

# گزارش پژوهی رایانش چند هسته‌ای

## Edge Detection Filter

نوید اسلامی - ۹۸۱۰۰۳۲۳  
سروش جهانزاد - ۹۸۱۰۰۳۸۹

۱۴۰۱ تیر ۲۶

موضوع پروژه‌ای که در نظر گرفته‌ایم، Edge Detection Sobel است. درواقع با استفاده از دو Convolution ای که مشتق‌های در راستای  $x$  و  $y$  را محاسبه می‌کند، یک تخمین از نرم مشتق کلی به دست می‌آوردم و بر حسب آن یک تصویر خروجی ایجاد می‌کنیم که نقاط سفید به معنی وجود گوشه و نرم بزرگ‌تر مشتق است و نقاط سیاه به معنی یکنواختی عکس است. حال به توصیف روند و تکنیک‌های مورد استفاده در پیاده‌سازی خود می‌پردازیم.

## ۱ پیاده‌سازی و بهینه‌سازی‌های مورد استفاده

در درجه‌ی اول، تصمیم گرفتیم که عمل تغییر Brightness را به صورت یک کرنل جدا از خود عمل فیلتر کردن تعریف کنیم. بعد از تغییر Brightness اما، تصویر ورودی را که فرض کردیم می‌تواند رنگی نیز باشد، به سیاه و سفید تبدیل می‌کنیم. (با میانگین‌گیری مقادیر کانال‌ها) همچنین تصمیم گرفتیم که عمل Thresholding را با استفاده دو مقدار  $t_1$  و  $t_2$  انجام دهیم، به این صورت که پیکسل‌های خروجی‌ای که مقداری کمتر از  $t_1$  داشتند را صفر کنیم و پیکسل‌های با مقدار بیشتر از  $t_2$  را کاملاً روشن کنیم. همچنین، برنامه‌ی مذکور را طوری پیاده‌سازی کردیم که بتواند هر اندازه تصویری را به درستی هندل کند، حتی تصاویر خیلی بزرگ مثل تصاویر 8K. در پیاده‌سازی این پروژه، ابتدا کد اصلی و رابط کاربری CLI را پیاده‌سازی کردیم و سپس به آن GUI اضافه کردیم.

### ۱.۱ پیاده‌سازی CLI

ابتدا به کمک کتابخانه‌ی OpenCV، تصویر مربوطه را از فایل می‌خوانیم. سپس، مقادیر ثابت الگوریتم را مثل تغییر روشنایی تصویر و نیز Threshold‌ها را تعریف می‌کنیم. در ادامه، داده‌هایی که نیاز داریم را در حافظه‌ی GPU در اختیار می‌گیریم. تصویر خوانده شده را با یک سطر صفر از بالا و *channel count* سطر صفر از پایین Pad می‌کنیم تا بتوانیم الگوریتمی که در GPU اجرا می‌کنیم را راحت‌تر پیاده‌سازی کنیم. توجه کنید اما که قبل از این صفر کردن‌ها، صرفاً تصویر اصلی ورودی را در مکان مناسب و مذکور کپی می‌کنیم و آن را با استفاده از کرنل مخصوص change\_brightness مطابق با ورودی ثابت آن روشن‌تر یا تاریک‌تر می‌کنیم.

این کرنل، صرفاً تک تک پیکسل‌ها را که به صورت یک سطر بزرگ در آرایه‌ی خوانده شده ذخیره شده‌اند را تغییر می‌دهد. با استفاده از بلوک‌ها و گریدهای مختلف، ریسه‌ها را مسئول تغییر اجزای مختلف تصویر مذکور می‌کنیم. هر ریسه نیز برای عملکرد بهتر، از تکنیک Loop Unrolling استفاده می‌کند و درواقع ۴ مقدار کanal متوالی را هندل می‌کند. همچنین، خود این هندل کردن نیز را طوری نوشتۀ‌ایم که به صورت Branchless اجرا شود و در نتیجه، بتوانیم سرعت و بهره‌وری بالاتری را در اجرای برنامه‌های خود داشته باشیم. با تکرار همین فرآیند و هندل کردن شیفت‌ها مختلف از مقادیر کanal‌ها، تغییر میزان روشنایی هر کanal را به خوبی انجام می‌دهیم و از پرس اضافه‌ای استفاده نمی‌کنیم. توجه کنید که از آن حایی که این مقادیر صرفاً بایت هستند، هندل کردن چهار مقدار معادل هندل کردن یک کلمه است. در نتیجه، Loop Unrolling به این شکل مشکلی برای Cache Locality ایجاد نمی‌کند و صرفاً تعداد دستورات اضافه‌ای که باید اجرا شود را کمینه می‌کند. با اتمام این فرآیند پس، تصویر خود را روشن‌تر و یا تاریک‌تر می‌کنیم و در همان آرایه‌ی ورودی ذخیره می‌کنیم تا بتوانیم بعداً از آن به راحتی استفاده کنیم.

