به نام خدا

تشخیص‌دهنده لبه

پردازش چندهسته‌ای - پروژه پایانی- دکتر هاجر فلاحتی

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

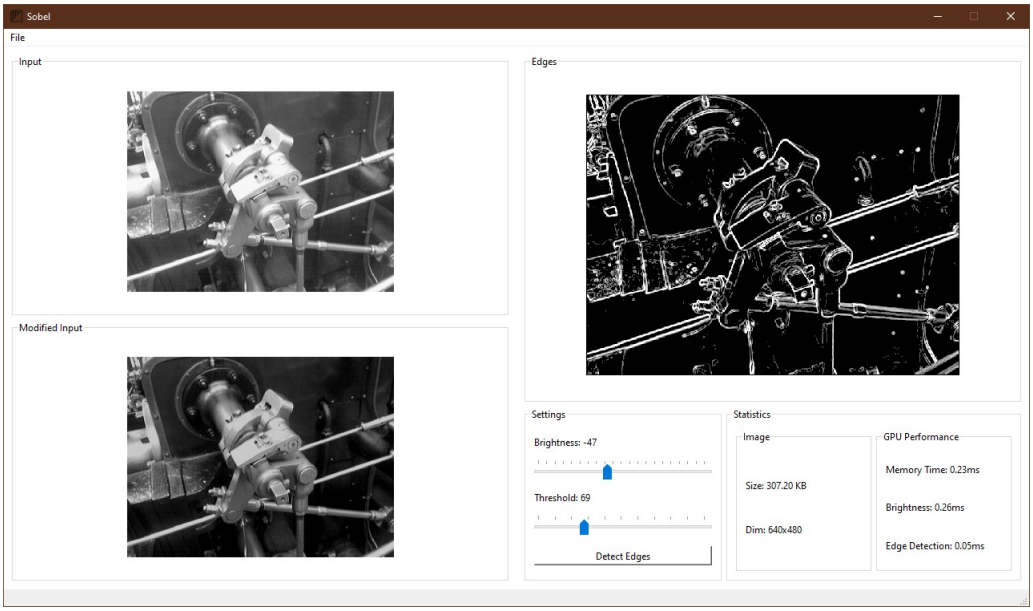
تابستان ۱۴۰۱

نویسندگان:

فاطمه خاشعی-۹۷۱۰۵۸۹۹

علی حاتمی تاجیک-۹۸۱۰۱۳۸۵

فهرست مطالب



شکل ۱: شمایل نهایی برنامه

۱ مقدمه

در این پروژه سعی شده است که یک تشخیص دهنده‌لبه [1] با استفاده از Spbel Operator پیاده‌سازی شود که برروی پردازنده‌های گرافیکی شرکت Nvidia (پلتفرم CUDA) اجرا شود. برای بهبود زمان اجرای برنامه از تکنیک‌هایی برای بالابردن بهره‌وری استفاده شده است و همچنین برای رابط کاربری بهتر از واسط کاربری گرافیکی استفاده شده است. شمایل نهایی در شکل ۱ آمده است.

۲ پیاده‌سازی

پیاده‌سازی انجام شده با استفاده از الگوریتمی که در WikiPedia آمده بود پیاده‌سازی ساده این الگوریتم را از کد متلبی که داده شده بود الهام گرفتیم و مطابق با آن الگورتیم را در C++ پیاده‌سازی کردیم.

۱.۲ پیاده‌سازی ساده

ساده‌ترین پیاده‌سازی که به ذهن می‌رسد، پیاده‌سازی کانلوشون و انجام آن روی عکس و در نهایت گرفتن جذر جمع مجذور دو عدد و اعمال آستانه [2] نهایی است. البته برای راحتی در پیاده‌سازی فیلترها به صورتی چیده شده اند که ضرب عضو به عضو آنها کفایت کند (چرخاندن فیلتر پیش از اجرای کرنل انجام گرفته است). در این پیاده‌سازی ساده هر ترد مسئول یک پیکسل از خروجی نهایی است و ۹ ضرب (سایز فیلتر) را انجام داده، نتایج را با یکدیگر جمع می‌کند و نتیجه کانولوشن را به ما می‌دهد. این ضرب‌ها با استفاد ها از دو حلقه حول مرکز فیلتر انجام می‌گیرد. کد کرنل پیاده‌سازی شده در ادامه آمده است.

```
__global__ void sobelCUDA(const uint8_t* image,
    const int8_t* xKernel,
    const int8_t* yKernel,
    uint8_t* output,
    int width,
    int height,
    int kernelDim,
```

