تمرین اول: برنامه نویسی موازی با کمک دستورات SIMD

محمد هاشمی ۸۱۰۱۹۷۴۲۳ md.hashemi@ut.ac.ir
شاهین منتظری ۸۱۰۱۹۷۳۹۰ shahin.montazeri@ut.ac.ir

چکیده

در این تمرین برنامه‌های کامپیوتری سه برنامه به صورت سریال و موازی نوشته شده است که یکی از آن‌ها برای پیدا کردن بیشترین مقدار و دو برنامه دیگر برای پردازش تصویر با الگوریتم‌های Sobel و Stereo Vision می‌باشد. در برنامه پیدا کردن بیشترین مقدار، ورودی‌های ما اعداد ممیز شناور با دقت ساده هستند. الگوریتم Sobel یکی از انواع روش‌های مناسب به منظور تشخیص لبه‌های تصویر می‌باشد و الگوریتم Stereo Vision برای پیدا کردن عمق تصویر استفاده می‌شود. ابتدا برنامه‌های کامپیوتری را با زبان C++ (سریال) نوشته و در محیط Visual Studio اجرا شدند؛ همچنین از کتابخانه OpenCV به دلیل سهولت در کار با (خواندن و نوشتن) تصاویر با فرمت‌های مختلف نیز استفاده شده است. سپس برنامه کامپیوتری موازی با استفاده از دستورهای SIMD مربوط به سری‌های مختلف SSE شرکت Intel نوشته شد که همان الگوریتم‌های Sobel و تشخیص عمق تصویر را به صورت موازی پیاده سازی شدند. در آخر با مقایسه زمان اجرای این برنامه‌ها، میزان بهبود عملکرد پیاده سازی موازی نسبت به پیاده سازی سریال مشخص می شود.

کلمات کلیدی

C++ ، OpenCV ، SIMD ، Image Processing ، Sobel Operator ، Image Edge Detection ، Stereo Vision .

۱- مقدمه

پردازش تصویر یا Image Processing ، امروزه به عنوان یکی از مولفه های اساسی در سیستم های هوشمند و پشتیبان تصمیم است، که غالبا بر روی تصاویر دیجیتال و توسط سیستم های کامپیوتری اعمال می شود. کاربردهای متنوعی که پردازش تصویر در زمینه های مختلف فنی، صنعتی، شهری، پزشکی و علمی دارد، آن را به یک موضوع بسیار فعال در میان زمینه های پژوهشی تبدیل کرده است.

OpenCV (بینایی ماشین متن باز) یک کتابخانه متن باز شامل بیش از صدها الگوریتم بهینه سازی شده به زبان C و C++ برای تحلیل تصویر و ویدیو است، که از زمان معرفی آن در سال ۱۹۹۹، به میزان زیادی از سوی جامعه محققین و توسعه دهندگان بینایی ماشین به عنوان ابزار توسعه پایه پذیرفته شده است. OpenCV در ابتدا در اینتل به منظور توسعه تحقیقات در زمینه بینایی ماشین و ارتقا کاربردهایی که شدیدا از پردازنده استفاده می کنند، توسعه داده شد. مزیت اصلی OpenCV ، در سرعت اجرای آن به خصوص در کاربردهای بی درنگ و البته متن باز بودن و رایگان بودن آن است. این مجموعه آموزشی تلاشی است برای آشنایی هر چه بیشتر جامعه محققین بینایی ماشین با این کتابخانه ارزشمند، که به صورت گام به گام و عملی همراه با مجموعه متنوعی از مثال ها، شما را برای توسعه ی برنامه های کاربردی خود آماده خواهد ساخت.

در این تمرین هدف پیاده سازی دو الگورتیم مهم Sobel و Stereo Vision در امور پردازش تصویر با استفاده از روش های برنامه های برنامه نویسی سریال و موازی است و مقایسه آن دو از نظر زمان اجرا یا Performance می باشد.

در پیاده سازی این دو الگوریتم از روش های برنامه نویسی سریال یا همان C++ و برنامه نویسی موازی یا SIMD (Single instruction multiple data) بهره برده ایم.

در روش سریال که همان C++ کلاسیک می باشد برنامه نوشته شده به صورت ترتیبی اجرا می شود و در حالی که اپراتور های ریاضیاتی مختلف در آن مانند ضرب، جمع، تفریق و دارای تعداد بیت محدود و با دقتی معین می باشد. در این روش، نوشتن برنامه نسبتا آسان تر از SIMD می باشد اما به دلیل کوچکتر بودن طول داده نیاز به تکرار بیشتر به منظور انجام تمام محاسبات ظاهر می شود و همین امر باعث افزایش مدت زمان لازم برای اجرای برنامه می شود. اما در مقابل در روش SIMD،

