МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

**НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №4

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 распараллеливать в потоках простые программы численного моделирования.

**Вариант №3:** *решение уравнения Пуассона методом Якоби на float-ах.*

Алгоритм моделирует установление стационарного распределения 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.

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

**8 ядер, 8 потоков, гипертрединга нет.**

* + 1. Компилятор**: Intel(R) C++ Intel(R) 64 Compiler for applications running on Intel(R) 64, Version 19.1.1.216 Build 20200306;**
    2. Ключи компиляции (наиболее важные): **/Qopenmp /O2 /tune:coffeelake /arch:CORE-AVX2;**
    3. Параметры программы: **Nx = Ny = 9000, Nt = 110**.
  1. Листинг программы из пункта 1(см. Приложение A);
  2. Листинг программы из пункта 2(см. Приложение B);
  3. Листинг программы из пункта 3(см. Приложение C);
  4. Все программы при любом количестве потоков выдают **Delta = 1.18546e-08.**
  5. Итоговая таблица с временами выполнения:

|  |  |  |  |
| --- | --- | --- | --- |
| **Потоки, кол-во** | **Вариант 1, cек.** | **Вариант 2, cек.** | **Вариант 3, cек.** |
| 1 | 6.988 | 5.425 | 5.41 |
| 2 | 4.844 | 2.87 | 2.855 |
| 3 | 4.516 | 2.082 | 2.058 |
| 4 | 4.577 | 1.721 | 1.727 |
| 5 | 4.669 | 1.568 | 1.573 |
| 6 | 4.614 | 1.579 | 1.58 |
| 7 | 4.61 | 1.569 | 1.656 |
| 8 | 4.779 | 1.682 | 1.736 |

Победителем по наилучшему показателю является вариант 2, однако вариант 3 практически никак не отличается по времени работы от варианта 2.

1 вариант ожидаемо хуже при любом количестве потоков, так как в нем не применялась оптимизация по памяти из 3 задания.

Все программы к 4-5 потоку достигают своей пиковой скорости работы, далее прироста либо нет, либо начинается только ухудшение производительности.

* 1. Производительность 1 и 3 программ на 8 потоках и сравнение с программой из задания 3:
     + 1. **Число инструкций на такт:**

**1 программа:** 0.25438819638

**3 программа:** 0.76863950807

**Из задания 3:** 1.51

У программ 1 и 3 довольно низкие показатели по сравнению с программой из задания 3. Похоже, что ядра процессора большую часть времени простаивают без дела.

Причем, однопоточная версия лучше 8-поточной 1-й программы примерно в 6 раз, что примерно равно количеству потоков.

* + - 1. **Процент кэш-промахов для кэша 3 уровня:**

**1 программа:** 73%.

**3 программа:** 22%.

**Из задания 3:** 19.76%

Из-за отсутствия оптимизации по памяти 1 программа практически всегда промахивается при доступе к LLC. Этим можно объяснять и низкий IPC, процессор долго ждет данные. Обратная ситуация у 3 программы и программы из здания 3, у которых схожие показатели.

* + - 1. **Процент кэш-промахов для кэша 1 уровня:**

**1 программа:** 3%.

**3 программа:** 2%.

**Из задания 3:** 1.96%

Примерно одинаковые результаты.

* + - 1. **Процент неправильно предсказанных переходов:**

**1 программа:** 1.08%.

**3 программа:** 0.99%.

**Из задания 3:** 1.68%

Примерно одинаковые результаты.

* + - 1. **Время работы:**

**1 программа:** 4.779 сек.

**3 программа:** 1.736 сек.

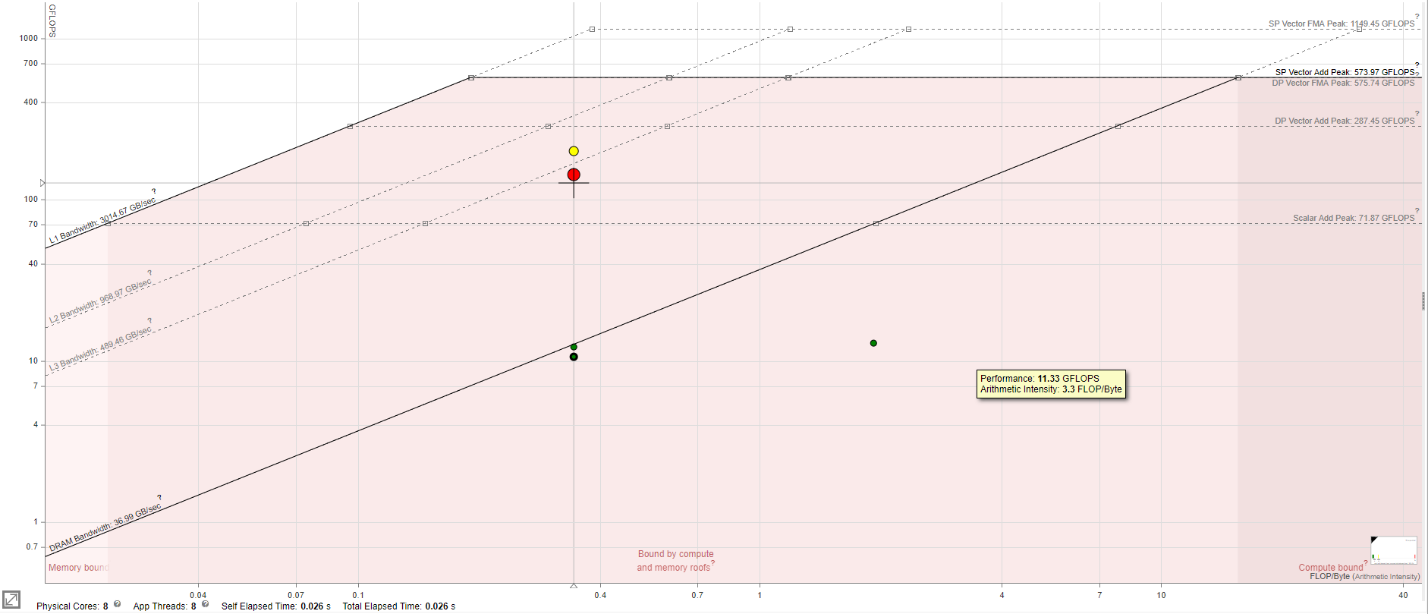
**Из задания 3:** 5.082 сек.

