بسم الله الرحمن الرحیم

موسسه آموزش عالی بعثت

پایان نامه تحصیلی برای دریافت درجه کارشناسی ارشد
رشته مهندسی کامپیوتر گرایش نرم افزار

موازی سازی حذف نویزهای غیرمحلی تصاویر ام آر آی با استفاده از پردازنده گرافیکی

مولف:
مهری سالاری

استاد راهنما:
دکتر فهیمه یزدان پناه

شهریور ۱۳۹۸

موسسه آموزش عالی بعثت کرمان

بخش کامپیوتر

این پایان نامه با عنوان موازی سازی حذف نویزهای غیرمحلی تصاویر ام آر آی با استفاده از پردازنده گرافیکی توسط خانم مهری سالاری دانشجوی رشته مهندسی کامپیوتر گرایش نرم افزار با شماره دانشجویی **۹۴۲۵۴۱۰۲۴** تدوین شده است و در تاریخ **۱۳۹۸/۰۶/۳۱** با درجه متوسط و نمره **۱۵/۵۰** مورد پذیرش هیئت محترم داوران قرار گرفت.

این پایان‌نامه هیچگونه مدرکی به عنوان فراغت از تحصیل دوره کارشناسی ارشد شناخته نمی‌شود.

سمت	نام و نام خانوادگی	امضاء
استاد راهنما	دکتر فهیمه یزدان پناه	
استاد مشاور		
داور اول	دکتر محمد علائی	
داور دوم	دکتر مصطفی قاضی زاده	

نماینده تحصیلات تکمیلی:
نام و نام خانوادگی
امضاء

معاون آموزشی و پژوهشی موسسه:
نام و نام خانوادگی
امضاء

به نام خدا

منشور اخلاق پژوهش

با استعانت از خدای سبحان و با اعتقاد راسخ به اینکه عالم محضر خداست و او همواره ناظر بر اعمال ماست و به منظور انجام شایسته‌ی پژوهش‌های اصیل، تولید دانش جدید و بهسازی زندگانی بشر، ما دانشجویان و اعضای هیات علمی دانشگاه‌ها و پژوهشگاه‌های کشور:

- ☐ تمام تلاش خود را برای کشف حقیقت و فقط حقیقت به کار خواهیم بست و از هرگونه جعل و تحریف در فعالیت‌های علمی پرهیز می‌کنیم.
- ☐ حقوق پژوهشگران، پژوهیدگان (انسان، حیوان، گیاه و اشیاء)، سازمان‌ها و سایر صاحبان حقوق را به رسمیت می‌شناسیم و در حفظ آن می‌کوشیم.
- ☐ به مالکیت مادی و معنوی آثار پژوهشی ارج می‌نهیم، برای انجام پژوهشی اصیل اهتمام ورزیده و از سرقت علمی و ارجاع نامناسب اجتناب می‌کنیم.
- ☐ ضمن پایبندی به انصاف و اجتناب از هرگونه تبعیض و تعصب، در کلیه فعالیت‌های پژوهشی، رهیافتی تعادلانه اتخاذ خواهیم کرد.
- ☐ ضمن امانت‌داری، از منابع و امکانات اقتصادی انسانی و فنی موجود استفاده بهره‌ورانه خواهیم کرد.
- ☐ از انتشار غیراخلاقی نتایج پژوهش نظیر انتشار موازی، همپوشان و چندگانه (تکه‌ای) پرهیز می‌کنیم.
- ☐ اصل محرمانه بودن و رازداری را محور تمام فعالیت‌های پژوهشی خود قرار می‌دهیم.
- ☐ در همه فعالیت‌های پژوهشی به منافع ملی توجه کرده و برای تحقق آن می‌کوشیم.
- ☐ خویش را ملزم به رعایت کلیه بخشنامه‌های علمی رشته خود، قوانین و مقررات، سیاست‌های حرفه‌ای، سازمانی، دولتی و راهبردهای ملی در همه مراحل پژوهش می‌دانیم.
- ☐ رعایت اصول اخلاق در پژوهش را اقدامی فرهنگی می‌دانیم و به منظور بالندگی این فرهنگ، به ترویج و اشاعه‌ی آن در جامعه اهتمام می‌ورزیم.