[1] Edge Detector
[2] Threshold

```
    int threshold)
{
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int idx = i * width + j;

    int center = (kernelDim - 1) / 2;
    float S1 = 0, S2 = 0;
    int jshift, ishift;
    int out;

    if (i >= center && j >= center &&
        i < height - center && j < width - center)
    {
        for (int ii = 0; ii < kernelDim; ii++) {
            for (int jj = 0; jj < kernelDim; jj++) {
                jshift = jj + j - center;
                ishift = ii + i - center;
                S1 += image[ishift * width + jshift] * xKernel[ii * kernelDim + jj];
                S2 += image[ishift * width + jshift] * yKernel[ii * kernelDim + jj];
            }
        }

        out = sqrtf(S1 * S1 + S2 * S2);
        out = out > 255 ? 255 : out;
        output[idx] = out > threshold ? out : 0;
    }
}
```

۲.۲ پیاده‌سازی بهینه‌سازی شده

مشکلی که در کد بالا وجود دارد این است که دسترسی به حافظه‌ای بالا وجود دارد و همینطور با علم به اینکه قرار است چند ضرب و جمع وجود داشته باشد اما با این حال از حلقه‌های تو در تو استفاده شده است که باعث می‌شود هم دسترسی به حافظه بیهوده داشته باشیم و هم دستورات مربوط به حلقه بار اضافی برای اجرا باشند. راه حل این است که ما فیلترها را از پیش داریم و میدانیم که از هر کدام سه ضرب و جمع بیهوده استفاده می‌شوند (چرا که عامل صفر را داریم ضرب می‌کنیم!). برای همین از این روش استفاده می‌کنیم که فیلترها را به صورت کد-سخت شده [۳] درون کد کرنل می‌آوریم. با این کار هم دسترسی به حافظه که مربوط به فیلترها بوده است را کم کرده‌ایم، هم حلقه‌ها را از بین برده‌ایم، هم از صفر بودن عامل‌های فیلتر برای کم کردن تعداد دستورات و هم از معلوم بودن فیلتر برای کم کردن تعداد ضرب‌ها (تبدیل ضرب در ۱ و ۱- به جمع و تفریق) استفاده کرده‌ایم. همچنین از الگوی دسترسی منظمی نیز برای کم‌شدن miss در حافظه کش استفاده شده است. در کرنل بهینه‌سازی‌شده از قطعه کد زیر به جای حلقه‌های تو در تو استفاده شده است:

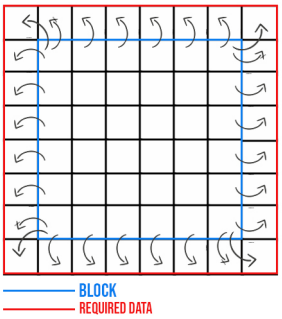
```
S1 = image[(i - 1) * width + (j + 1)] - image[(i - 1) * width + (j - 1)] +
  (image[(i + 1) * width + (j + 1)]) - image[(i + 1) * width + (j - 1)] +
    2 * ( image[i * width + (j + 1)] - image[i * width + (j - 1)]);

S2 = image[(i - 1) * width + (j - 1)] + (image[(i - 1) * width + (j + 1)]) +
     2 * (image[(i - 1) * width + j] - image[(i + 1) * width + j])
     - image[(i + 1) * width + (j - 1)] - image[(i + 1) * width + (j + 1)];
```

۳.۲ افزودن Shared Memory

در این قسمت برای کم کردن دسترسی به حافظه از Shared Memory استفاده می‌کنیم که پیش از عملیات اصلی آن قسمتی از عکس که در اجرای کد اصلی استفاده می‌شود را در حافظه اشتراکی بریزیم. در قسمت ۳ به این مسئله می‌پردازیم که این کار نه تنها باعث افزایش سرعت نمی‌شود بلکه از سرعت می‌کاهد. در هر بلوک ۱۰۲۴ ریسه در چیدمان ۳۲ در ۳۲ کار می‌کنند و هر کدام مسئول یک پیکسل از خروجی هستند که هرکدام به پیکسل‌های اطراف خودشان نیاز دارند که با این حساب در هر بلوک از داده‌های یک قسمت ۳۴ در ۳۴ از عکس اصلی استفاده می‌شود.