1 программа медленнее 3 в 2.75 раза. В примерно такое же количество раз (около 3) у этой программы больше промахов при доступе к LLC и хуже IPC.

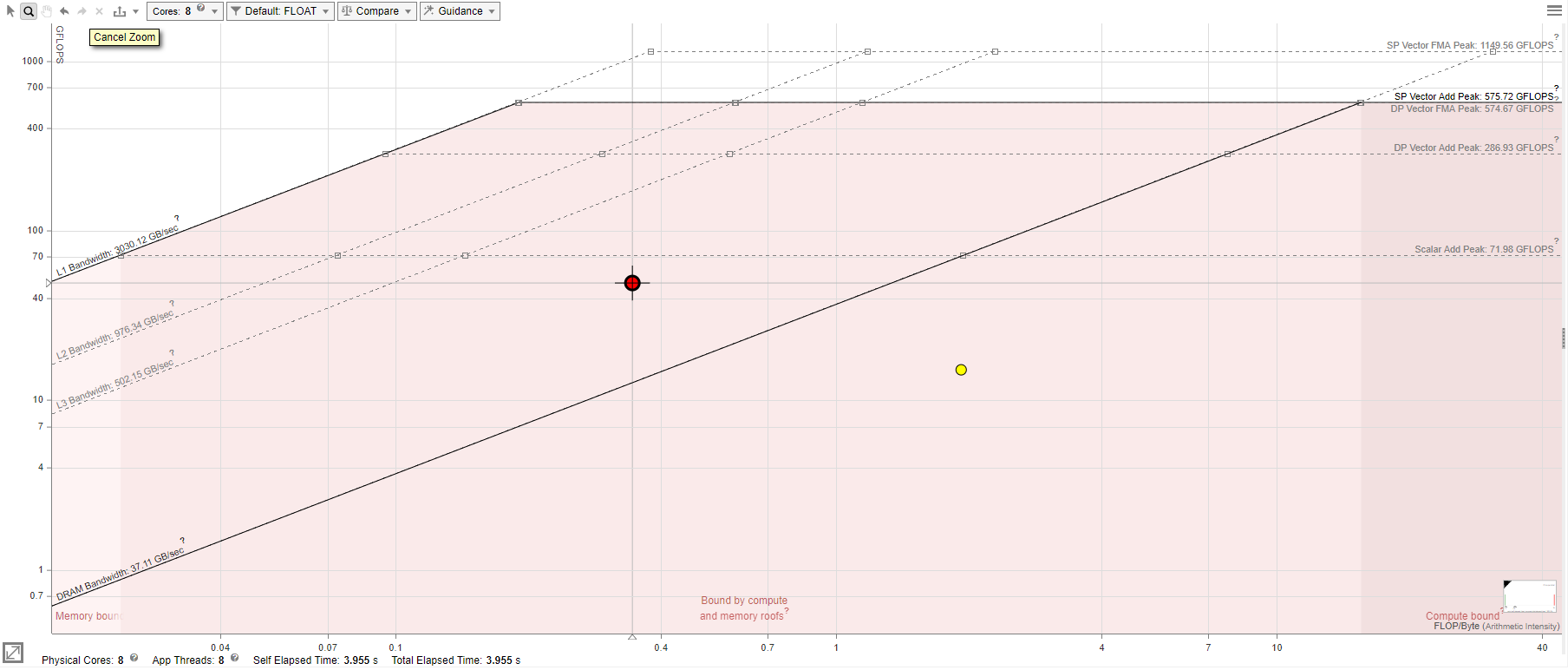
Программа из здания 3 очень близка по времени работы к программе 1. Из соотношения их IPC можно сказать, что инструкции в достаточной мере вместились бы и на одном ядре, но в многопоточной 1 программе они были просто размазаны по 8 ядрам, которые все равно простаивали в ожидании данных для инструкций.