تعهدنامه

اینجانب مهری سالاریی بـه شـماره دانشـجویی ۹۴۲۵۴۱۰۲۴ دانشـجوی مقطـع کارشناسـی ارشـد رشـته مهندسـی کـامپیوتر گرایش نرم افزار دانشکده کامپیوتر موسسه آموزش عالی بعثت کرمان نویسنده پایان نامه با عنوان موازی سـازی حـذف نویزهای غیرمحلی تصاویر MRI با استفاده از پردازنده کارت گرافیکی تحت راهنمایی خانم دکتر فهیمه یزدان پنـاه تأییـد می‌کنم که این پایان‌نامه نتیجه پژوهش اینجانب می باشد و در عین حال که موضوع آن تکراری نیست، در صورت استفاده از منابع دیگران، نشانی دقیق و مشخصات کامل آن درج شده است. همچنین موارد زیر را نیز تعهد می‌کنم:

۱- برای انتشار تمام یا قسمتی از داده‌ها یا دستاوردهای پایان‌نامه خود در مجـامع و رسـانه‌هـای علمـی اعـم از همـایش‌هـا و مجلات داخلی و خارجی به صورت مقاله، کتاب، ثبت اختراع و .... به صورت مکتوب یا غیرمکتـوب، بـا کسـب مجـوز از موسسه آموزش عالی بعثت کرمان و استاد(ان) راهنما اقدام نمایم.

۲- از درج اسامی افراد خارج از کمیته پایان‌نامه در جمع نویسندگان مقاله‌های مستخرج از پایان‌نامه، بدون مجوز استاد(ان) راهنما اجتناب نمایم و اسامی افراد کمیته پایان نامه را در جمع نویسندگان مقاله درج نمایم.

۳- از درج نشانی یا وابستگی کاری (affiliation) نویسندگان سازمان‌هـای دیگـر (غیـر از موسسـه آمـوزش عـالی بعثـت کرمان) در مقاله‌های مستخرج از پایان‌نامه بدون تأیید استاد(دان) راهنما اجتناب نمایم[1].

۴- کلیه ضوابط و اصول اخلاقی مربوط به استفاده از موجودات زنده یا بافتهای آنها را برای انجام پایان نامه رعایت نمایم.

۵- در صورت اثبات تخلف (در هر زمان) مدرک تحصیلی صادر شده توسط دانشگاه شهید بـاهنر کرمـان از درجـه اعتبـار ساقط و اینجانب هیچ‌گونه ادعایی نخواهم داشت.

کلیه حقوق مادی و معنوی این اثر (مقالات مستخرج، برنامه های رایانه ای، نرم افزارها و تجهیزات ساخته شـده) مطـابق بـا آیین‌نامه مالکیت فکری، متعلق به موسسه آموزش عالی بعثت کرمـان اسـت و بـدون اخـذ اجـازه کتبـی از دانشـگاه قابـل واگذاری به شخص ثالث نیست. همچنین استفاده از اطلاعات و نتایج این پایان نامـه بـدون ذکـر مرجـع مجـاز نمـی باشـد. چنانچه مبادرت به عملی خلاف این تعهدنامه محرز گردد، دانشگاه شهید باهنر کرمان در هر زمان و به هر نحو مقتضی حق هرگونه اقدام قانونی را در استیفای حقوق خود دارد.

نام و نام خانوادگی دانشجو: مهری سالاری

امضا و تاریخ:

تقدیم به:

پدرم ، که عالمانه به من آموخت تا چگونه در عرصه زندگی، ایستادگی را تجربه نمایم.

مادرم، دریای بیکران فداکاری و عشق که وجودم برایش همه رنج بود و وجودش برایم همه مهر.

همسرم، اسطوره زندگی‌ام ، پناه خستگی‌ام و امید بودنم.

تشکر و قدردانی:

بدین‌وسیله بر خود لازم می‌دانم مراتب تشکر، قدردانی و سپاس خود را از اساتید عزیز،فرهیخته و گران‌قدرم:

سرکار خانم دکتر فهیمه یزدان پناه که، همواره با زحمات ، حمایت‌های بی‌دریغ و راهنمایی‌های ارزشمندشان ، ره گشای مسائل من بودند.