طول داده چندین برابر بزرگتر از حالت سریال می‌باشد و این امر باعث کاهش تعداد تکرارهای مربوط به حلقه های درون برنامه و در ادامه کاهش زمان اجرای برنامه می شود؛ زیرا با استفاده از یک دستور در یک تعداد سیکل برابر می توان داده های بیشتری را محاسبه نمود. اما واضح است که نیاز به طراحی خاص در پردازنده موردنظر آشکار می شود؛ بدین منظور شرکت intel اقدام به طراحی و تولید پردازنده های خاص با طول داده زیاد در معماری واحد های Arithmetic و حتی instruction Register خود نموده است. نتیجه این اقدامات باعث شده است که از حدود دهه ۹۰ میلادی نسل های مختلف پردازنده ها مانند Pentium ها و Corei ها دارای طراحی لازم بدین منظور شده اند و اکنون که اکثر پردازنده ها از سری های Corei5 و Corei7 هستند تقریبا تمامی دستور های مربوط به SIMD را Support می کنند. ما نیز در این پروژه با بهره گیری از هر دو روش برنامه نویسی C++ و SIMD الگوریتم های مزبور را پیاده سازی نموده و عملکرد آن ها را مقایسه می کنیم.

۲- پیدا کردن بیشترین مقدار

در تمرین اول یک آرایه ی ۱۰۰۰۰ خانه ای را انتخاب کرده و به صورت رندوم مقادیر ممیز شناور (flout) می دهیم. سپس به صورت سریال و با کمک زبان C بیشترین مقدار آن را پیدا کرده زمان انجام محاسبات را به دست می آوریم. سپس همین برنامه را با کمک SIMD به صورت موازی پیاده سازی می کنیم و زمان اجرای آن را نیز به دست می آوریم. مشاهده می کنیم که جواب های نهایی یکی هستند که نشان از درست بودن برنامه موازی شده از جهت عملیاتی است و مشاهده می کنیم که زمان اجرا کمتر از حالت قبل می شود. اگر برنامه چند بار اجرا کنیم مشاهده می کنیم که زمان اجرا تغییر می کند. ما این برنامه را ده بار محاسبه کرده و به طور متوسط زمان اجرا حدود ۳ برابر بهبود می یابد. در شکل ۱ نتایج حاصل از تعدادی از تکرارهای اجرای برنامه را مشاهده می کنیم.

```
max of value with serial way= 99.993889
Time for Serial = 27052

max of value with Parallel way= 99.993889
Time for Parallel = 12841

Speedup = 2.106689
```

```
max of value with serial way= 99.993889
Time for Serial = 26040

max of value with Parallel way= 99.993889
Time for Parallel = 6818

Speedup = 3.819302
```

شکل (۱): نتایج برنامه پیدا کردن بیشترین مقدار

زمان اجراهای برنامه را در جدول ۱ مشاهده فرمایید.

جدول (۱): زمان اجرای برنامه بیشترین مقدار

زمان اجرای سریال	زمان اجرای موازی	میزان بهبود یافته (n برابر)	میانگین بهبود یافته
۲۷,۰۵۲	۱۲,۸۴۱	۲,۱	۲,۹
۲۶,۰۴۰	۶,۸۱۸	۳,۸۲	
۲۶,۹۰۵	۹,۲۵۶	۲,۹	
۲۶,۱۶۵	۷,۱۲۱	۳,۶۷	
۲۹,۹۶۴	۱۰۱۵۲	۲,۹۵	
۳۰,۴۲۰	۸,۱۲۱	۳,۷۴	
۲۶,۳۳۳	۹,۹۲۰	۲,۶۵	
۲۶,۲۲۶	۸,۰۶۰	۳,۲۵	
۲۶,۳۱۲	۱۱,۸۸۴	۲,۲۱	

در نمونه کد ۱ میتوانید بخش مربوط به موازی‌سازی این برنامه را مشاهده کنید.

```
__m128 vec;
__m128 Max = _mm_setzero_ps();
for (int i = 0; i < VECTOR_SIZE; i += 4) {
    vec = _mm_loadu_ps(&in[i]);
    Max = _mm_max_ps(Max, vec);
}
__m128 sh1 = _mm_shuffle_ps(Max, Max, _MM_SHUFFLE(3, 3, 2, 3));
__m128 maxsh1 = _mm_max_ps(Max, sh1);
__m128 sh2 = _mm_shuffle_ps(maxsh1, maxsh1, _MM_SHUFFLE(3, 3, 3, 3));
__m128 maxsh2 = _mm_max_ps(maxsh1, sh2);
max = _mm_cvtss_f32(maxsh2);
```

نمونه کد (۱)

۳- لبه یابی با Sobel

یکی از متداول ترین عملیات ها در پردازش تصویر تشخیص لبه یا Edge Detection می باشد؛ زیرا لبه مرز میان یک شی و زمینه آن را نشان می دهد. به عبارت دیگر لبه تغییر دو سطح خاکستری یا مقدار مربوط به روشنایی دو پیکسل مجاور است که در مکان خاصی از تصویر رخ می دهد. هر چه این اختلاف بیشتر باشد، تشخیص لبه، تغییرات روشنایی و همچنین لبه های نویزی ساده تر است.

