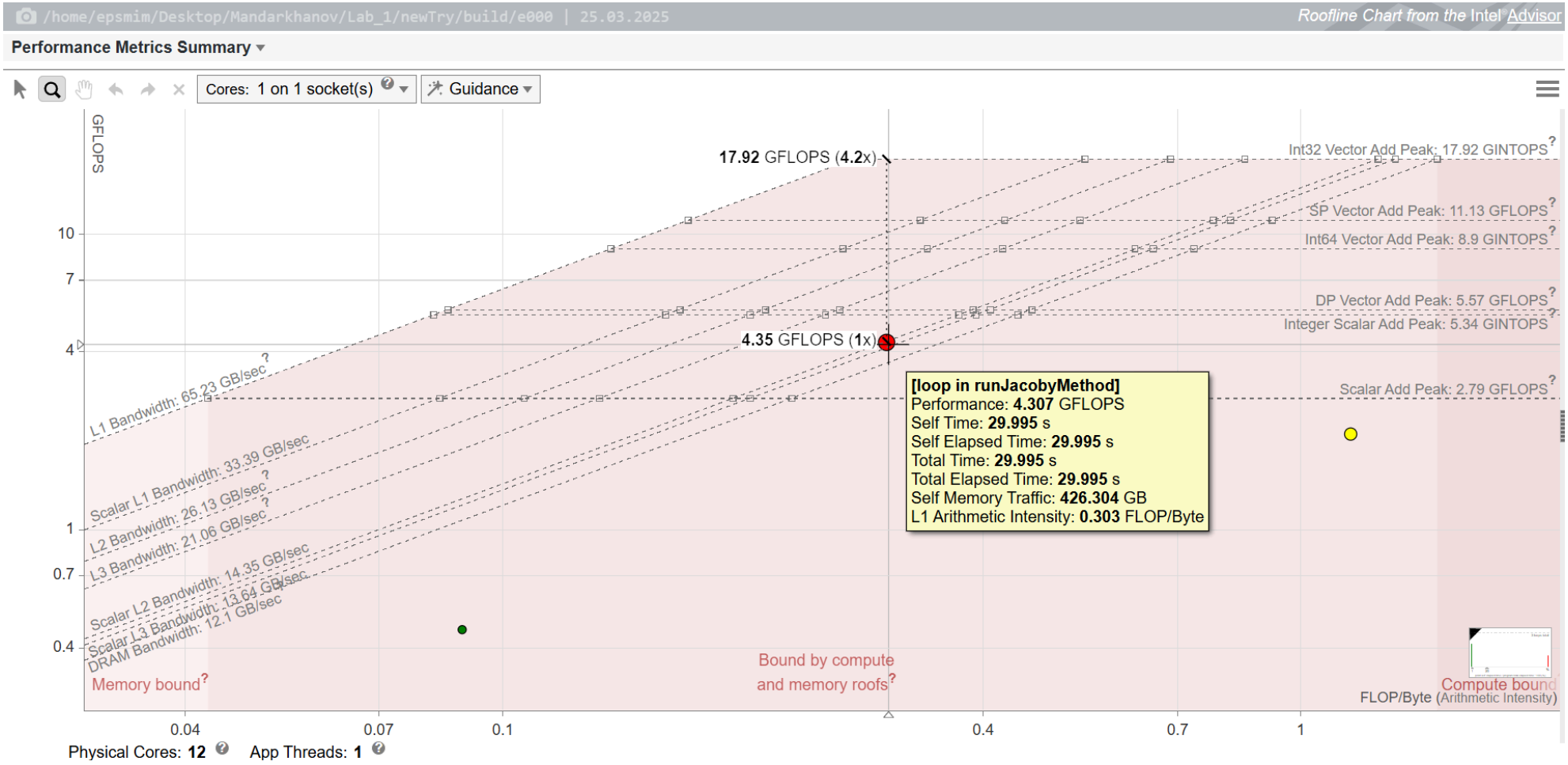
```
                                                _mm_mul_ps(secondKoef_m128, _mm_add_ps(av_phi_bot, av_phi_top)));
                __m128 second_line  = _mm_mul_ps(thirdKoef_m128,
                                                _mm_add_ps(_mm_add_ps(v_phi_left_bot, v_phi_left_top),
                                                           _mm_add_ps(v_phi_right_bot, v_phi_right_top)));
                __m128 third_line   = _mm_add_ps(_mm_mul_ps(_mm_set1_ps(0.25f),
                                                           _mm_add_ps(_mm_add_ps(av_rho_bot, av_rho_top),
                                                                      _mm_add_ps(v_rho_left, v_rho_right))),
                                                _mm_mul_ps(_mm_set1_ps(2.0f), av_rho_cnt));
                __m128 result       = _mm_mul_ps(mainKoef_m128,
                                                 _mm_add_ps(first_line, _mm_add_ps(second_line, third_line)));

                _mm_store_ps(&phi_new[vec_index], result);

                for (int di = 0; di < VEC_SIZE; di++) {
                    d = fabs(phi[vec_index + di] - phi_new[vec_index + di]);
                    if (d > stepDelta) stepDelta = d;
                }
            }
        }

        int N_x_remains_index = countVecInLine * VEC_SIZE + VEC_SIZE + 1;
        for (int i = 1; i < N_y - 1; i++) {
            index = i * N_x;
            for (int j = N_x_remains_index; j < N_x - 1; j++) {
                index++;

                phi_new[index] = mainKoef * (firstKoef * (phi[index - 1] + phi[index + 1]) +
                                             secondKoef * (phi[index - N_x] + phi[index + N_x]) +
                                             thirdKoef * (phi[index - N_x - 1] + phi[index - N_x + 1] + phi[index + N_x -
1] + phi[index + N_x + 1]) +
                                             2.0f * rho[index] +
                                             (rho[index - N_x] + rho[index + N_x] + rho[index - 1] + rho[index + 1]) *
0.25f);