* 1. **Roofline-модель с точками, соответствующими основному циклу программы для 8 поточных программ 1 и 3:**

**Для программы 3:**

****

**Для программы 1:**

****

Точки основного цикла находятся на одном и том же уровне по оси пропускной способности памяти, но по оси производительности процессора в разных. В купе с тем, что в прошлом пункте было выяснено влияние кэш-промахов на IPC и его соотношения у разных программ, можно сделать вывод, что дальнейшему увеличению ускорения при распараллеливании препятствует неспособность памяти успевать обеспечивать процессор данными для исполнения инструкций.

1. **ВЫВОДЫ:**
   1. Научились распараллеливать в потоках простые программы численного моделирования;
   2. Память все еще является бутылочным горлышком производительности программ.
2. **ПРИЛОЖЕНИЕ A. Программа 1**

constexpr auto ALIGN = 32;

constexpr auto VECTOR\_SIZE = 8;

constexpr auto SHIFT1 = 1;

constexpr auto Nx = 9000;

constexpr auto Ny = 9000;

constexpr auto Nt = 111;

constexpr float Xa = 0.0f;

constexpr float Xb = 4.0f;

constexpr float Ya = 0.0f;

constexpr float Yb = 4.0f;

constexpr float hx = ((Xb - Xa) / (Nx - 1));

constexpr float hy = ((Yb - Ya) / (Ny - 1));

constexpr float coeff1 = (0.2f / ((1.0f / (hx \* hx) + 1.0f / (hy \* hy))));

constexpr float coeff2 = (0.5f \* (5.0f / (hx \* hx) - 1.0f / (hy \* hy)));

constexpr float coeff2b = (0.5f \* (5.0f / (hy \* hy) - 1.0f / (hx \* hx)));

constexpr float coeff3 = (0.25f \* (1.0f / (hx \* hx) + 1.0f / (hy \* hy)));

inline float X(size\_t j) { return (Xa + (j)\*hx); }

inline float Y(size\_t i) { return (Ya + (i)\*hy); }

constexpr float Xs1 = (Xa + (Xb - Xa) / 3.0f);

constexpr float Xs2 = (Xa + (Xb - Xa) \* 2.0f / 3.0f);

constexpr float Ys1 = (Ya + (Yb - Ya) \* 2.0f / 3.0f);

constexpr float Ys2 = (Ya + (Yb - Ya) / 3.0f);

constexpr float R = (0.1f \* ((Xb - Xa) > (Yb - Ya) ? (Yb - Ya) : (Xb - Xa)));

constexpr float GRID\_SIZE = (Nx \* Ny);

constexpr int THREADS = 8;

constexpr int CHUNK = Ny / THREADS;

inline float& get(float\* p, size\_t i, size\_t j) { return p[(i)\*Nx + (j)]; }

#include <iostream>

#include <fstream>

#include <cerrno>

#include <ctime>

#include <algorithm>

#include <immintrin.h>

using namespace std;

/\* prepare for vectorization \*/

\_\_m256 v\_coeff1;

\_\_m256 v\_coeff2;

\_\_m256 v\_coeff2b;

\_\_m256 v\_coeff3;

\_\_m256 v\_delta;

\_\_m256 global\_v\_delta;

\_\_m256 v\_coeff4;

\_\_m256 v\_coeff5;

inline void compute\_process(float\* F0, float\* F1, float\* p, int i) {

    for(int j = 1; j < Nx / VECTOR\_SIZE - 1; j++) {

        \_\_m256 rez = \_mm256\_mul\_ps(v\_coeff1, (

                        \_mm256\_fmadd\_ps(v\_coeff3, \_mm256\_add\_ps(((\_\_m256\*)(F0 + (i - 1) \* Nx - SHIFT1))[j], ((\_\_m256\*)(F0 + (i - 1)\*Nx + SHIFT1))[j]),

                        \_mm256\_fmadd\_ps(v\_coeff2b, \_mm256\_add\_ps(((\_\_m256\*)(F0 + (i - 1) \* Nx))[j], ((\_\_m256\*)(F0 + (i + 1) \* Nx))[j]),

                        \_mm256\_fmadd\_ps(v\_coeff2, \_mm256\_add\_ps(((\_\_m256\*)(F0 + (i) \* Nx - SHIFT1))[j], ((\_\_m256\*)(F0 + (i) \* Nx + SHIFT1))[j]),

                        \_mm256\_fmadd\_ps(v\_coeff3, \_mm256\_add\_ps(((\_\_m256\*)(F0 + (i + 1) \* Nx - SHIFT1))[j], ((\_\_m256\*)(F0 + (i + 1) \* Nx + SHIFT1))[j]),

                        \_mm256\_fmadd\_ps(v\_coeff5, ((\_\_m256\*)(p + (i) \* Nx))[j],

                            \_mm256\_mul\_ps(v\_coeff4, (

                                \_mm256\_add\_ps(

                                    \_mm256\_add\_ps(

                                        \_mm256\_add\_ps(((\_\_m256\*)(p + (i - 1) \* Nx))[j], ((\_\_m256\*)(p + (i) \* Nx - SHIFT1))[j]),

                                            ((\_\_m256\*)(p + (i) \* Nx + SHIFT1))[j]),

                                                ((\_\_m256\*)(p + (i + 1) \* Nx))[j])

                                    )

                                )

                            )

                        )

                        )

                        )

                        )

                    )

                );