[3] Hard-Coded



شکل ۲: خواندن داده‌های لبه

برای این کار ابتدا هر ترد پیکسل منطبق بر پیکسل خروجی را از ورودی به حافظه اشتراکی می‌آورد. سپس ریسه‌هایی که در لبه‌های قسمت ۳۲ در ۳۲ عکس اصلی قرار دارند پیکسل‌های مجاور خود که در خارج از محدوده ۳۲ در ۳۲ قرار دارند می‌آورند. شکل ۲ نشان می‌دهد که مسئول خواندن هر پیکسل که در حافظه اشتراکی قرار می‌گیرد با کدام ترد متناظر است (تنها برای پیکسل‌های کناری در شکل نشان‌داده شده‌اند).
حافظه اشتراکی به وسیله قطعه کد زیر که پیش از کد اصلی قرار می‌گیرد پر می‌شود:

```
__shared__ uint8_t sdata[34 * 34];
if (i < height && j < width) {
    // Main data
    sdata[(y + 1) * 34 + x + 1] = image[idx];

    // Fix Boudnries
    if (y == 0 && blockIdx.y != 0) {
        sdata[x + 1] = image[(i - 1) * width + j];
        if (x == 0 && blockIdx.x != 0) {
            sdata[0] = image[(i - 1) * width + (j - 1)];
        }
    }
    if (x == 0 && blockIdx.x != 0) {
        sdata[y34 + 34] = image[(i)*width + j - 1];
        if (y == 31 && i != height - 1) {
            sdata[33 * 34] = image[(i + 1) * width + j - 1];
        }
    }
    if (y == 31 && i != height - 1) {
        sdata[33 * 34 + x + 1] = image[((i + 1) * width + j)];
        if (x == 31 && j != width - 1) {
            sdata[33 * 34 + 33] = image[(i + 1) * width + j + 1];
        }
    }
    if (x == 31 && j != width - 1) {
        sdata[y34 + 34 + 33] = image[i * width + j + 1];
        if (y == 0 && blockIdx.y != 0) {
            sdata[33] = image[(i - 1) * width + j + 1];
        }
    }
}

// waits untill shared data is completed
__syncthreads();
```

حال قطعه کد اصلی به جای اینکه از حافظه داده خود را بردارد از حافظه اشتراکی این کار را انجام می‌دهد.

۴.۲ تغییر روشنایی

برای تغییر روشنایی از یک کرنل ساده استفاده می‌شود که کد آن در ادامه آمده است. پس از افزودن مقدار مورد نظر به هر پیکسل، اگر از باند مربوط به یک پیکسل گذشته بود آنرا به محدوده برمی‌گردانیم.

```
__global__ void changeBrightnessCUDA(uint8_t* input, const int width,
    const int height, const int brightness)
{
    int val;

    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int idx = i * width + j;

    if (i < height && j < width) {
        val = input[idx] + brightness;
        // Truncate the result (0..255)
        if (val > 255) {
            val = 255;
        }
        else if (val < 0) {
            val = 0;
        }
        input[idx] = val;
    }
}
```

۵.۲ پیکربندی اجرای کرنل

اجرای کرنل‌ها روند کلی زیر را طی می‌کند:

۱. حافظه عکس اصلی تخصیص داده می‌شود.

۲. عکس در حافظه کپی می‌شود.

۳. کرنل تغییر روشنایی اجرا می‌شود.

۴. صبر می‌شود تا تغییرات صورت بگیرد.

۵. نتیجه به صورت آسنکرون روی استریمی جدا به هاست منتقل می‌شود.