```

```
                d = fabs(phi[index] - phi_new[index]);
                if (d > stepDelta) stepDelta = d;
            }
        }


        if ((stepDelta - globalDelta) < 0.0000001) {
            globalDelta = stepDelta;
            swapFloatPointers(&phi, &phi_new);
            iterNumber++;
        }
        else {
            printf("Delta is growwing!\nJacoby method stopped\n");
            break;
        }
    }

        clock_gettime(CLOCK_BOOTTIME, &curTime);
        t2 = curTime.tv_sec * 1000000000 + curTime.tv_nsec;
        tDiff = (double) (t2 - t1) / 1000000000.0;
        printf("Time = %g s\n", tDiff);

    fillFile(phi, "phi_align.dat");

    if(iterNumber % 2 == 1) swapFloatPointers(&phi, &phi_new);
    free(phi);
}
```

```
epsmim@comrade:~/Desktop/Mandarkhanov/Lab_2$ gcc main128align.c -lm -Ofast -march=native -o align.out
epsmim@comrade:~/Desktop/Mandarkhanov/Lab_2$ time ./align.out
Time = 28.5871 s

real    0m29.279s
user    0m28.694s
sys     0m0.535s
```

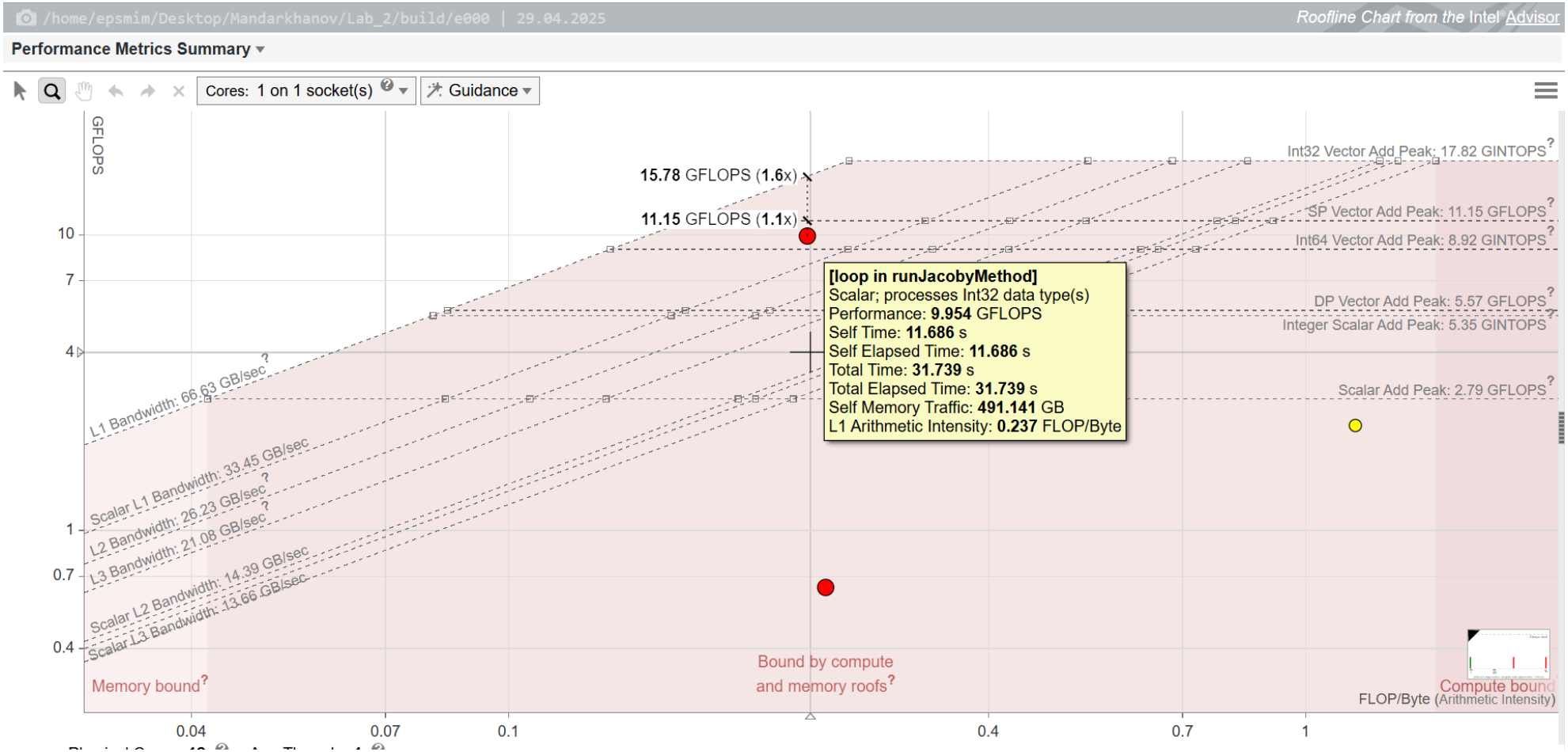
Roofline-модель

Не векторизованная программа



- Упирается в L3 кэш-память
- Желтая точка справа - initRho()

Векторизованная программа



- Тут уже по памяти проблем нет, так как все вектора уже лежат в L3 кэш-памяти, кэш-промахов значительно меньше
- Точка красная снизу - скалярный подсчет оставшегося вектора, который не поместился в m128