        \_\_m256 prev\_rez = ((\_\_m256\*)(F1 + (i) \* Nx))[j];

        v\_delta = \_mm256\_max\_ps(v\_delta, \_mm256\_max\_ps(\_mm256\_sub\_ps(prev\_rez, rez), \_mm256\_sub\_ps(rez, prev\_rez)));

        ((\_\_m256\*)(F1 + (i) \* Nx))[j] = rez;

    }

}

int main() {

    /\* allocate memory \*/

    float \*F0 = (float\*)\_mm\_malloc(GRID\_SIZE \* sizeof(float), ALIGN);

    float \*F1 = (float\*)\_mm\_malloc(GRID\_SIZE \* sizeof(float), ALIGN);

    float \*p = (float\*)\_mm\_malloc(GRID\_SIZE \* sizeof(float), ALIGN);

    if(!F0 || !F1 || !p) {

        perror("\_mm\_malloc");

        exit(errno);

    }

    /\* prepare for vectorization \*/

    v\_coeff1 = \_mm256\_set1\_ps(coeff1);

    v\_coeff2 = \_mm256\_set1\_ps(coeff2);

    v\_coeff2b = \_mm256\_set1\_ps(coeff2b);

    v\_coeff3 = \_mm256\_set1\_ps(coeff3);

    v\_delta = \_mm256\_setzero\_ps();

    v\_coeff4 = \_mm256\_set1\_ps(0.25f);

    v\_coeff5 = \_mm256\_set1\_ps(2.0f);

    clock\_t start, end;

#pragma omp parallel proc\_bind(close) num\_threads(THREADS) private(v\_delta) // OMP\_PLACES is "cores" by default

    {

        /\* init arrays \*/

#pragma omp for schedule(static, CHUNK)

        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;

            }

        }

#pragma omp barrier

        /\* compute process \*/

#pragma omp single

        start = clock();

        for (int n = 0; n < Nt - 1; n++) {

#pragma omp single

            global\_v\_delta = \_mm256\_setzero\_ps();

            v\_delta = \_mm256\_setzero\_ps();

#pragma omp for schedule(static, CHUNK)

            for (int i = 2; i < Ny - 3; i++) {

                compute\_process(F0, F1, p, i);

            }

#pragma omp critical

            {

                global\_v\_delta = \_mm256\_max\_ps(v\_delta, global\_v\_delta);

            }

#pragma omp single

            swap(F0, F1);

#pragma omp barrier

        }

#pragma omp single

        end = clock();

    }

    /\* print max delta \*/

    float max\_delta = 0.0f;

    float\* vec\_delta = (float\*)(&global\_v\_delta);

    for (int i = 0; i < VECTOR\_SIZE; i++) {

        max\_delta = max(max\_delta, vec\_delta[i]);

    }

    cout << "n = " << Nt - 1 << " sigma = " << max\_delta << endl;

    cout << "Total time: " << (double)(end - start) / CLOCKS\_PER\_SEC << " sec." << endl;

    cout << "Generating plot..." << endl;

    auto plot\_file = fstream("computation.plot", ios::binary | ios::trunc | ios::out);

    if (!plot\_file.good()) {

        cerr << "Error while openning file!" << endl;

        return -1;

    }

    plot\_file.write((char\*)F0, GRID\_SIZE \* sizeof(float));

    plot\_file.close();

    system("wsl gnuplot plot\_script");  // call Linux to create plot

    \_mm\_free(F0);

    \_mm\_free(F1);

    \_mm\_free(p);

    return 0;

}

**ПРИЛОЖЕНИЕ B. Программа 2**

constexpr auto ALIGN = 32;

constexpr auto VECTOR\_SIZE = 8;

constexpr auto SHIFT1 = 1;

constexpr auto Nx = 9000;

constexpr auto Ny = 9000;

constexpr auto Nt = 111;

constexpr float Xa = 0.0f;

constexpr float Xb = 4.0f;

constexpr float Ya = 0.0f;

constexpr float Yb = 4.0f;

constexpr float hx = ((Xb - Xa) / (Nx - 1));

constexpr float hy = ((Yb - Ya) / (Ny - 1));

constexpr float coeff1 = (0.2f / ((1.0f / (hx \* hx) + 1.0f / (hy \* hy))));

constexpr float coeff2 = (0.5f \* (5.0f / (hx \* hx) - 1.0f / (hy \* hy)));

constexpr float coeff2b = (0.5f \* (5.0f / (hy \* hy) - 1.0f / (hx \* hx)));

constexpr float coeff3 = (0.25f \* (1.0f / (hx \* hx) + 1.0f / (hy \* hy)));

inline float X(size\_t j) { return (Xa + (j)\*hx); }

inline float Y(size\_t i) { return (Ya + (i)\*hy); }

constexpr float Xs1 = (Xa + (Xb - Xa) / 3.0f);

constexpr float Xs2 = (Xa + (Xb - Xa) \* 2.0f / 3.0f);

constexpr float Ys1 = (Ya + (Yb - Ya) \* 2.0f / 3.0f);

constexpr float Ys2 = (Ya + (Yb - Ya) / 3.0f);

constexpr float R = (0.1f \* ((Xb - Xa) > (Yb - Ya) ? (Yb - Ya) : (Xb - Xa)));

constexpr float GRID\_SIZE = (Nx \* Ny);

constexpr int THREADS = 8;

constexpr int CHUNK = Ny / THREADS;

inline float& get(float\* p, size\_t i, size\_t j) { return p[(i)\*Nx + (j)]; }

#include <iostream>

#include <fstream>

#include <cerrno>

#include <ctime>

#include <algorithm>

#include <immintrin.h>

#include <omp.h>

using namespace std;

