بسم الله الرحمن الرحیم

گزارش تمرین کامپیوتری شماره سه درس برنامه سازی موازی

انجام دهندگان:

علیرضا زارع نژاد – 810196474

محمد معین شفی – 810196492

پاییز 1399

برای کامپایل کدها کافیست اسکریپت install.sh را اجرا نمایید. بعد از اجرای آن دو پرونده‌ی part1.out و part2.out ایجاد می‌شوند که با اجرای آن ها می‌توانید خروجی‌های برنامه‌ها را مشاهده نمایید.

سوال 1

در این سوال برای کارکردن با داده های تصویر از کتابخانه opencv استفاده گردیده است که امکانات زیادی برای کار کردن با داده های چندرسانه ای در اختیار ما قرار داده است.

در این بخش می خواهیم عمل قدر مطلق تفاضل را برای هر پیکسل محاسبه کنیم و در تصویر نهایی بنویسیم.

ابتدا کتابخانه های مرتبط را include می کنیم.

```
art1.cpp > ...
    #include <iostream>

    #include <opencv2/opencv.hpp>
    #include <opencv2/highgui.hpp>
    #include "ipp.h"

    using namespace std;
    using namespace cv;
```

دو تصویر به ما داده می شود که در دایرکتوری اصلی برنامه قرار گرفته اند. به کمک define مسیر یا به عبارتی اسم آن ها را مشخص می کنیم.

```
#define FIRST_IMAGE_PATH "CA03__Q1__Image__01.png"
#define SECOND_IMAGE_PATH "CA03__Q1__Image__02.png"
```

ابتدا حالت سریال را توضیح می دهیم. کد در زیر آورده شده است.

سریال

```
Mat first_image, second_image, diff_image;
Ipp64u start, end;
first_image = imread(FIRST_IMAGE_PATH, IMREAD_GRAYSCALE);
second_image = imread(SECOND_IMAGE_PATH, IMREAD_GRAYSCALE);
if(first_image.empty())
{
    cout << "couldn't open the first image" << endl;
    return -1;
}
if(second_image.empty())
{
    cout << "couldn't open the second image" << endl;
    return -1;
}
int NROWS = first_image.rows;
int NCOLS = first_image.cols;

diff_image.create(NROWS, NCOLS, CV_8UC1);
```

متغیر های first_image و second_image و diff_image از نوع Mat را تعریف کردیم و به کمک تابع imread تصویر اول و دوم را در حالت GRAYSCALE ذخیره می کنیم. در صورتی که عمل خواندن با خطا مواجه شود با پیغام مناسب خروجی می دهیم.

تعداد سطر ها و ستون های دو تصویر با یکدیگر برابر هستند. آن ها را در NROWS و NCOLS ذخیره می کنیم.

در گام بعدی diff_image را با تابع create و مشخص شدن سطر ها و ستون های تصویر خروجی آماده می کنیم.

```
uint8_t* first_image_data = (uint8_t*)first_image.data;
uint8_t* second_image_data = (uint8_t*) second_image.data;
uint8_t* diff_image_data = (uint8_t*)diff_image.data;

start = ippGetCpuClocks();
for(int i = 0; i < NROWS; i++)
    for(int j = 0; j < NCOLS; j++)
    {
        int index = i * NCOLS + j;
        diff_image_data[index] = abs(second_image_data[index] - first_image_data[index]);
    }
end = ippGetCpuClocks();
int serial_time = (Ipp32u)(end - start);
cout << "Serial takes " << serial_time << " clock cycles"<< endl;
imwrite("q1_serial.png", diff_image);
return serial_time;
}
```

حال سه متغیر first_image_data و second_image_data و diff_image_data را که پوینتری از unit8_t هستند را تعریف کرده و هر کدام را با دیتای تصویر مرتبط پر نماییم. هدف این است که از روی دیتای پیکسل های تصویر اول و دوم ، پیکسل متناظر تصویر خروجی را پر کنیم.

متغیر start را قبل از شروع حلقه مقدار دهی می کنیم که زمان شروع را مشخص می کند.