الگو های تشخیص لبه شامل سه مرحله ی فیلترینگ، مشتق گیری و تشخیص می شوند. در مرحله ی فیلترینگ، تصویر از یک فیلتر به منظور حذف نویز عبور داده می شود. در مرحله مشتق گیری، موقعیت هایی در تصویر با تغییرات شدت نویز زیاد برجسته می شود. سرانجام در مرحله تشخیص با آستانه گیری از نقاطی که در مرحله مشتق گیری برجسته شده اند، نقاط به لبه تبدیل می گردند. الگوریتم ها و روش های مختلفی به منظور تشخیص لبه تصویر وجود دارند از جمله Sobel،

Canny، Roberts و غیره. در این تمرین از روش Sobel استفاده می کنیم.

این الگوریتم لبه ها را با استفاده از تخمین زدن مشتق های در راستای X و Y پیدا می کند که لبه ها را در نقاط با بیشترین گرادیان تصویر بر می گرداند. الگوریتم Sobel در دو راستای X و Y از تصویر گرادیان می گیرد. دو ماتریس نشان داده شده در شکل ۲، ماتریس هایی هستند که در تصویر Convolve می شوند تا مشتق های در راستای X و Y را محاسبه شوند.

سپس با استفاده از معادله زیر اندازه تخمینی گرادیان و جهت گرادیان را بدست می آوریم:

$$G = \sqrt[2]{{G_x}^2 + {G_y}^2}$$

$$\theta = \tan^{-1}(\frac{G_y}{G_x})$$

برای نوشتن این برنامه از کتابخانه OpenCV و IPP شرکت اینتل به منظور پیاده سازی موازی استفاده شده است. در ابتدا برنامه را به صورت سریال و با یک حلقه کامل که تمام سطر و ستون های تصویر را پوشش دهد نوشته و زمان اجرای آن را به دست آورده ام. برای نوشتن این بخش از کد از Class از پیش تعیین شده Mat در OpenCV استفاده کرده ایم. Mat در واقع نوعی ماریس است که در کارهای پردازش تصویر بسیار کاربرد دارد.

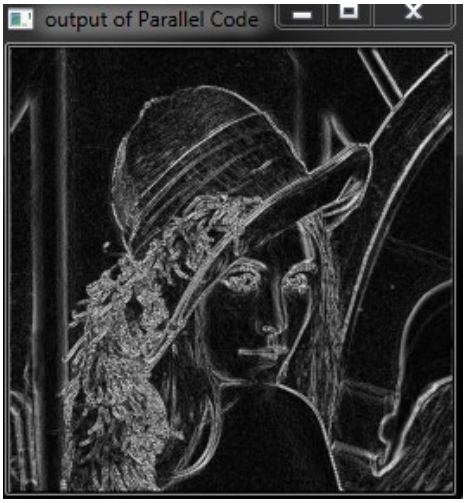
سپس مشابه همان پیاده سازی سریال، پیاده سازی موازی انجام شده دو حلقه تو در تو در برای پیمایش تمام سطر و ستون های تصویر در نظر گرفته شد، اما از دستور های SIMD استفاده شد. در SIMD با انتخاب نوع داده _m128i و _m128می توان طول داده را به ۱۲۸ بیت رساند یعنی ۱۶ بایت و هر بایت ۸ بیت، هر پیکسل تصویر نیز ۱ بایت معادل unsigned char است که باعث می شود ما نوع داده _m128i را انتخاب کنیم. در نتیجه در SIMD می توان در هر لحظه ۱۶ بایت را Process نمود و این کار از حجم محاسبات می کاهد؛ زیرا طول حلقه ها کوتاه تر شده و در نتیجه تعداد تکرار ها کاهش می یابد. بدین ترتیب زمان اجرای برنامه کاهش چشمگیری پیدا می کند. در این برنامه از کد های SSE2، SSE3 و SSE4 استفاده شده است.

برنامه را با ورودی عکس "Lenna.png" چندین بار اجرا کردیم و مشاهده می کنیم که برنامه ای که به صورت موازی اجرا شده است حدود ۴۵ برابر از حالت سریال سریع تر است.

یک بار دیگر برنامه را با عکس "girlface.bmp" چندین بار اجرا کرده و افزایش ۴۰ برابری سرعت اجرا را مشاهده کردیم.

(الف): تصویر ورودی

(ب): خروجی کد سریال



(ج): خروجی کد موازی

```
Execution time of Serial: 16269 s
Execution time of Parallel: 382 s
Speedup = 42
```

(د): نمونه‌ای از نتایج

شکل (۲): ورودی شکل اول به الگوریتم Sobel

در تصاویر ۳ و ۴ تصاویر ورودی، خروجی از طریق کد سریال، خروجی از طریق کد موازی شده و نتایج سرعت اجرای چند نمونه از اجراها را مشاهده می کنیم. نتایج مربوط به سرعت هر الگوریتم را برای دو تصویر در جداول ۲ و ۳ مشاهده نمایید.