بعد از تغییر روشنایی تصویر، پردازنده‌ی اصلی را با GPU همگام می‌کنیم تا بتوانیم به صورت امن اعمال بعدی را انجام دهیم. در این مرحله، سطرهای صفری که برای Padding نیاز داشتیم را به کمک cudaMemcpy صفر می‌کنیم و آرایه‌ای که باید تصویر خروجی را در خودش ذخیره کند را در حافظه‌ی

GPU می‌گیریم. در این لحظه نیز، اندازه‌ی بلوک‌ها و گرید را تنظیم می‌کنیم تا بتوانیم فیلتر را اعمال کنیم. در ادامه، با استفاده از این مقادیر، کرنل tiled\_sobel را فراخوانی می‌کنیم تا فیلتر را اعمال کند.

کرنل مذکور، برای این که بتواند خروجی را به خوبی تولید کند، از ایده‌ی Tiling استفاده می‌کند. به عبارتی، هر بلوک از ریسه‌های یک بخش به طول ۱۲۸ و عرض ۸ پیکسلی از ورودی را هندل می‌کند و به آن فیلتر را اعمال می‌کند. همچنین، کد را طوری زده‌ایم که بلوک‌های داخل یک گرید بتوانند با شیفت دادن خود به هندل کردن بخش‌های مشابه بعدی از تصویر پردازند. بدون این که Cache Locality را خیلی هم بریزند. برای این که بتوانیم از Data Sharing حاصل از این ایده استفاده کنیم، از حافظه‌ی مشترک GPU نیز استفاده می‌کنیم. به این شکل عمل می‌کنیم که پیکسل‌های مربوط به هر بخش از تصویر که برای یک بلوک است، و نیز پیکسل‌های مجاور آن بخش را ابتدا به سیاه و سفید تبدیل می‌کنیم (سه بایت ورودی به یک بایت سیاه سفید در smem تبدیل می‌شود) و مقادیر تبدیل شده حاصل را در یک ساختار کاملاً مشابه در حافظه‌ی مشترک smem ذخیره می‌کنیم.

در همین راستا، ابتدا قسمت میانی بخش مربوط به یک بلوک را هندل می‌کنیم و به صورت Branchless بررسی می‌کنیم که آیا پیکسلی که باید لود شود واقعاً داخل تصویر هست یا نه. در صورتی که تبود، چون بلوک ما حافظه‌ی مشترک برای خود دارد، می‌تواند به راحتی و بدون نگرانی صرفاً در آن‌ها صفر ذخیره کند. همچنین توجه کنید که حافظه‌ی مشترک ما Row Major اختصاص داده شده است، اما اندیس هر ریسه اول  $x$ -هایش عوض می‌شود و سپس  $y$ -هایش. در نتیجه، برای این که Bank Conflict نداشته باشیم، از smem طوری استفاده می‌کنیم که  $u$  مربوط به سطر باشد و  $x$  مربوط به شماره‌ی هر عنصر در هر سطر باشد. در نتیجه، به این شکل دسترسی‌های به حافظه در این حالت متواالی خواهد شد و مشکلی سر Bank Conflict نخواهیم داشت. در ادامه، باز سعی کرده‌ایم که بدون پرش عناصر مرزی بخش مربوط به بلوک را در smem بیاوریم. اما در این حالت نیاز بود که یک شرطی داشته باشیم، که سعی شده است صرفاً از آن یک شرط استفاده شود. در نتیجه، با استفاده از روابط ریاضی‌ای که در کد مشهود است، ابتدا دو ستون چپ و راست و بخش میانی سطرهای بالا و پایین (به جز دو پیکسل اول و آخر آن‌ها) را می‌آوریم و در smem ذخیره می‌کنیم. در آخر نیز به روشهای کاملاً مشابه، ۴ پیکسلی که جا مانده بود را لود می‌کنیم. این کار را به این دلیل انجام داده‌ایم که پیاده‌سازی ساده‌تری داشته باشد.