چکیده:

امروزه فیلتر (NLM) ابزار غیرموضعی به‌عنوان الگوریتم پیشرفته برای حذف نویز در تصاویر ام.آر.آی موردتوجه است. که به دسته‌ای از روش‌های حذف نویز مربوط می‌شود که برای آن شدت حذف نویز در یک پیکسل به‌عنوان میانگین اندازه‌گیری شده شدت‌ها در پیکسل‌های درست انتخاب‌شده است. منطق محاسبه اندازه NLM به دو مقوله متفاوت بستگی دارد: اولاً، اندازه‌ها در میان پیکسل‌ها به شباهت در یک ناحیه از پیکسل‌ها بجای پیکسل‌های تکی تعریف می‌شود. با این روش انتخاب بافت‌های موضعی تصویر حاصل می‌شود. علی‌رغم موفقیتی که با الگوریتم NLM بسیار طبیعی خود به دست آورده است، ازنقطه‌نظر محاسباتی بسیار موردتوجه است، برای کاربردهای به‌روز مناسب نیست، حتی برای پردازش آفلاین و برای تنظیم پارامترهایش بدون واردکردن محدودیت‌های شدید در ناحیه‌های پیکسل مشکل دارد. برای فائق آمدن بر این مشکلات چندین تصحیح سازی در این پایان‌نامه پیشنهاد گردیده است. در طی دهه گذشته، پردازنده گرافیکی به‌عنوان پلتفرم سخت‌افزاری موردتوجه قرارگرفته‌اند که مکمل دستگاه‌های پردازش مرکزی در کامپیوترهای مدرن می‌باشند. این به خاطر هزینه پائین موردهای پردازنده گرافیکی است که حتی در اکثر کامپیوترهای شخصی وجود دارند و همین‌طور به خاطر پیشرفت‌های فنّاورانه روزافزون است که عملکرد محاسباتی را تا بیش از دو برابر سریع‌تر از پردازنده اصلی یا برای الگوریتم‌های مناسب برای موازی‌سازی بزرگ بهبود داده است. از طرف دیگر ماهیت شدید موازی تعداد زیادی از الگوریتم‌های حذف نویز با ویژگی‌های سخت‌افزاری پردازنده گرافیکی هماهنگی و سازگاری دارند که آن‌ها را ابزاری کامل برای افزایش سرعت الگوریتم تبدیل می‌سازند. چند آزمایش روی تصاویر MR می‌شود تا کارایی الگوریتم در چندین زمینه را نشان دهد. مقایسه‌ها و زمان محاسبه‌ای الگوریتم‌های NLM با استفاده از پردازنده گرافیکی نیز ارائه می‌شود.

واژگان کلیدی: نویز غیر محلی، حذف نویز تصویر، تصویربرداری مغناطیسی، واحد پردازش گرافیکی.