جدول (۲): زمان اجرای برنامه Sobel عکس اول

زمان اجرای سریال	زمان اجرای موازی	میزان بهبود یافته (n برابر)	میانگین بهبود یافته
۱۶,۲۶۹	۰,۳۸۲	۴۲	
۳۷,۴۶۵	۰,۶۷۳	۵۵	
۳۲,۹۶۸	۰,۷۰۵	۴۶	۴۵
۲۱,۰۵۰	۰۴۰۲	۵۲	
۱۲,۳۶۵	۰,۴۰۴	۳۰	

جدول (۳): زمان اجرای برنامه Sobel عکس دوم

زمان اجرای سریال	زمان اجرای موازی	میزان بهبود یافته (n برابر)	میانگین بهبود یافته
۴۲,۲۶۸	۰,۸۴۲	۵۰	
۲۸,۱۱۸	۰,۴۸۴	۵۸	
۳۲,۱۹۵	۱,۰۴۷	۳۰	۴۱,۴
۲۶,۷۶۲	۰,۶۵۷	۴۰	
۲۸,۸۱۲	۰,۹۸۲	۲۹	

در نمونه کد ۲ می‌توانید بخش‌های مربوط به موازی سازی برنامه را مشاهده نمایید.

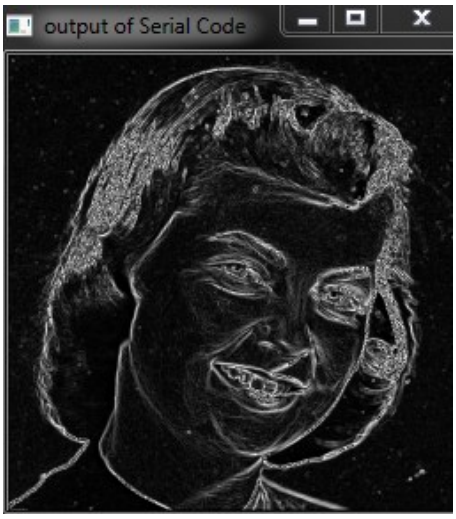
```
__m128i *pSrc;
__m128i x_gradient = _mm_setzero_si128();
__m128i y_gradient = _mm_setzero_si128();
__m128i m1, m2, m3, m4, m5, m6, m7, m8, m9, m10, m11 = _mm_setzero_si128();
start = ippGetCpuClocks();
for (int i = 1; i < image.rows - 1; i++)        // = y
    for (int j = 1; j < image.cols - 1; j += 16) {      // = x
        //Gx
        m1 = _mm_loadu_si128((__m128i *)(in_image + (i - 1) * image.cols + (j + 1)));
        m2 = _mm_loadu_si128((__m128i *)(in_image + (i)     * image.cols + (j + 1)));
        m3 = _mm_loadu_si128((__m128i *)(in_image + (i + 1) * image.cols + (j + 1)));
        m4 = _mm_loadu_si128((__m128i *)(in_image + (i - 1) * image.cols + (j - 1)));
        m5 = _mm_loadu_si128((__m128i *)(in_image + (i)     * image.cols + (j - 1)));
        m6 = _mm_loadu_si128((__m128i *)(in_image + (i + 1) * image.cols + (j - 1)));
        m7 = _mm_add_epi8(_mm_add_epi8(_mm_add_epi8(m2, m2), m1), m3);
        x_gradient = _mm_sub_epi8(_mm_sub_epi8(_mm_sub_epi8(_mm_sub_epi8(m7, m4), m5), m5), m6);
        //Gy
        m1 = _mm_loadu_si128((__m128i *)(in_image + (i + 1) * image.cols + (j - 1)));
        m2 = _mm_loadu_si128((__m128i *)(in_image + (i + 1) * image.cols + (j)));
        m3 = _mm_loadu_si128((__m128i *)(in_image + (i + 1) * image.cols + (j + 1)));
        m4 = _mm_loadu_si128((__m128i *)(in_image + (i - 1) * image.cols + (j - 1)));
        m5 = _mm_loadu_si128((__m128i *)(in_image + (i - 1) * image.cols + (j)));
        m6 = _mm_loadu_si128((__m128i *)(in_image + (i - 1) * image.cols + (j + 1)));
        m7 = _mm_add_epi8(_mm_add_epi8(_mm_add_epi8(m2, m2), m1), m3);
        y_gradient = _mm_sub_epi8(_mm_sub_epi8(_mm_sub_epi8(_mm_sub_epi8(m7, m4), m5), m5), m6);
        m8 = _mm_adds_epu8(_mm_abs_epi8(x_gradient), _mm_abs_epi8(y_gradient));
        _mm_storeu_si128((__m128i *)(out_image_p + i * image.cols + j), m8);
    }
```