۶. حین جابجایی عکسی که روشنایی آن تغییر کرده است کرنل تشخیص لبه اجرا می‌شود.

۷. نتیجه نهایی به هاست منتقل می‌شود.

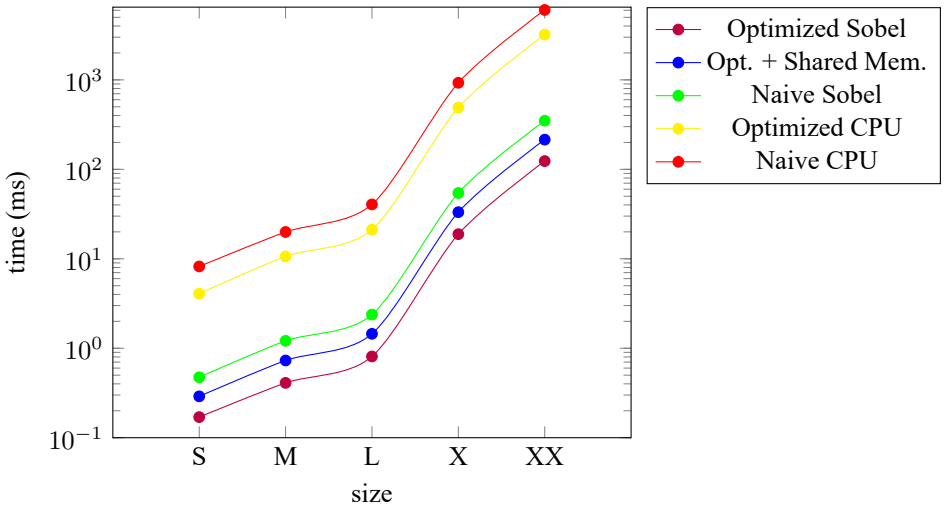
در ادامه کد اجرایی آمده است. البته برای خوانایی گزارش به صورت خلاصه آمده است.

```
  dim3 block(BLOCK_DIM, BLOCK_DIM);
  dim3 grid(width/BLOCK_DIM + (width%BLOCK_DIM!=0),
    height/BLOCK_DIM + (width % BLOCK_DIM != 0));

  cudaMalloc((void**)&dev_input, imageSize);
  cudaMalloc((void**)&dev_edge, imageSize);
  cudaMemcpy(dev_input, input, imageSize, cudaMemcpyHostToDevice);

  changeBrightnessCUDA <<<grid, block, 0, main>>> (dev_input, width, height, brightness);
  cudaGetLastError();
  cudaDeviceSynchronize();

  cudaMemcpyAsync(bright, dev_input, imageSize, cudaMemcpyDeviceToHost, mem);
```



شکل ۳: نتایج بنچمارک انجام شده روی متدهای مختلف

```
14
15  sobelOptimizedShCUDA <<<grid, block, 0, main>>> (dev_input, dev_edge, width, height,
      threshold);
16  cudaEventRecord(stop);
17  cudaGetLastError();
18  cudaDeviceSynchronize();
```

در اینجا با فرستادن غیرهمگام تصویری که روشنایی آن تغییر کرده است، از اورلپ صورت گرفته را در زمان نهایی ذخیره می‌کنیم و Thoughput حافظه را افزایش می‌دهیم.
دلیل نحوه انتخاب این سایز برای بلوک در بخش ۳ توضیح داده شده است.

۳ بررسی و مقایسه عملکرد

در ابتدا به مقایسه علملکرد کرنل‌های متفاوت می‌پردازیم. نتیجه Benchmark اجرا شده روی کدهای مختلف در نمودار شکل ۳ آمده است. (دقت شود که اندازه محور زمان لگاریتمی است). در این شکل به وضوح زمان بهتر اجرا بر روی GPU آمده است. برای این تست از سخت‌افزار زیر استفاده شده است:

CPU 11th Gen Intel ®Core™i7-1165G7 @ 2.80GHz

GPU NVIDIA GeForce MX450

سایز تصاویر به شرح زیر است:

