МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №3

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 оптимизировать использование памяти в простых программах 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.

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: **Microsoft (R) C/C++ Optimizing Compiler Version 19.25.28614 for x64;**

   2.1.3. Ключи компиляции (наиболее важные): **/O2 /arch:AVX2;**

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

   2.2.Времена работы программы:

Количество вычисляемых временных шагов для одних и тех же данных, кол-во:	**Время, сек.:**
1(до оптимизации)	6.802
2	5.825 (удалось догнать авто-векторизованную версию)
3	5.628
4	5.253
5	5.247
6	5.082
7	5.176

   2.3.Листинг самой быстрой версии программы **(см. приложение 4.)**.

2.4.Производительность **(в сравнении с AVX2 + FMA и авто-векторизованной версиями из прошлой лабораторной работы)**

2.4.1.1. Число инструкций на такт: 1.51 (в 1.54 раз лучше версии до оптимизации, но примерно такое же, как и у авто-векторизованной). Процессор меньше тактов ждет данные для исполнения команд, потому что улучшилась локальность данных;

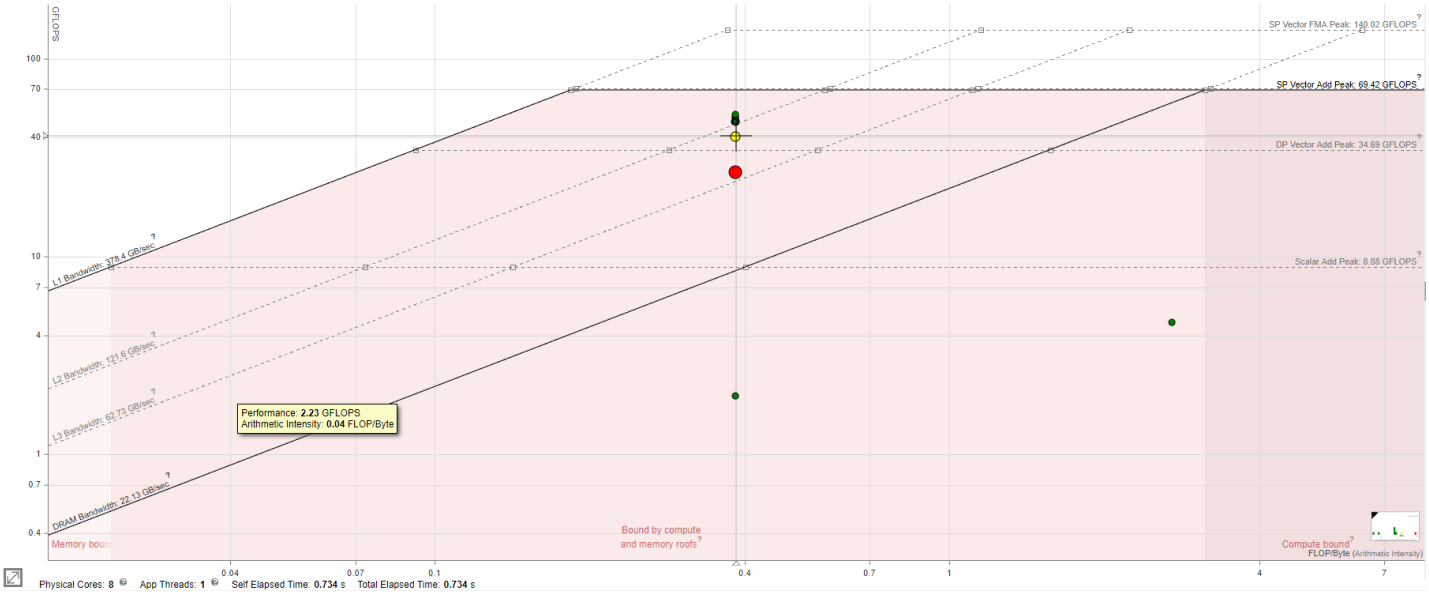
2.4.1.2. Процент кэш-промахов для кэша 3 уровня: 19.76% (в 1.87 раз лучше версии до оптимизации, но такое же, как и у авто-векторизованной (на 3% лучше)). Как и ожидалось, при более частом использовании одни и тех же данных из кэша, кол-во промахов уменьшается.