نمونه کد (۲): مربوط به قسمت موازی سازی برنامه Sobel

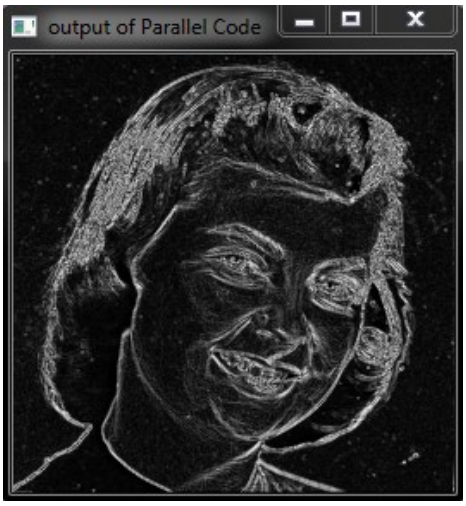
۴ – تشخیص عمق تصویر با Stereo Vision

در این بخش از گذارش بـه بررسـی روش Stereo Vision و استفاده از ابزار های مربوط به این طرح در C++ می پـردازیم. در این قسمت از گذارش سعی بر این است که با در نظر گـرفتن دو

(الف) تصویر اصلی



(ب) خروجی کد سریال



(ج) خروجی کد موازی

```
first input image width: 512
first input image height: 512
Execution time of Serial: 42268 s
Execution time of Parallel: 842 s
Speedup = 50
```

(د) یک نمونه از نتایج

شکل (۳): ورودی شکل دوم به الگوریتم Sobel

عکس که از دو جهت چپ و راست گرفته شده است و الگوریتمی تحت عنوان Kernels که اساس آن سنجش اختلاف بر اساس رابطه زیر می خواهیم عمق اجسام را در تصاویر بدست آوریم.

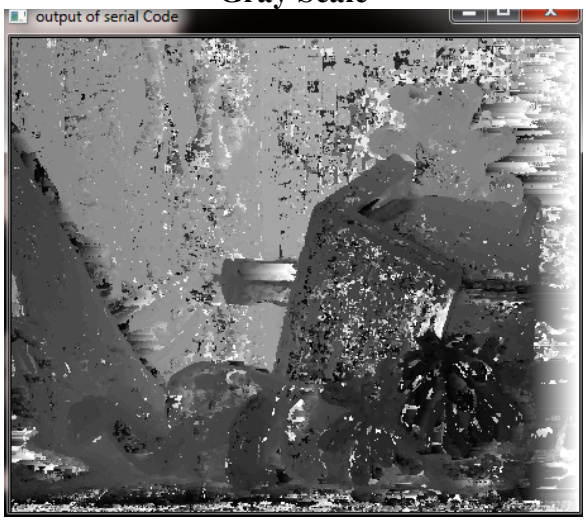
$$S(i,j,k)=\sum_{l=i-\frac{w-1}{2}}^{i+\frac{w-1}{2}}\sum_{m=j-\frac{w-1}{2}}^{j+\frac{w-1}{2}}\left|I_R(l,m)-I_L(l,m+k)\right|$$

در ابتدا یک پنجره به اندازه W که در این مثال مقدار ۳ از ما خواسته شده را در یکی از عکس ها (به طور خاص در این جا ما از عکس سمت راست اسفاده کردیم) در نظر می گیریم و در عکس دیگر همین پنجره را به دفعات BETA که در این مثال خاص مقدار ۴۵ به ما داده شده است به صورت یک پیکسل یک پیکسل حرکت می دهیم. میانگین هر ۹ پیکسل موجود در یک پنجره را گرفته و از تمامی مقادیر BETA کم می کنیم. حال بر اساس مقدار کمترین تا بیشترین پیکسل های موجود در این ۴۵ پنجره را دسته بندی می کنیم و به هر کدام از آن ها عددی بین ۱ تا ۴۵ را اختصاص می دهیم. پس از هر بار عمل مینیمم گیری مقادیر میانگین اختلاف مینیمم را در متغییری ذخیره می کنیم (مثال متغیری مانند min) و برای پنجره بعدی مقدار مینیمم را برابر با این عدد گذاشته و باقی اختلاف ها را با این عدد مقایسه می کنیم و پیکسل ها را آپدیت می کنیم. برای هر بار تکرار این حرکت نیاز داریم که مقدار مینیمم را آپدیت کنیم.