S 729 x 480

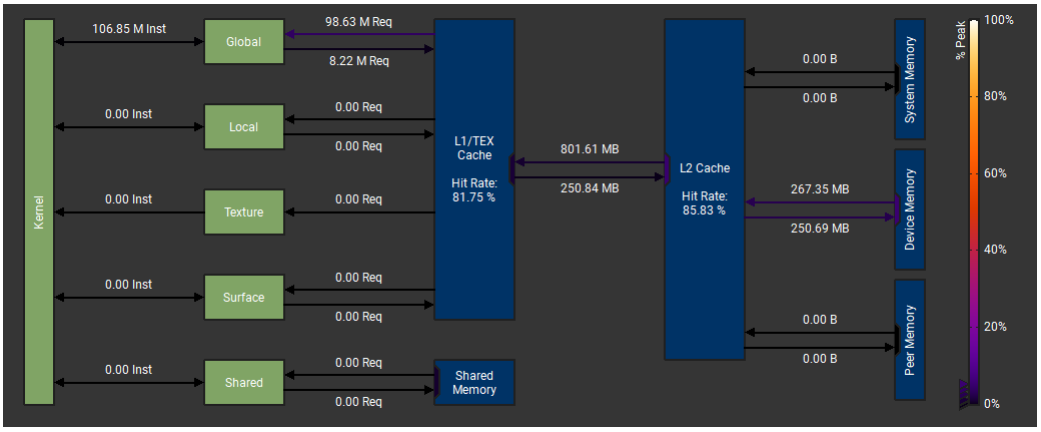
M 1166 x 768

L 1639 x 1080

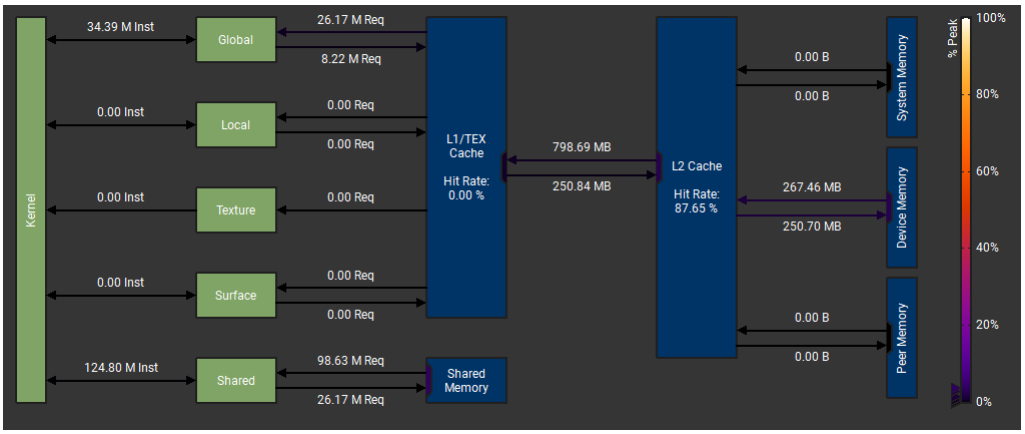
X 7881 x 5192

XX 20000 x 13176

حال به برخی توضیحات می‌پردازیم.