/\* prepare for vectorization \*/

\_\_m256 v\_coeff1;

\_\_m256 v\_coeff2;

\_\_m256 v\_coeff2b;

\_\_m256 v\_coeff3;

\_\_m256 v\_delta;

\_\_m256 global\_v\_delta;

\_\_m256 v\_coeff4;

\_\_m256 v\_coeff5;

inline void compute\_process(float\* F0, float\* F1, float\* p, int i) {

    for(int j = 1; j < Nx / VECTOR\_SIZE - 1; j++) {

        \_\_m256 rez = \_mm256\_mul\_ps(v\_coeff1, (

                        \_mm256\_fmadd\_ps(v\_coeff3, \_mm256\_add\_ps(((\_\_m256\*)(F0 + (i - 1) \* Nx - SHIFT1))[j], ((\_\_m256\*)(F0 + (i - 1) \* Nx + SHIFT1))[j]),

                        \_mm256\_fmadd\_ps(v\_coeff2b, \_mm256\_add\_ps(((\_\_m256\*)(F0 + (i - 1) \* Nx))[j], ((\_\_m256\*)(F0 + (i + 1) \* Nx))[j]),

                        \_mm256\_fmadd\_ps(v\_coeff2, \_mm256\_add\_ps(((\_\_m256\*)(F0 + (i) \* Nx - SHIFT1))[j], ((\_\_m256\*)(F0 + (i) \* Nx + SHIFT1))[j]),

                        \_mm256\_fmadd\_ps(v\_coeff3, \_mm256\_add\_ps(((\_\_m256\*)(F0 + (i + 1) \* Nx - SHIFT1))[j], ((\_\_m256\*)(F0 + (i + 1) \* Nx + SHIFT1))[j]),

                        \_mm256\_fmadd\_ps(v\_coeff5, ((\_\_m256\*)(p + (i) \* Nx))[j],

                            \_mm256\_mul\_ps(v\_coeff4, (

                                \_mm256\_add\_ps(

                                    \_mm256\_add\_ps(

                                        \_mm256\_add\_ps(((\_\_m256\*)(p + (i - 1)\*Nx))[j], ((\_\_m256\*)(p + (i) \* Nx - SHIFT1))[j]),

                                            ((\_\_m256\*)(p + (i) \* Nx + SHIFT1))[j]),

                                                ((\_\_m256\*)(p + (i + 1) \* Nx))[j])

                                    )

                                )

                            )

                        )

                        )

                        )

                        )

                    )

                );

        \_\_m256 prev\_rez = ((\_\_m256\*)(F1 + (i) \* Nx))[j];

        v\_delta = \_mm256\_max\_ps(v\_delta, \_mm256\_max\_ps(\_mm256\_sub\_ps(prev\_rez, rez), \_mm256\_sub\_ps(rez, prev\_rez)));

        ((\_\_m256\*)(F1 + (i) \* Nx))[j] = rez;

    }

}

int main() {

    /\* allocate memory \*/

    float \*F0 = (float\*)\_mm\_malloc(GRID\_SIZE \* sizeof(float), ALIGN);

    float \*F1 = (float\*)\_mm\_malloc(GRID\_SIZE \* sizeof(float), ALIGN);

    float \*p = (float\*)\_mm\_malloc(GRID\_SIZE \* sizeof(float), ALIGN);

    if(!F0 || !F1 || !p) {

        perror("\_mm\_malloc");

        exit(errno);

    }

    /\* prepare for vectorization \*/

    v\_coeff1 = \_mm256\_set1\_ps(coeff1);

    v\_coeff2 = \_mm256\_set1\_ps(coeff2);

    v\_coeff2b = \_mm256\_set1\_ps(coeff2b);

    v\_coeff3 = \_mm256\_set1\_ps(coeff3);

    v\_delta = \_mm256\_setzero\_ps();

    v\_coeff4 = \_mm256\_set1\_ps(0.25f);

    v\_coeff5 = \_mm256\_set1\_ps(2.0f);

    clock\_t start, end;

#pragma omp parallel proc\_bind(close) num\_threads(THREADS) private(v\_delta) // OMP\_PLACES is "cores" by default

    {

        /\* init arrays \*/

#pragma omp for schedule(static, CHUNK)

        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;

            }

        }

#pragma omp barrier

        /\* compute process \*/

#pragma omp single

        start = clock();

        for (int n = 0; n < Nt - 1; n += 6) {

#pragma omp single

            global\_v\_delta = \_mm256\_setzero\_ps();

            v\_delta = \_mm256\_setzero\_ps();

            int num\_thread = omp\_get\_thread\_num();

            int shift = num\_thread \* CHUNK;

            int starting\_correction = (num\_thread == 0 ? 0 : 1);    // correction for zero thread

            int finalizing\_correction = (num\_thread == THREADS - 1 ? 0 : 1);    // correction for last thread

            for (int k = 2 - 7 \* starting\_correction; k < 5 + 2 \* (1 - starting\_correction); k++) {

                compute\_process(F0, F1, p, k + shift);