فهرست مطالب

عنوان **صفحه**

فهرست جدول‌ها

عنوان **صفحه**

فهرست شکل‌ها

عنوان **صفحه**

فصل اول:
مقدمه و کلیات تحقیق

۱-۱ مقدمه

شاید بتوان به زبان ساده، هدف اصلی علم پردازش تصویر را به دست آوردن اطلاعات موردنیاز از تصاویر یا قابل‌استفاده کردن آن‌ها در کاربردی خاص دانست. به‌عنوان‌مثال می‌توان به جدا کردن عنبیه از تصویر چشم برای شناسایی هویت افراد اشاره کرد. کیفیت پایین تصویر، بازده فرآیند پردازش را به‌صورت قابل‌توجهی می‌کاهد و حتی ممکن است منجر به تصمیم‌گیری نادرست شود. افزایش کیفیت تصاویر شامل دودسته‌ی کلی ارتقاء یا بهسازی تصویر و بازسازی تصویر است. ارتقای تصویر شامل روش‌هایی شهودی است که بدون دانستن مدل خرابی تصویر، سعی در افزایش کیفیت آن برای کاربرد موردنظر دارد. ولی بازسازی تصویر شامل روش‌های عینی است که با دانستن مدل خرابی و اعمال عکس آن، تصویر را بهبود می‌بخشد [۱]. یکی از زمینه‌های مهم افزایش کیفیت تصاویر، حذف نویز تصویر است. این زمینه را می‌توان از هر دو دیدگاه بهسازی و بازسازی تصویر بررسی کرد. تفاوت حذف نویز با روش‌های کلاسیک بازسازی تصویر در این است که باوجود دانستن مدل نویز، امکان اعمال عکس مدل خرابی وجود ندارد و ازاین‌رو باوجود داشتن مدل نویز در بیشتر مواقع باید روشی شهودی برای افزایش کیفیت تصویر ارائه کرد. نویز به سیگنال‌های ناخواسته‌ای اطلاق می‌شود که باعث تداخل در اطلاعات اصلی شده و آن‌ها را تحت تأثیر قرار می‌دهد. تصاویر ممکن است در سه مرحله‌ی اخذ، انتقال و نگهداری دچار نویز شوند[۱]. در مرحله‌ی اخذ، سنسورهای معیوب دوربین یا خود ماهیت صحنه می‌تواند باعث نویز شود. به‌عنوان‌مثال در تصویربرداری از یک ناو جنگی در دریا، اشعه خورشید به سطح دریا می‌تابد و ازآنجا به لنز دوربین منعکس می‌شود چون شدت نور منعکس‌شده به دلیل وجود امواج در سطح آب تغییر می‌کند، نویز اخذ تصویر ایجاد می‌شود. در مرحله‌ی انتقال، به دلیل وجود کانال‌های نویزی تصویر دچار نویز می‌شود. در مرحله‌ی ذخیره‌سازی نیز به دلیل وجود بیت‌های معیوب حافظه یا اغتشاشات خارجی، نویز ایجاد می‌شود. از سایر منشأهای نویز در تصاویر می‌توان به خطاهای زمان‌بندی در تبدیل آنالوگ به دیجیتال و خطای سنکرون در ذخیره دیجیتال شود.

امروزه روش‌های حذف نویز بسیاری وجود دارند. از روش‌های حوزه، فرکانس گرفته تا روش‌های پیشرفته اخیر همچون فیلترینگ دوطرفه تبدیل موجک و پردازش چند دقتی که با پیشرفت سریع دستگاه‌های تصویربرداری دیجیتال مدرن و افزایش کاربردهای گسترده در زندگی روزمره نیاز به الگوریتم‌های حذف نویز جدید برای دستیابی به کیفیت بالای تصویر افزایش‌یافته است. به‌ویژه روش‌های حذف نویز حوزه تبدیل که سیگنال در آن به‌طور پراکنده نمایش داده می‌شود. این روش‌ها گروه‌هایی از ضرایب تبدیل با دامنه بالا را که بیشتر انرژی سیگنال بدون نویز، در آن‌ها متمرکزشده‌اند، حفظ کرده و باقیمانده ضرایب را که بیشتر ناشی از نویز می‌باشند، حذف می‌کنند. در این روش‌ها

سیگنال بدون نویز را می‌توان به‌طور مؤثری تخمین زد [۳].

تبدیل‌های چند دقتی می‌توانند به یک پراکندگی خوب برای جزئیاتی همچون لبه‌ها و تکین‌ها دست یابند. اساساً الگوریتم‌های حذف نویز چند دقتی بر مبنای آستانه گذاری موجک[۱] می‌باشند. عملکرد این روش‌ها بر این بنا می‌باشند که همه ضرایب موجک را که از یک مقدار معین (آستانه) کمتر باشند، به سمت صفر مقداردهی می‌شود. بااین‌حال وجود الگوریتم‌های مبتنی بر موجک و پردازش چند دقتی به‌تنهایی برای نمایش تمام جزئیات کافی نیست[۴-۳-۲]. برای غالب شدن بر این مسئله ماریسون و پارکس[۲] یک طرح حذف نویز بر مبنای آنالیز مؤلفه اصلی پیشنهاد داده‌اند [۵]. لی‌زنگ و همکارانش الگوریتم فوق را با استفاده از تطبیق بلوک گسترش دادند [۶]. به‌هرحال با افزایش سطح نویز محاسبه درست آنالیز مؤلفه اصلی با مشکل مواجه می‌شود. یک نقطه‌ضعف بزرگ و اساسی روش‌های آنالیز مؤلفه اصلی در این است که هر بار محاسباتی بالایی را به پردازشگر اعمال می‌کنند و زمان زیادی برای تخمین تصویر بدون نویز نیاز دارند. اخیراً یک روش حذف نویز قدرتمند توسط کاستادین دابو بر مبنای تطبیق بلوک و فیلترینگ سه‌بعدی پیشنهادشده است. در این الگوریتم، نمایش پراکنده به‌وسیله گروه‌بندی تکه‌های مشابه دوبعدی تصویر، درون یک آرایه سه‌بعدی انجام می‌شود. دو تکنیک اصلی تطبیق بلوک و فیلتر سه‌بعدی، گروه‌بندی و فیلترینگ اشتراکی است. یک بلوک تصویر مرجع [۹] با اندازه ثابت تعیین‌شده و بلوک‌های مشابه، توسط تطبیق بلوک به دست می‌آیند.