حال کافی است یک حلقه تو در تو داشته باشیم. حلقه ی اول روی سطر های پیش می رود و حلقه ی دوم روی ستون ها.

فرض کنیم که i نشان دهنده سطر و j نشان دهنده ستون باشد.می دانیم که آرایه های دو بعدی بدین صورت ذخیره می شوند که ایتدا تمامی اعضای سطر اول به صورت ستونی زیر هم قرار می گیرند و سپس اعضای سطر دوم زیر آن ها و الی آخر. فلذا اندیس پوینتر ما به اندازه حاصل ضرب شماره سطر و تعداد ستون ها به علاوه ستون جاری از شروع داده خواهد بود.

در اینجا کافی است از تابع abs استفاده کرده و مقدار تفاصل را به دست بیاوریم.

پس از اتمام حلقه مقدار end را خاتمه کار هست را مقدار هی می کنیم.

مدت زمان اجرا برنامه در حقیقت فاضل مقدار end و start خواهد بود.

در انتها تصویر خروجی را در فایل q1_serial.png ذخیره می کنیم.

پارالل

```
__m128i* first_image_data = (__m128i*) first_image.data;
__m128i* second_image_data = (__m128i*) second_image.data;
__m128i* diff_image_data = (__m128i*) diff_image.data;

__m128i m1, m2, sub0, sub1, abs;
int index = 0;

start = ippGetCpuClocks();
for(int i = 0; i < NROWS; i++)
    for(int j = 0; j < NCOLS/16; j++)
    {
        index = i * (NCOLS/16) + j;
        m1 = _mm_loadu_si128((__m128i*)(first_image_data + index));
        m2 = _mm_loadu_si128((__m128i*)(second_image_data + index));
        sub0 = _mm_subs_epu8(m1, m2);
        sub1 = _mm_subs_epu8(m2, m1);
        abs = _mm_or_si128(sub0, sub1);
        _mm_store_si128((__m128i*)(diff_image_data + index), abs);
    }
end = ippGetCpuClocks();
int parallel_time = (Ipp32u)(end - start);
cout << "Parallel takes " << parallel_time << " clock cycles"<< endl;
imwrite("q1_parallel.png", diff_image);
return parallel_time;
```

در این جا سه متغیر first_image_data و second_image_data و diff_image_data را از نوع تایپ __m128i به عنوان پوینتر نگه داری میکنیم که هر کدام اشاره به محل شروع دیتا تصاویر می کنند.

در این بخش حلقه ای اول ما همچنان یکی یکی اضافه خواهد شد ولی برای ستون ها چون هر بار load کردن سبب می شود 16 مقدار 8 بیتی داخل رجیستر ها قرار بگیرد، شرط پایان حلقه ستون ها را برابر با رسیدن j به تعداد ستون ها تقسیم بر 16 قرار دادیم.