توجه کنید که وجود Padding‌هایی که قبل‌تر ایجاد کردیم در این مرحله خیلی مهم می‌شود، چون دیگر لازم نیست شرطی را برای وجود پیکسل‌های بالا و پایین چک بکنیم و فقط کافی است که شرط بیرون زدن از چپ و راست را بررسی کنیم. این موضوع نیز Divergence خیلی کمی را ایجاد می‌کند، چون صرفاً در تعداد خیلی کمی از Warp‌ها ریسه‌ای مشغول اجرا کد داخل شرط می‌شود. در آخر تمام این محاسبات نیز، برای این که محاسبات فاز بعد که فیلتر را اعمال می‌کنیم به درستی پیش برود، ریسه‌های بلو را با دستور syncthreads همگام می‌کنیم. این باعث می‌شود که تمام پیکسل‌های مربوط به بخش مذکور در smem به طور کامل لود شوند و در محاسبات آینده خدشهاین ایجاد نشود.

در ادامه، چون هر پیکسل داخل بخش مربوطه که مرزی نیست تمام همسایگی‌اش را دارد، می‌تواند به راحتی مشتق‌های  $x$  و  $y$  اش را محاسبه کند. این کار را اما باید با دقت انجام داد و هر دو مشتق را همزمان با هم و از پیکسل بالا-چپ تا پایین-راست حساب کنیم. این به این دلیل است که به این شکل، خوادن‌های از smem متواالی می‌شوند و Bank Conflict خیلی کمی را خواهیم داشت. پس به این شکل، هر کدام از مشتق‌ها محاسبه می‌شود. در آخر نیز، با استفاده از روش‌های Branchless، بررسی می‌کنیم که هر مؤلفه از مشتق منفی هست یا نه و اگر منفی بود، با استفاده از روش‌های بیتی و Twos Complement آن‌ها را منفی می‌کنیم. (رفتار مذکور را می‌توانید در کد مشاهده کنید) در آخر نیز این دو اندازه‌ی به دست آمده را با هم جمع می‌کنیم و در صورتی که پیکسل مربوطه در خروجی وجود داشت، با تکنیک‌های Branchless آن را در بازه‌ی [0, 255] می‌آوریم. این حاصل جدید را نیز با Threshold



شکل ۱: تصویر  $1000 \times 1500$  با تغییرات روشنایی آن.

مقایسه می‌کنیم و باز به صورت Branchless خروجی را روشن کامل و یا خاموش کامل می‌کنیم و یا حتی تغییری در آن ایجاد نمی‌کنیم. تمام این Branchless بودن‌ها میزان Divergence را خیلی کم می‌کند. در نهایت هم برای این که ریسنهایی که کارشان تمام شده است داده‌های *smem* را خراب نکنند، باز دستور `_syncthreads` را اجرا می‌کنیم.  
 پس به این شکل، با استفاده از این بهبودها، بلوک‌های مختلف به هندل کردن بخش‌های مختلف از تصویر و خروجی آن می‌پردازند و حاصل را تولید می‌کنند. پس فقط کافی است که حاصل تولید شده را به همراه تصویر CPU Brightness Changed در *Biاوریم* و در فایل‌های مناسب ذخیره کنیم.

## ۲.۱ پیاده‌سازی GUI

### ۲ نمونه‌های اجرای برنامه

نمونه‌هایی از اجرای این برنامه را می‌توانید در شکل‌های ۱ تا ۴ مشاهده کنید. همانطور که مشاهده می‌شود، با `offset` مختلف در تغییر روشنایی خروجی حاصل تغییر می‌کند. همچنین، پیاده‌سازی ما برای هر اندازه‌ای از تصویر کار می‌کند، حتی تصاویر 8K (این شکل‌ها را برای کم کردن حجم فایل گزارش نیاورده‌ایم) توجه کنید که در تمام این خروجی‌ها Thresholding با  $t_1 = 20$  و  $t_2 = 240$  انجام شده است.