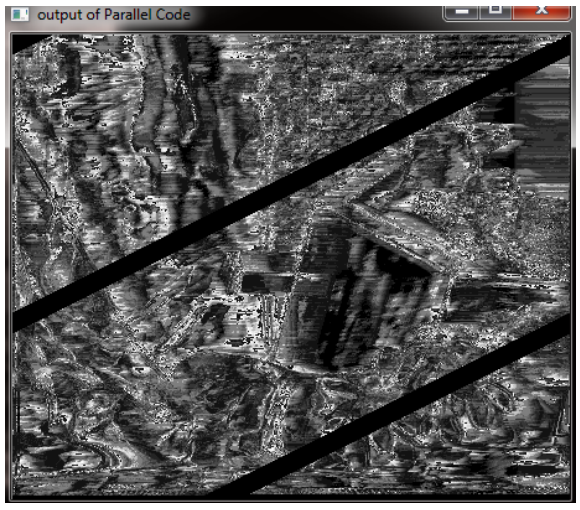
در این تمرین سعی بر آن شده است که ابتدا حالت سریال را بررسی کنیم و پس از بررسی حالت سریال به سراغ حالت برداری برویم. برای حالت سریال از تصویر موجود در سایت http://vision.middlebury.edu/stereo استفاده می کنیم. پس از تبدیل دو عکس راست و چپ این تصویر (که با یکدیگر ۱۰ پیکسل فاصله دارند) به حالت Gray Scale ، عملیات توضیح داده شده در بالا را روی آنها انجام می دهیم. به دلیل سایز پنجره و ۱۰۰٪ نبودن دقت طبیعی است که در عکس ما اختلال هایی ایجاد شود و وضوح عکس کامل نباشد. بر اساس شدت نور سفید در عکس می توان عمق اجسام موجود در عکس را تشخیص داد به این گونه که هرچه عمق جسم در عکس بیشتر باشد جسم به سمت سفیدی بیشتر می رود. حالت ساده و عمق کد سریال و کد موازی عکس در شکل ۴ موجود است.

پس از بررسی حالت سریال حال به سراغ حالت موازی می رویم. برای حالت موازی از دستورات SIMD استفاده شده است به اینگونه که سه بردار ۱۶ تایی از ماتریس عکس خوانده می شود و با ۳ بردار ۱۶ تایی دیگر به صورت همزمان تفریق و سپس مقایسه می شود. در صورت برابر نبود و کمتر بودن شدت عکس