شکل ۴: آنالیز مموری حالت بدون حافظه اشتراکی



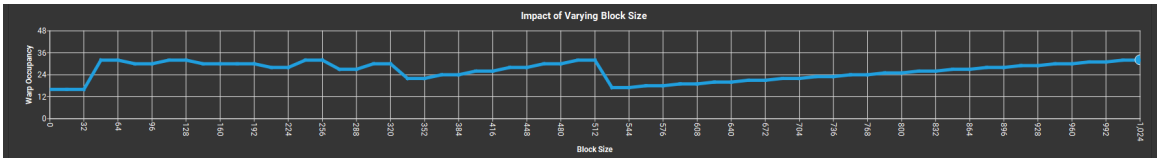
شکل ۵: آنالیز مموری حالت حافظه اشتراکی

۱.۳ دلیل بیشتر بودن زمان هنگام استفاده از حافظه اشتراکی

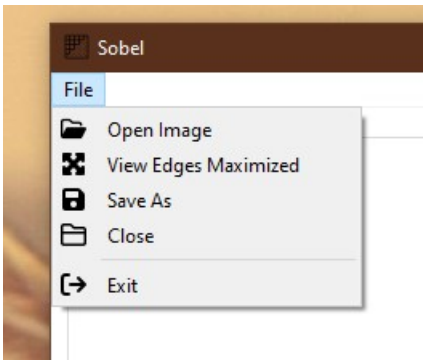
پس از اینکه متوجه این مورد شدیم که زمان اجرای هنگامی که از حافظه اشتراکی استفاده می‌شود بر خلاف انتظارات بیشتر از حالتی است که تماما از حافظه اصلی می‌خوانیم با استفاده از نرم‌افزار Nsight Conpute به پروفایلینگ تست انجام شده پرداختیم. پس از بررسی‌های انجام شده به این نتیجه رسیدیم که دلیل بیشتر شدن زمان این موضوع است که در کد divergantهایی ایجاد شده است که کارایی را پایین آورده است. همچنین در حالتی که از حافظه اصلی استفاده می‌شود Hit Rate بالایی در کش مرحله اول داریم که باعث می‌شود بدون حافظه اشتراکی دسترسی به حافظه همانند حالت اشتراکی باشد. در شکل ۵ و ۴ آنالیزهای مربوط به حافظه آمده است. همچنین آنالیز کلی صورت گرفته در محل `-rep./resources/profile.ncu` موجود است.

۲.۳ دلیل انتخاب سایز بلوک

با توجه به آنالیزی که روی کد انجام گرفته است، اندازه ۱۰۲۴ برای بلوک مناسب‌ترین سایز است و چون سر و کارمان با عکسهاست، اندازه ۱۰۲۴ در ۱ می‌تواند پرتی زیادی در ابعاد ایجاد کند برای همین از بلوک‌های مربعی ۳۲ در ۳۲ استفاده کرده‌ایم که کمترین بلوک ممکن در مساحت استفاده شود. در شکل ۶ آنالیز تعداد بلوک آمده است.



شکل ۶: آنالیز تعداد بلوک حافظه



شکل ۷: منو فایل

۴ رابط کاربری گرافیکی

همانطور که از شکل ۱ پیداست، رابط گرافیکی شامل پنج بخشی کلی است:

- نمایش ورودی
- نمایش ورودی تغییر یافته (روشنایی)
- نمایش خروجی
- انجام تنظیمات قبل از اجرای الگوریتم
- آمار مربوط به اجرا

تنظیمات قبل از اجرا شامل تنظیمات روشنایی و آستانه می‌شود. در منوی فایل همچنین باز کردن، ذخیره‌کردن، نمایش بزرگتر تصویر خروجی (زمانی که تصاویر بزرگ باشند، خروجی که نمایش داده می‌شود از دقت بالایی برخوردار نیست) و بستن برنامه است (شکل ۷). هنگام بستن و یا تعویض ورودی اگر تغییرات ذخیره نشده‌ای وجود داشته باشد از کاربر تاییدی برای ذخیره کردن یا نکردن گرفته می‌شود. برای پیاده‌سازی این رابط کاربری از فریم‌ورک Qt استفاده شده است.

۵ جمع‌بندی

در این پروژه یک تشخیص‌دهنده لبه برای CUDA پیاده‌سازی شده است که دارای ویژگی‌های زیر است:

- استفاده از مکانیزم‌هایی برای بهینه‌سازی الگوریتم:
  - حذف دسترسی حافظه برای فیلتر
  - حذف کردن حلقه [۴]

[4] Loop Unrolling

- در نظر نگرفتن عامل‌های صفر
- استفاده از الگوی دسترسی به حافظه منظم
- راهکاری برای استفاده از حافظه اشتراکی با تداخل بانک صفر (به قصد افزایش سرعت)
- استفاده از اندازه بلوک بهینه
- استفاده از کپی غیرهمگام برای همپوشانی کار ممووری و محاسبات

- پشتیبانی از اندازه‌های بزرگ (8k)
- پیاده‌سازی آستانه
- پیاده‌سازی GUI