همچنین می‌توانید اثر بودن و یا نبودن Thresholding را در شکل ۵ مشاهده کنید.

## GPU vs. CPU ۳

برای مقایسه زمان اجرای برنامه‌ی عادی CPU با پیاده‌سازی GPU، یک برنامه‌ی معمول برای این فیلتر با همین قابلیت‌ها را فایل *Code/Code.cpp* نوشته‌ایم. هر دو برنامه را، فقط بخش محاسباتشان را، (و نه بخش کپی کردن داده و گرفتن حافظه و...) را با استفاده از یک High-Resolution Clock



.offset = 50 (ج)

(ب) حاصل اصلی.

.offset = -50 (آ)

شکل ۲: حاصل فیلتر شدهی تصویر  $1000 \times 1500$  با تغییرات روشنایی آن.



.offset = 50 (ج)

(ب) تصویر اصلی.

.offset = -50 (آ)

شکل ۳: تصویر  $1920 \times 1200$  با تغییرات روشنایی آن.

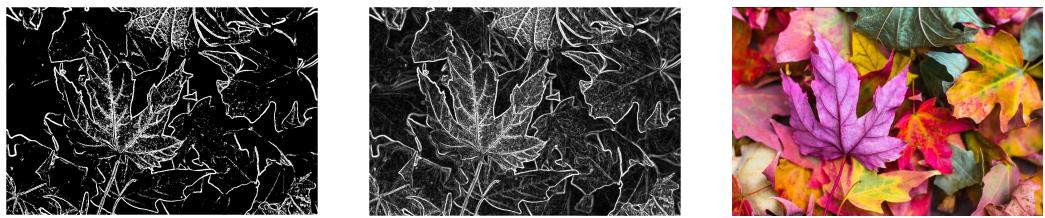


.offset = 50 (ج)

(ب) تصویر اصلی.

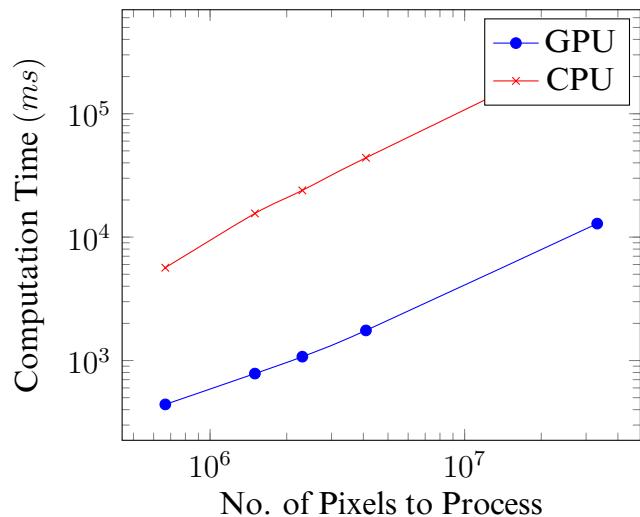
.offset = -50 (آ)

شکل ۴: حاصل فیلتر شدهی تصویر  $1920 \times 1200$  با تغییرات روشنایی آن.



$t_1 = 100, t_2 = 150$  (ج)       $t_1 = 0, t_2 = 255$  (ب)      (ا) تصویر اصلی.

شکل ۵: مقایسه‌ی اثر اضافه کردن Thresholding به خروجی در یک تصویر  $1000 \times 667$



شکل ۶: مقایسه‌ی زمان اجرای پیاده‌سازی‌های GPU و CPU.

مقایسه‌ی می‌کنیم. نتیجه‌ی مقایسه‌ی این دو برنامه را می‌توانید در شکل ۶ مشاهده کنید. همانطور که مشاهده می‌شود، پیاده‌سازی GPU تا  $27/6$  برابر بهتر از پیاده‌سازی معمول CPU عمل می‌کند! پس به هدف خود رسیده‌ایم و یک پیاده‌سازی بهینه داریم.