(الف) عکس سمت راست پس از تبدیل به حالت Gray Scale



(ب) عکس پردازش شده برای بدست آوردن عمق در حالت سریال



(ج) عکس پردازش شده برای بدست آوردن عمق در حالت موازی

شکل (۴): نتایج عمق عکس اول

هر کدام از پیکسل های مقایسه شده در بردار ۱۶ تـایی بـه یـک بردار ۱۶ تایی که به صورت پیش فرض مقـدار صـفر را دارا بـوده یک واحد اضافه می شود. پس از ۴۵ مرتبه تکرار این عمـل حـال نتایج به جای یک پیکسل برای ۱۶ پیکسل موجـود اسـت و ایـن ۱۶ پیکسل به صورت موازی در بـردار عکـس ذخیـره مـی شـود. (بخش موازی این کد را در نمونه کد ۳ مشـاهد کنیـد.) طبیعـی است که این روش سرعتی بسیار بالاتری نسبت بـه روش سـریال به ما می دهد ولی میزان کیفیت آن کاهش می‌یابد.

```
__m128i minsum=_mm_set1_epi8(0xff);
__m128i setone= _mm_set1_epi8(1);
__m128i ks = _mm_set1_epi8(0);
__m128i setff = _mm_set1_epi8(220);
__m128i kb = _mm_set1_epi8(0);
for (int i = 1; i < NROWS - 1; i++) {
    for (int j = 0; j < NCOLS - 16; j += 16) {
        minsum = _mm_set1_epi8(0xff);
        ks = _mm_set1_epi8(0);
        if ((NCOLS/16 - j) < BETA)
            re= BETA-NCOLS - j*16;
        else
            re = BETA;
        for (int k = 0; k < re; k++) {
            __m128i mr1 = _mm_loadu_si128((const __m128i*)(pSrcR + (i - 1)*NCOLS + j  -1));
            __m128i mr2 = _mm_loadu_si128((const __m128i*)(pSrcR + (i - 1)*NCOLS + j +1));
            __m128i mr3 = _mm_loadu_si128((const __m128i*)(pSrcR + (i - 1)*NCOLS + j ));
            __m128i mr4 = _mm_loadu_si128((const __m128i*)(pSrcR + (i - 1)*NCOLS + j + NCOLS -1));
            __m128i mr5 = _mm_loadu_si128((const __m128i*)(pSrcR + (i - 1)*NCOLS + j + NCOLS +1));
            __m128i mr6 = _mm_loadu_si128((const __m128i*)(pSrcR + (i - 1)*NCOLS + j + NCOLS ));
            __m128i mr7 = _mm_loadu_si128((const __m128i*)(pSrcR + (i - 1)*NCOLS + j + 2 * (NCOLS - 1)));
            __m128i mr8 = _mm_loadu_si128((const __m128i*)(pSrcR + (i - 1)*NCOLS + j + 2 * (NCOLS +1)));
            __m128i mr9 = _mm_loadu_si128((const __m128i*)(pSrcR + (i - 1)*NCOLS + j + 2 * NCOLS ));
            __m128i ml1 = _mm_loadu_si128((const __m128i*)(pSrcL + (i - 1)*NCOLS +k+ j - 1));
            __m128i ml2 = _mm_loadu_si128((const __m128i*)(pSrcL + (i - 1)*NCOLS +k+ j + 1));
            __m128i ml3 = _mm_loadu_si128((const __m128i*)(pSrcL + (i - 1)*NCOLS +k+ j));
            __m128i ml4 = _mm_loadu_si128((const __m128i*)(pSrcL + (i - 1)*NCOLS +k+ j + NCOLS - 1));
            __m128i ml5 = _mm_loadu_si128((const __m128i*)(pSrcL + (i - 1)*NCOLS +k+ j + NCOLS + 1));
            __m128i ml6 = _mm_loadu_si128((const __m128i*)(pSrcL + (i - 1)*NCOLS +k+ j + NCOLS));
            __m128i ml7 = _mm_loadu_si128((const __m128i*)(pSrcL + (i - 1)*NCOLS +k+ j + 2 * (NCOLS - 1)));
            __m128i ml8 = _mm_loadu_si128((const __m128i*)(pSrcL + (i - 1)*NCOLS +k+ j + 2 * (NCOLS + 1)));
            __m128i ml9 = _mm_loadu_si128((const __m128i*)(pSrcL + (i - 1)*NCOLS +k+ j + 2 * NCOLS));
            __m128i sum = _mm_adds_epi8(_mm_add_epi8(_mm_add_epi8(_mm_add_epi8(_mm_sub_epi8(mr1, ml1), _mm_sub_epi8(mr2, ml2)), _mm_add_epi8(_mm_sub_epi8(mr3, ml3), _mm_sub_epi8(mr4, ml4))),
                _mm_add_epi8(_mm_add_epi8(_mm_sub_epi8(mr5, ml5), _mm_sub_epi8(mr6, ml6)), _mm_add_epi8(_mm_sub_epi8(mr7, ml7), _mm_sub_epi8(mr8, ml8)))), _mm_sub_epi8(mr9, ml9));
            __m128i cmp = _mm_cmplt_epi8(minsum, sum);
            kb = _mm_add_epi8(cmp, setone);
            ks = _mm_add_epi8(kb, ks);
            minsum = _mm_min_epi8(sum, minsum);
        }
        ks = _mm_adds_epi8(_mm_add_epi8(_mm_add_epi8(_mm_add_epi8(_mm_add_epi8(_mm_add_epi8(ks, _mm_add_epi8(_mm_add_epi8(ks, ks), _mm_add_epi8(ks, ks))),
            _mm_add_epi8(ks, ks)), _mm_add_epi8(ks, _mm_add_epi8(ks, ks))), _mm_add_epi8(ks, ks)), _mm_adds_epi8(ks,ks)), _mm_adds_epi8(_mm_adds_epi8(
        _mm_store_si128((__m128i*)(pRes + (i - 1)*NCOLS + j), ks);
    }
}
```

نمونه کد (۳): برای بخش موازی Stereo Vision

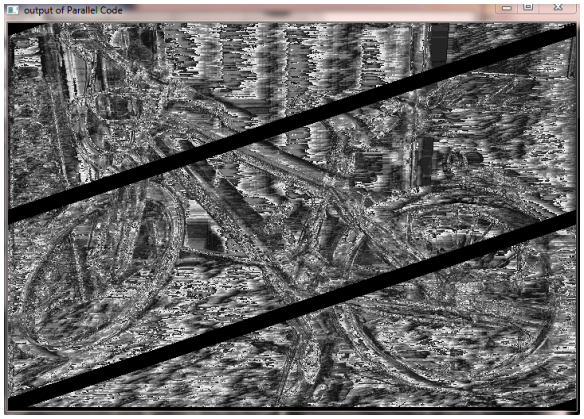
پس از بررسی پردازش های سریال و موازی حال نوبت آن اسـت که تفاوت سرعت و کیفت این حالات را با یکدیگر بسنجیم. بـرای این کاز از چند عکس مختلف استفاده شده که در شکل های ۵ و ۶ مشاهده می کنیم ، الهام گرفتیم و نتایج کلی در جدول‌هـای ۴ و ۵ و ۶ مشاهده می شود.