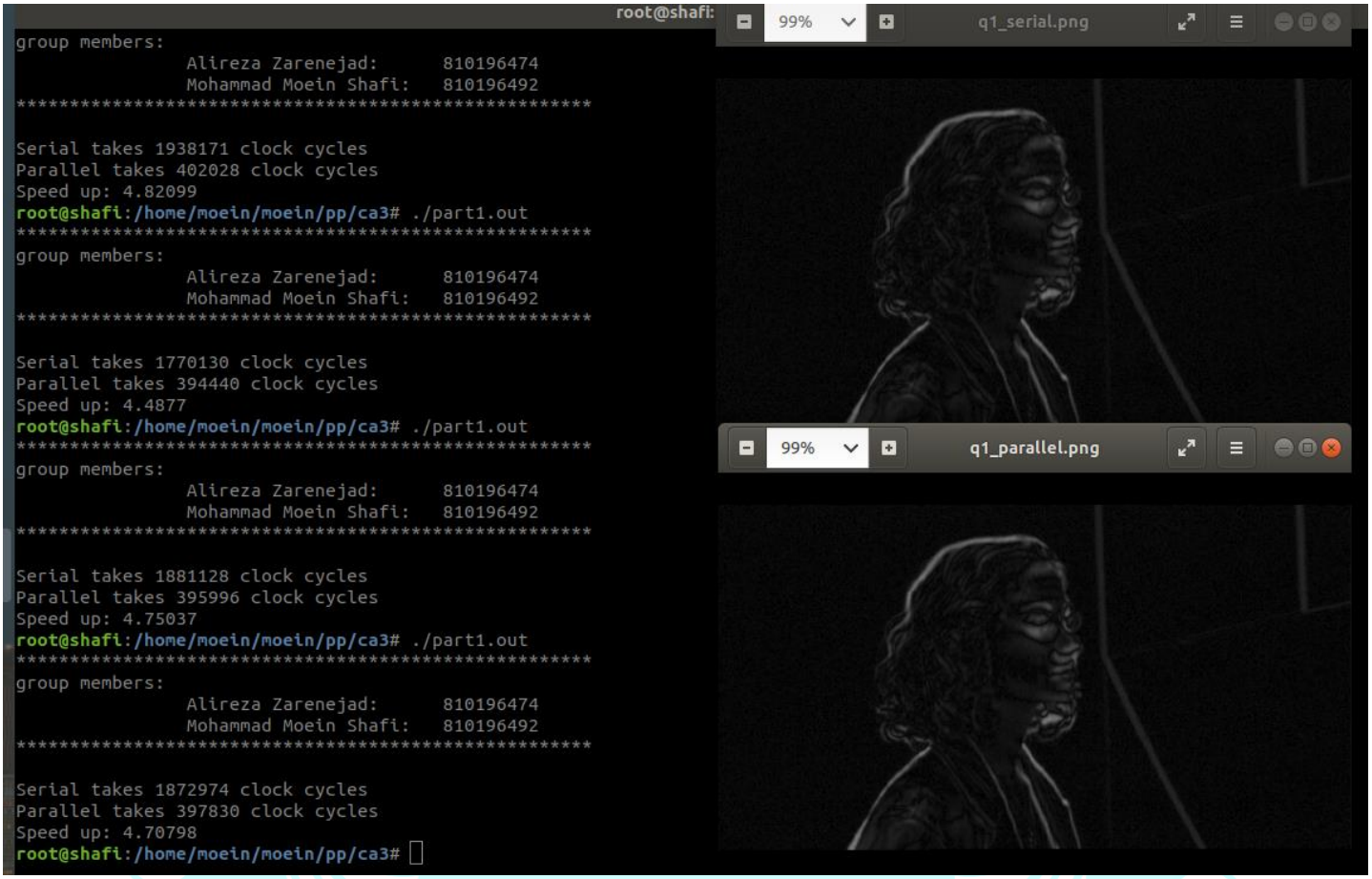
ابتدا با دستور _mm_loadu_si128 از شروع داده به علاوه 16 ، index مقدار جدید 8 بیتی را درون رجیستر m1 لود می کنیم. به صورت مشابه m2 را برای تصویر دوم می سازیم.

ابتدا آرایه اول را از آرایه دوم کم کردیم (در حالت saturation) و یکبار آرایه دوم را از آرایه اول کم کردیم (بار هم در حالت saturation) . و در نهایت حاصل را or کردیم .نتیجه ای که منفی شده بود برابر صفر شده است(به علت حالت saturation) و بنابر این با or کردن خروجی مثبت به دست می آید. سپس با دستور _mm_store_si128 مقدار جدید را در پیکسل متناظر تصویر خروجی ذخیره می کنیم.

در انتها تصویر را با نام q1_parallel.png ذخیره می کنیم.

پس از اجرا تصویری مشابه زیر به دست خواهد آمد.

و مقدار speed up طی چند اجرا بدین صورت است (تقریبا برابر با 4.7):



سوال 2

موارد مربوط به اضافه کردن کتابخانه‌ها همانند بخش یک می‌باشد.