            }

            for (int k = 2 - 6 \* starting\_correction; k < 4 + 2 \* (1 - starting\_correction); k++) {

                compute\_process(F1, F0, p, k + shift);

            }

            for(int k = 2 - 5 \* starting\_correction; k < 3 + 2 \* (1 - starting\_correction); k++) {

                compute\_process(F0, F1, p, k + shift);

            }

            for (int k = 2 - 4 \* starting\_correction; k < 2 + 2 \* (1 - starting\_correction); k++) {

                compute\_process(F1, F0, p, k + shift);

            }

            for (int k = 2 - 3 \* starting\_correction; k < 1 + 2 \* (1 - starting\_correction); k++) {

                compute\_process(F0, F1, p, k + shift);

            }

            int last\_element = shift + CHUNK;

            for (int i = shift + 7 - 2 \* starting\_correction; i < last\_element - 3 - 2 \* finalizing\_correction; i++) {

                compute\_process(F0, F1, p, i);

                compute\_process(F1, F0, p, i - 1);

                compute\_process(F0, F1, p, i - 2);

                compute\_process(F1, F0, p, i - 3);

                compute\_process(F0, F1, p, i - 4);

                compute\_process(F1, F0, p, i - 5);

            }

#pragma omp barrier

            for (int k = 0; k < 1 + finalizing\_correction; k++) {

                compute\_process(F1, F0, p, last\_element - 4 - 2 \* finalizing\_correction + k);

            }

            for (int k = 0; k < 2 + 2 \* finalizing\_correction; k++) {

                compute\_process(F0, F1, p, last\_element - 5 - 2 \* finalizing\_correction + k);

            }

            for (int k = 0; k < 3 + 3 \* finalizing\_correction; k++) {

                compute\_process(F1, F0, p, last\_element - 6 - 2 \* finalizing\_correction + k);

            }

            for (int k = 0; k < 4 + 4 \* finalizing\_correction; k++) {

                compute\_process(F0, F1, p, last\_element - 7 - 2 \* finalizing\_correction + k);

            }

            for (int k = 0; k < 5 + 5 \* finalizing\_correction; k++) {

                compute\_process(F1, F0, p, last\_element - 8 - 2 \* finalizing\_correction + k);

            }

#pragma omp barrier

#pragma omp critical

            {

                global\_v\_delta = \_mm256\_max\_ps(v\_delta, global\_v\_delta);

            }

        }

#pragma omp single

        end = clock();

    }

    /\* print max delta \*/

    float max\_delta = 0.0f;

    float\* vec\_delta = (float\*)(&global\_v\_delta);

    for (int i = 0; i < VECTOR\_SIZE; i++) {

        max\_delta = max(max\_delta, vec\_delta[i]);

    }

    cout << "n = " << Nt - 1 << " sigma = " << max\_delta << endl;

    cout << "Total time: " << (double)(end - start) / CLOCKS\_PER\_SEC << " sec." << endl;

    cout << "Generating plot..." << endl;

    auto plot\_file = fstream("computation.plot", ios::binary | ios::trunc | ios::out);

    if (!plot\_file.good()) {

        cerr << "Error while openning file!" << endl;

        return -1;

    }

    plot\_file.write((char\*)F0, GRID\_SIZE \* sizeof(float));

    plot\_file.close();

    system("wsl gnuplot plot\_script");  // call Linux to create plot

    \_mm\_free(F0);

    \_mm\_free(F1);

    \_mm\_free(p);

    return 0;

}

**ПРИЛОЖЕНИЕ C. Программа 3**

constexpr auto ALIGN = 32;

constexpr auto VECTOR\_SIZE = 8;

constexpr auto SHIFT1 = 1;

constexpr auto Nx = 9000;

constexpr auto Ny = 9000;

constexpr auto Nt = 111;

constexpr float Xa = 0.0f;

constexpr float Xb = 4.0f;

constexpr float Ya = 0.0f;

constexpr float Yb = 4.0f;

constexpr float hx = ((Xb - Xa) / (Nx - 1));

constexpr float hy = ((Yb - Ya) / (Ny - 1));

constexpr float coeff1 = (0.2f / ((1.0f / (hx \* hx) + 1.0f / (hy \* hy))));

constexpr float coeff2 = (0.5f \* (5.0f / (hx \* hx) - 1.0f / (hy \* hy)));

constexpr float coeff2b = (0.5f \* (5.0f / (hy \* hy) - 1.0f / (hx \* hx)));

constexpr float coeff3 = (0.25f \* (1.0f / (hx \* hx) + 1.0f / (hy \* hy)));

inline float X(size\_t j) { return (Xa + (j)\*hx); }

inline float Y(size\_t i) { return (Ya + (i)\*hy); }

constexpr float Xs1 = (Xa + (Xb - Xa) / 3.0f);

constexpr float Xs2 = (Xa + (Xb - Xa) \* 2.0f / 3.0f);

constexpr float Ys1 = (Ya + (Yb - Ya) \* 2.0f / 3.0f);

constexpr float Ys2 = (Ya + (Yb - Ya) / 3.0f);