زمان اجرای سریال	زمان اجرای موازی	میزان بهبود یافته (n برابر)	میانگین بهبود یافته
۴۷۰,۶۵۹	۸,۱۵۳	۵۰	۳۶,۴
۳۹۲,۴۰۵	۲۷,۴۴۲	۱۴	
۳۷۱,۲۳۷	۷,۶۶۲	۴۸	
۳۹۵,۸۳۶	۲۵,۸۴۴	۱۵	
۳۷۷,۱۰۰	۴,۸۰۰	۵۵	

جدول (۴): زمان اجرای برنامه Stereo Vision عکس اول

(الف) تصویر اصلی

(ب) خروجی کد سریال



(ج) خروجی کد موازی

```
Execution time of Serial: 763318
Execution time of Serial: 28131
Speedup = 27
```

(د) یکی از نتیجه‌های حاصله

شکل (۵): نتایج عمق تصویر دوم

جدول (۵): زمان اجرای برنامه Stereo Vision عکس دوم

زمان اجرای سریال	زمان اجرای موازی	میزان بهبود یافته (n برابر)	میانگین بهبود یافته
۷۶۳,۳۱۸	۲۸,۱۳۱	۲۷	۳۸
۷۵۷,۱۶۸	۳۴,۲۶۶	۲۲	
۷۵۹,۰۲۲	۳۳,۱۷۹	۲۲	
۷۸۶,۸۹۵	۱۲,۹۳۷	۶۰	
۷۷۰,۰۰۴	۱۲,۹۰۰	۵۹	

جدول (۶): زمان اجرای برنامه Stereo Vision عکس سوم

زمان اجرای سریال	زمان اجرای موازی	میزان بهبود یافته (n برابر)	میانگین بهبود یافته
۸۳۰,۲۷۶	۱۴,۷۸۵	۵۶	۴۴,۲
۸۰۴,۷۸۶	۳۸,۷۳۶	۲۰	
۷۸۲,۲۰۰	۱۷,۷۳۹	۴۴	
۷۷۸,۴۱۲	۱۳,۰۴۱	۵۹	
۸۰۰,۴۱۵	۱۸,۸۳۷	۴۲	

با کنار هم گذاشتن این نتیجش مشاهده می کنیم که به صورت میانگین برای تمامی حالت ها ، حالت موازی حدود ۳۸ برابر سریع تر از حالت سریال کار می کند.

۵- نتیجه

پس از بررسی حالت های مختلف برنامه نویسی موازی مشاهده می شود که استفاده از دستورات SIMD می تواند سرعت پردازش ما را بسیار افزایش دهد اما باید توجه داشت که این افزایش سرعت که در برخی از مثال ها تا حدود ۵۰ برابر نیز می باشد در صورتی امکان پذیر است که کیفیت کار پایین آید. پس برای کاربرد های خاص که نیاز به دقت بالا نیست و سرعت اولیوت بالایی دارد استفاده از روش های موازی SIMD با وجود داشتن خطا ، مناسب می باشد.

مراجع

[1] وبسایت http://vision.middlebury.edu

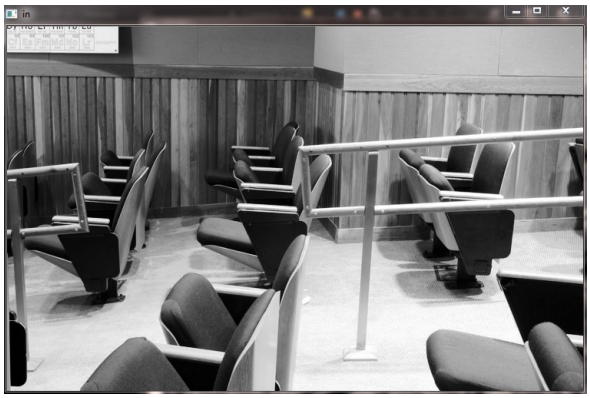
[2] وبسایت شرکت INTEL www.intel.com

[3] لکچر های درس محاسبات با کارایی بالا

[4] وبسایت www.wikipedia.com

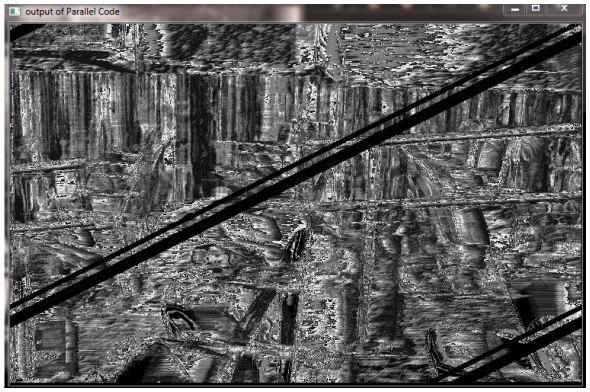
ضمایم

کدهای مربوط به هر بخش در پوشه خود موجود می‌باشند

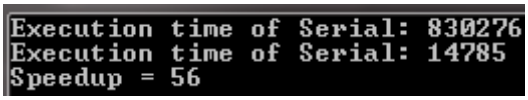


(الف) تصویر اصلی

(ب) خروجی کد سریال



(ج) خروجی کد موازی



(د) یکی از نتیجه‌های حاصله

شکل (۶): نتایج عمق تصویر سوم