به علت اینکه اندازه تصاویر داده شده مضرب 16 نبودند ، آن‌ها را به برش زدیم تا مضرب 16 شوند. این برش زدن در تصاویر داده اختلالی ایجاد نکرد. تصاویر جدید با پسوند 16x_ ذخیره شدند.

```
#define FIRST_IMAGE_PATH "CA03__Q2__Image__01_16x.png"
#define SECOND_IMAGE_PATH "CA03__Q2__Image__02_16x.png"
```

ابتدا به توضیح بخش سریال می‌پردازیم:

روند کلی کار به این صورت است که ابتدا تصویر نهایی را برابر با تصویر اول قرار دادیم ، سپس در آن اندازه‌هایی (شماره سطر و ستون) که تصویر دوم موجود بود، تصویر دوم را با فرمول داده شده بر روی تصویر اول اضافه کردیم.

به این منظور ابتدا متغیرهای first_image و second_image و result_image را به صورت زیر تعریف نمودیم:

```
Mat first_image, second_image, result_image;
Ipp64u start, end;
first_image = imread(FIRST_IMAGE_PATH, IMREAD_GRAYSCALE);
result_image = imread(FIRST_IMAGE_PATH, IMREAD_GRAYSCALE);
second_image = imread(SECOND_IMAGE_PATH, IMREAD_GRAYSCALE);
if(first_image.empty())
{
    cout << "couldn't open the first image" << endl;
    return -1;
}
if(second_image.empty())
{
    cout << "couldn't open the second image" << endl;
    return -1;
}
```

همانطور که گفته شد، مقدار سطر و ستون‌های دو تصویر با هم برابر نیستند. بنابراین آن‌ها در متغیرهای زیر ذخیره کردیم:

```
int NROWS = first_image.rows;
int NCOLS = first_image.cols;
int NROWS_2 = second_image.rows;
int NCOLS_2 = second_image.cols;
```

حال سه متغیر first_image_data و second_image_data و diff_image_data را که پوینتری از unit8_t هستند را تعریف کرده و هر کدام را با دیتای تصویر مرتبط پر نماییم. هدف این است که از روی دیتای پیکسل های تصویر اول و دوم ، پیکسل متناظر تصویر خروجی را پر کنیم.

```
uint8_t* first_image_data = (uint8_t*)first_image.data;
uint8_t* result_image_data = (uint8_t*)result_image.data;
uint8_t* second_image_data = (uint8_t*)second_image.data;
```

حال کافی است یک حلقه تو در تو داشته باشیم. حلقه ی اول روی سطر های پیش می رود و حلقه ی دوم روی ستون ها. همانطور که گفته شد، تنها روی سطرها و ستون‌های تصویر دوم پیش می‌رویم. در اینجا نیاز است تا طبق فرمول داده شده، تصویر دوم را در 0.5 ضرب نموده و با تصویر اول جمع کنیم. از آنجا که ضرب در 0.5 معادل یک شیفت به راست است، از شیفت استفاده می‌کنیم.

فرض کنیم که i نشان دهنده سطر و j نشان دهنده ستون باشد.می دانیم که آرایه های دو بعدی بدین صورت ذخیره می شوند که ابتدا تمامی اعضای سطر اول به صورت ستونی زیر هم قرار می گیرند و سپس اعضای سطر دوم زیر آن ها و الی آخر. فلذا اندیس پوینتر ما به اندازه حاصل ضرب شماره سطر و تعداد ستون ها به علاوه ستون جاری از شروع داده خواهد بود.

متغیر start را قبل از شروع حلقه مقدار دهی می کنیم که زمان شروع را مشخص می کند.

```
start = ippGetCpuClocks();
for(int i = 0; i < NROWS_2; i++)
    for(int j = 0; j < NCOLS_2; j++)
        {
            int index = i * NCOLS + j;
            int index2 = i * NCOLS_2 + j;
            result_image_data[index] = first_image_data[index] + second_image_data[index2] >> 1;
        }
end = ippGetCpuClocks();
```

در انتها نیز زمان اجرا به سادگی محاسبه شده و تصویر نهایی را در پرونده‌ی q2_serial.png ذخیره می‌کنیم.

```
int serial_time = (Ipp32u)(end - start);
cout << "Serial takes " << serial_time << " clock cycles"<< endl;
imwrite("q2_serial.png", result_image);
return serial_time;
```