2.4.1.3. Процент кэш-промахов для кэша 1 уровня: 1.96% (практически не изменилось по сравнению с версией до оптимизации и авто-векторизованной версии);

2.4.1.4. Процент неправильно предсказанных переходов: 1.68% (практически не изменилось по сравнению с версией до оптимизации и авто-векторизованной версии);

2.4.1.5. Время работы: 5.082 сек. (в 1.33 раз лучше версии до оптимизации и в 1.14 лучше авто-векторизованной). Ускорение сравнимо с уменьшением количества промахов при доступе к LLC.

2.5.Roofline-модель с точками, соответствующими основному циклу программы для наиболее быстрого варианта программы:



Цикл программы для расчетной сетки по оси Y содержит в себе несколько циклов по оси X для разных временных шагов. На модели видны точки, соответствующие этим циклам. Красной и самой жирной точке соответствует первый подцикл по X (он в самом начале после входа в тело цикла по Y). В этот момент данные только выбираются в кэш и в этом месте нет никакого прироста по сравнению с версией до оптимизации (точка почти не сдвинулась с места). Однако, появились желтые и зеленые точки, соответствующие дальнейшим подциклам по X. Они используют уже выбранные в кэш данные на предыдущем цикле и скорость работы с памятью в них вплотную приближается к пропускной способности кэша 2 уровня, а для некоторых циклов превосходит её. Арифметическая интенсивность выросла, потому что теперь часть основного цикла работает с памятью быстрее.

3. ВЫВОДЫ:

3.1. Научились оптимизировать использование памяти в простых программах численного моделирования;

3.2. Проанализировали полученные результаты.

4. ПРИЛОЖЕНИЕ:

```
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);

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 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) * Nx - SHIFT1))[j], ((__m256*)(F0 + (i)*Nx +
SHIFT1))[j]),
					_mm256_fmadd_ps(v_coeff2b,
_mm256_add_ps(((__m256*)(F0 + (i) * Nx))[j], ((__m256*)(F0 + (i + 2) * Nx))[j]),
					_mm256_fmadd_ps(v_coeff2,
_mm256_add_ps(((__m256*)(F0 + (i + 1) * Nx - SHIFT1))[j], ((__m256*)(F0 + (i + 1) * Nx +
SHIFT1))[j]),
					_mm256_fmadd_ps(v_coeff3,
_mm256_add_ps(((__m256*)(F0 + (i + 2) * Nx - SHIFT1))[j], ((__m256*)(F0 + (i + 2) * Nx +
SHIFT1))[j]),

					_mm256_fmadd_ps(v_coeff5, ((__m256*)(p + (i + 1)
* Nx))[j],
						_mm256_mul_ps(v_coeff4, (
							_mm256_add_ps(
								_mm256_add_ps(

	_mm256_add_ps(((__m256*)(p + (i)*Nx))[j], ((__m256*)(p + (i + 1) * Nx -
SHIFT1))[j]),
											((__m256*)(p +
(i + 1) * Nx + SHIFT1))[j]),

	((__m256*)(p + (i + 2) * Nx))[j])
											)
										)
									)
								)
							)
						)
```

```
                        );

                __m256 prev_rez = ((__m256*)(F1 + (i + 1) * 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 + 1) * 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);
        }

        /* 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 */
        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);

        /* compute process */
        clock_t start, end;

        start = clock();
        for(int n = 0; n < Nt - 1; n += 6) {
                v_delta = _mm256_setzero_ps();

                for (int k = 1; k <= 5; k++) {
                        compute_process(F0, F1, p, k);
                }

                for (int k = 1; k <= 4; k++) {
                        compute_process(F1, F0, p, k);
```

```
            }

            for (int k = 1; k <= 3; k++) {
                  compute_process(F0, F1, p, k);
            }

            for (int k = 1; k <= 2; k++) {
                  compute_process(F1, F0, p, k);
            }

            compute_process(F0, F1, p, 1);

            for (int i = 6; i < Ny - 3; 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);
            }

            compute_process(F1, F0, p, Ny - 3);

            for (int k = 0; k < 2; k++) {
                  compute_process(F0, F1, p, Ny - 4 + k);
            }

            for (int k = 0; k < 3; k++) {
                  compute_process(F1, F0, p, Ny - 5 + k);
            }

            for (int k = 0; k < 4; k++) {
                  compute_process(F0, F1, p, Ny - 6 + k);
            }

            for (int k = 0; k < 5; k++) {
                  compute_process(F1, F0, p, Ny - 7 + k);
            }
            // don’t need swap F0 and F1, because new data already in F0 array
      }
      end = clock();

      /* 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]);
      }
      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;
}
```