constexpr float R = (0.1f \* ((Xb - Xa) > (Yb - Ya) ? (Yb - Ya) : (Xb - Xa)));

constexpr float GRID\_SIZE = (Nx \* Ny);

constexpr int THREADS = 8;

constexpr int CHUNK = Ny / THREADS;

constexpr int CACHE\_LINE\_SIZE = 64; // bytes, from CPU-Z

inline int RED\_ZONE\_FLAG(int thread\_num) { return CACHE\_LINE\_SIZE \* thread\_num; }

inline int BLUE\_ZONE\_FLAG(int thread\_num) { return CACHE\_LINE\_SIZE \* thread\_num + 1; }

inline float& get(float\* p, size\_t i, size\_t j) { return p[(i)\*Nx + (j)]; }

#include <iostream>

#include <fstream>

#include <cerrno>

#include <ctime>

#include <algorithm>

#include <immintrin.h>

#include <omp.h>

using namespace std;

/\* prepare for vectorization \*/

\_\_m256 v\_coeff1;

\_\_m256 v\_coeff2;

\_\_m256 v\_coeff2b;

\_\_m256 v\_coeff3;

\_\_m256 v\_delta;

\_\_m256 global\_v\_delta;

\_\_m256 v\_coeff4;

\_\_m256 v\_coeff5;

inline void compute\_process(float\* F0, float\* F1, float\* p, int i) {

    for (int j = 1; j < Nx / VECTOR\_SIZE - 1; j++) {

        \_\_m256 rez = \_mm256\_mul\_ps(v\_coeff1, (

            \_mm256\_fmadd\_ps(v\_coeff3, \_mm256\_add\_ps(((\_\_m256\*)(F0 + (i - 1) \* Nx - SHIFT1))[j], ((\_\_m256\*)(F0 + (i - 1) \* Nx + SHIFT1))[j]),

                \_mm256\_fmadd\_ps(v\_coeff2b, \_mm256\_add\_ps(((\_\_m256\*)(F0 + (i - 1) \* Nx))[j], ((\_\_m256\*)(F0 + (i + 1) \* Nx))[j]),

                    \_mm256\_fmadd\_ps(v\_coeff2, \_mm256\_add\_ps(((\_\_m256\*)(F0 + (i)\*Nx - SHIFT1))[j], ((\_\_m256\*)(F0 + (i)\*Nx + SHIFT1))[j]),

                        \_mm256\_fmadd\_ps(v\_coeff3, \_mm256\_add\_ps(((\_\_m256\*)(F0 + (i + 1) \* Nx - SHIFT1))[j], ((\_\_m256\*)(F0 + (i + 1) \* Nx + SHIFT1))[j]),

                            \_mm256\_fmadd\_ps(v\_coeff5, ((\_\_m256\*)(p + (i)\*Nx))[j],

                                \_mm256\_mul\_ps(v\_coeff4, (

                                    \_mm256\_add\_ps(

                                        \_mm256\_add\_ps(

                                            \_mm256\_add\_ps(((\_\_m256\*)(p + (i - 1) \* Nx))[j], ((\_\_m256\*)(p + (i)\*Nx - SHIFT1))[j]),

                                            ((\_\_m256\*)(p + (i)\*Nx + SHIFT1))[j]),

                                        ((\_\_m256\*)(p + (i + 1) \* Nx))[j])

                                    )

                                )

                            )

                        )

                    )

                )

            )

            )

        );

        \_\_m256 prev\_rez = ((\_\_m256\*)(F1 + (i)\*Nx))[j];

        v\_delta = \_mm256\_max\_ps(v\_delta, \_mm256\_max\_ps(\_mm256\_sub\_ps(prev\_rez, rez), \_mm256\_sub\_ps(rez, prev\_rez)));

        ((\_\_m256\*)(F1 + (i)\*Nx))[j] = rez;

    }

}

int main() {

    /\* allocate memory \*/

    float \*F0 = (float\*)\_mm\_malloc(GRID\_SIZE \* sizeof(float), ALIGN);

    float \*F1 = (float\*)\_mm\_malloc(GRID\_SIZE \* sizeof(float), ALIGN);

    float \*p = (float\*)\_mm\_malloc(GRID\_SIZE \* sizeof(float), ALIGN);

    int \*synchro\_flags = (int\*)calloc(THREADS \* CACHE\_LINE\_SIZE, sizeof(int));

    if(!F0 || !F1 || !p || !synchro\_flags) {

        perror("\_mm\_malloc");

        exit(errno);

    }

    /\* prepare for vectorization \*/

    v\_coeff1 = \_mm256\_set1\_ps(coeff1);

    v\_coeff2 = \_mm256\_set1\_ps(coeff2);

    v\_coeff2b = \_mm256\_set1\_ps(coeff2b);

    v\_coeff3 = \_mm256\_set1\_ps(coeff3);

    v\_delta = \_mm256\_setzero\_ps();

    v\_coeff4 = \_mm256\_set1\_ps(0.25f);

    v\_coeff5 = \_mm256\_set1\_ps(2.0f);

    clock\_t start, end;

#pragma omp parallel proc\_bind(close) num\_threads(THREADS) private(v\_delta) // OMP\_PLACES is "cores" by default