اکنون به توضیح روش موازی می‌پردازیم:

موارد مربوط به تعریف متغیرهای first_image و second_image و result_image و ذخیره‌ی تعداد ستون‌ها و سطرها به مانند قسمت سریال صورت می‌گیرد. در این جا سه متغیر first_image_data و second_image_data و result_image_data را از نوع تایپ __m128i به عنوان پوینتر نگه داری میکنیم که هر کدام اشاره به محل شروع دیتا تصاویر می کنند.

```
Mat first_image, second_image, result_image;
Ipp64u start, end;
first_image = imread(FIRST_IMAGE_PATH, IMREAD_GRAYSCALE);
result_image = imread(FIRST_IMAGE_PATH, IMREAD_GRAYSCALE);
second_image = imread(SECOND_IMAGE_PATH, IMREAD_GRAYSCALE);
if(first_image.empty())
{
    cout << "couldn't open the first image" << endl;
    return -1;
}
if(second_image.empty())
{
    cout << "couldn't open the second image" << endl;
    return -1;
}
int NROWS = first_image.rows;
int NCOLS = first_image.cols;
int NROWS_2 = second_image.rows;
int NCOLS_2 = second_image.cols;

__m128i* first_image_data = (__m128i*) first_image.data;
__m128i* result_image_data = (__m128i*) result_image.data;
__m128i* second_image_data = (__m128i*) second_image.data;
```

در این بخش حلقه ای اول ما همچنان یکی یکی اضافه خواهد شد ولی برای ستون ها چون هر بار load کردن سبب می شود 16 مقدار 8 بیتی داخل رجیستر ها قرار بگیرد، شرط پایان حلقه ستون ها را برابر با رسیدن j به تعداد ستون ها تقسیم بر 16 قرار دادیم.

ابتدا با دستور _mm_loadu_si128 از شروع داده به علاوه index ، 16 مقدار جدید 8 بیتی را درون رجیستر m1 لود می کنیم. به صورت مشابه m2 را برای تصویر دوم می سازیم. سپس مقادیر متغیر m2 را، همانطور که در قسمت سریال توضیح داده شد، شیفت به راست می‌دهیم. حال در اینجا نیاز است تا آخرین بیت را که اضافه شده، از بین ببریم، به این منظور کل متغیر m2 را با مقدار 01111111 (که همان 127 است) باید and کنیم. برای ذخیره‌ی این مقدار از متغیر shift استفاده شده است. در نهایت نتیجه را با m1 جمع می‌کنیم و در m1 ذخیره می‌کنیم. لازم به ذکر است برای استفاده کمتر از تعریف متغیرها، از تعریف متغیر جدیدی برای ذخیره‌ی مقدار حاصلجمع جلوگیری شد تا رجیسترهای کمتری مصرف شود.

```
start = ippGetCpuClocks();
__m128i shift = _mm_set1_epi8(127);
__m128i m1, m2;
int index, index2;
for(int i = 0; i < NROWS_2; i++)
    for(int j = 0; j < NCOLS_2 / 16; j++)
    {
        index = i * (NCOLS / 16) + j;
        index2 = i * (NCOLS_2 / 16) + j;

        m1 = _mm_loadu_si128(first_image_data + index);
        m2 = _mm_loadu_si128(second_image_data + index2);
        m2 = m2 >> 1;
        m2 = _mm_and_si128(shift, m2);
        m1 = _mm_adds_epu8(m1, m2);
        _mm_store_si128(result_image_data + index, m1);
    }

end = ippGetCpuClocks();
```

در انتها نیز زمان اجرا به سادگی محاسبه شده و تصویر نهایی را در پرونده‌ی q2_parallel.png ذخیره می‌کنیم.

```
int parallel_time = (Ipp32u)(end - start);
cout << "Parallel takes " << parallel_time << " clock cycles"<< endl;
imwrite("q2_parallel.png", result_image);
return parallel_time;
```

پس از اجرا تصویری مشابه زیر به دست خواهد آمد.

و مقدار speed up طی چند اجرا بدین صورت است (تقریبا برابر با 4.2):