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.

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))**.
      2. Компилятор: **icc (ICC) 19.1.0.166 20191121;**
      3. Ключи компиляции: **-O2 -ip -xcoffeelake -axcoffeelake;**
      4. Параметры программы: **Nx = Ny = 9000, Nt = 110**.
   2. Программа, векторизованная с помощью компилятора**:**
      1. Текст программы: **(см. приложение 4.1)**;
      2. Отчет компилятора о векторизации:



* + 1. Производительность:
       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. **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.
       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.

* + - 1. **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. **Время работы:** 5.82 сек. (в 3.26 раз лучше скалярной версии).
  1. Ручная векторизация программы:
     1. Итоговый текст программы: **(см. приложение 4.2)**;
     2. Этапы векторизации:

| Этап, тип | Время, сек. |
| --- | --- |
| AVX2 | 6.41 |
| AVX2 + FMA | 6.66 |

* 1. Производительность:
     1. Все характеристики оказались примерно равными характеристикам авто-векторизованной версии, кроме бОльшего количества кэш-промахов LLC (39.42%).
     2. Видимо из-за этих промахов программа с ручной векторизацией оказалась медленнее авто-векторизованной версии.
  2. **Roofline-модель с точкой, соответствующей основному циклу программы (красная точка посередине) для наиболее быстрого варианта векторизованной программы (авто-векторизованная версия):**

**A screenshot of a computer

Description automatically generated**

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.**

1. **ВЫВОДЫ:**
   1. Научились векторизовать простые программы численного моделирования;
   2. Авто-векторизация дает хороший прирост в производительности (в моем случае так вообще лучший) без приложения особых усилий.
2. **ПРИЛОЖЕНИЕ:**
   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;

}

* 1. 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;

}