    {

        /\* init arrays \*/

#pragma omp for schedule(static, CHUNK)

        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;

            }

        }

#pragma omp barrier

        /\* compute process \*/

#pragma omp single

        start = clock();

        for (int n = 0; n < Nt - 1; n += 6) {

#pragma omp single

            global\_v\_delta = \_mm256\_setzero\_ps();

            v\_delta = \_mm256\_setzero\_ps();

            int num\_thread = omp\_get\_thread\_num();

            int shift = num\_thread \* CHUNK;

            int starting\_correction = (num\_thread == 0 ? 0 : 1);    // correction for zero thread

            int finalizing\_correction = (num\_thread == THREADS - 1 ? 0 : 1);    // correction for last thread

            int value = -1;

            if (num\_thread != 0) {

#pragma omp atomic read

                value = synchro\_flags[BLUE\_ZONE\_FLAG(num\_thread - 1)];

                while (value < n) {

#pragma omp atomic read

                    value = synchro\_flags[BLUE\_ZONE\_FLAG(num\_thread - 1)];

                }

            }

            for (int k = 2 - 7 \* starting\_correction; k < 5 + 2 \* (1 - starting\_correction); k++) {

                compute\_process(F0, F1, p, k + shift);

            }

            for (int k = 2 - 6 \* starting\_correction; k < 4 + 2 \* (1 - starting\_correction); k++) {

                compute\_process(F1, F0, p, k + shift);

            }

            for (int k = 2 - 5 \* starting\_correction; k < 3 + 2 \* (1 - starting\_correction); k++) {

                compute\_process(F0, F1, p, k + shift);

            }

            for (int k = 2 - 4 \* starting\_correction; k < 2 + 2 \* (1 - starting\_correction); k++) {

                compute\_process(F1, F0, p, k + shift);

            }

            for (int k = 2 - 3 \* starting\_correction; k < 1 + 2 \* (1 - starting\_correction); k++) {

                compute\_process(F0, F1, p, k + shift);

            }

            int last\_element = shift + CHUNK;

            for (int i = shift + 7 - 2 \* starting\_correction; i < last\_element - 3 - 2 \* finalizing\_correction; i++) {

                compute\_process(F0, F1, p, i);

                compute\_process(F1, F0, p, i - 1);

                compute\_process(F0, F1, p, i - 2);

                compute\_process(F1, F0, p, i - 3);

                compute\_process(F0, F1, p, i - 4);

                compute\_process(F1, F0, p, i - 5);

            }

#pragma omp atomic write

            synchro\_flags[RED\_ZONE\_FLAG(num\_thread)] = n + 6;

            if (num\_thread != THREADS - 1) {

#pragma omp atomic read

                value = synchro\_flags[RED\_ZONE\_FLAG(num\_thread + 1)];

                while (value < n) {

#pragma omp atomic read

                    value = synchro\_flags[RED\_ZONE\_FLAG(num\_thread + 1)];

                }

            }

            for (int k = 0; k < 1 + finalizing\_correction; k++) {

                compute\_process(F1, F0, p, last\_element - 4 - 2 \* finalizing\_correction + k);

            }

            for (int k = 0; k < 2 + 2 \* finalizing\_correction; k++) {

                compute\_process(F0, F1, p, last\_element - 5 - 2 \* finalizing\_correction + k);

            }

            for (int k = 0; k < 3 + 3 \* finalizing\_correction; k++) {

                compute\_process(F1, F0, p, last\_element - 6 - 2 \* finalizing\_correction + k);

            }

            for (int k = 0; k < 4 + 4 \* finalizing\_correction; k++) {

                compute\_process(F0, F1, p, last\_element - 7 - 2 \* finalizing\_correction + k);

            }

            for (int k = 0; k < 5 + 5 \* finalizing\_correction; k++) {

                compute\_process(F1, F0, p, last\_element - 8 - 2 \* finalizing\_correction + k);

            }

#pragma omp atomic write

            synchro\_flags[BLUE\_ZONE\_FLAG(num\_thread)] = n + 6;

#pragma omp critical

            {

                global\_v\_delta = \_mm256\_max\_ps(v\_delta, global\_v\_delta);

            }

        }

#pragma omp single

        end = clock();

    }

    /\* print max delta \*/

    float max\_delta = 0.0f;

    float\* vec\_delta = (float\*)(&global\_v\_delta);

    for (int i = 0; i < VECTOR\_SIZE; i++) {

        max\_delta = max(max\_delta, vec\_delta[i]);

    }

    cout << "n = " << Nt - 1 << " sigma = " << max\_delta << endl;

    cout << "Total time: " << (double)(end - start) / CLOCKS\_PER\_SEC << " sec." << endl;

    cout << "Generating plot..." << endl;

    auto plot\_file = fstream("computation.plot", ios::binary | ios::trunc | ios::out);

    if (!plot\_file.good()) {

        cerr << "Error while openning file!" << endl;

        return -1;

    }

    plot\_file.write((char\*)F0, GRID\_SIZE \* sizeof(float));

    plot\_file.close();

    system("wsl gnuplot plot\_script");  // call Linux to create plot

    \_mm\_free(F0);

    \_mm\_free(F1);

    \_mm\_free(p);

    return 0;

}