۱-۲ معرفی پردازنده گرافیکی

واحد پردازش گرافیکی ابزاری اختصاصی برای رندر کردن گرافیکی (طبیعی جلوه دادن تصویر) در کامپیوترهای شخصی، ایستگاه‌های کاری و یا در استوانک‌های بازی است. این واحد گاهی اوقات واحد پردازنده بصری نیز نامیده می‌شود. درواقع نحوه‌ی عملکرد پردازنده گرافیکی در پردازش اطلاعات همانند پردازنده اصلی است ولی وظیفه اصلی آن، پردازش اطلاعات مرتبط با تصاویر است. به‌عبارت‌دیگر، پردازنده گرافیکی، مغز یک کارت گرافیکی است و بسیاری از مشخصات یک کارت گرافیکی به آن وابسته است. به‌عنوان‌مثال، میزان حافظه‌ای که یک کارت گرافیکی پشتیبانی می‌کند یا نوع درگاه ارتباطی کارت گرافیکی، به آن بستگی دارد [۴]. در اینجا لازم است قبل از هر چیز با سیر تکامل پردازنده‌های گرافیکی و همچنین رویکردهای گوناگونی که به این تراشه‌های کوچک اجازه می‌دهند رقبای قدرتمند خود یعنی پردازنده‌های اصلی را در بسیاری از کارهای مهم پشت سر بگذارند، آشنا شویم.

---

[۱] Wavelet
[۲] Marison and Parks

۱-۳ معرفی کودا

کودا ساخته‌شده توسط ان ویدیا یک معماری محاسباتی موازی داده و همه‌منظوره است، و به‌طورمعمول در پژوهش‌های مربوط به کاربردهای پردازنده گرافیکی به‌منظور افزایش کارایی مورداستفاده قرار می‌گیرد. این مدل برنامه‌نویسی از اجرای عملیات یکپارچه بر روی پردازنده اصلی و پردازنده گرافیکی پشتیبانی می‌کند. اصطلاح Host به پردازنده اصلی و حافظه آن Host memory اشاره دارد درحالی‌که Device به پردازنده گرافیکی و حافظه آن Device memory اشاره می‌کند [۵].

۱-٤ فیلترNLM چندگزینه‌ای در مالتی جی پی یو[۱]

ما الگوریتم‌های NLM مالتی-جی پی یو را با استفاده از ابزار کودا برای جی پی یوهای ان ویدیا ساخته‌ایم. کودا یک API شکل کاربرد معمول می‌یابد را و کنترل روی اینکه چگونه اقدام در سخت‌افزار پردازنده گرافیکی محاسبه شود، ارائه می‌دهد.

در کودا سیستم از یک هاست (سی پی یو) و یک یا چند دستگاه دیگر تشکیل می‌شود، که خیلی زیاد با پردازشگرها موازی می‌باشند. این هاست می‌تواند داده‌های کاربردی را بین هاست و حافظه دستگاه انتقال دهد، و تعداد متغیری از عملیات (که کرنل نامیده می‌شوند) برای اجرا روی دستگاه را فعال و تحریک سازد. کرنل‌ها ویژگی مهم موازی‌سازی داده‌ای را نشان می‌دهند، باعث می‌شوند داده‌ها به‌طور هم‌زمان توسط تعداد زیادی از Threedها اجرا شوند.

نخ‌هایی که همان کرنل را اجرا می‌کنند در چندین سطح شبکه‌ای از بخش‌های نخ سازمان‌دهی می‌شوند. هر بخش نخ تا ۱۰۲۴ نخ دارد که ویژگی‌های دستگاه بستگی دارد. نمونه‌های درون دستگاهی[۲] مانند (threadldz،threadldy،thread ldx) و (blockldz،block ldy،block ldx) شبکه نخ و پیکربندی‌های شاخص بلوک به هر نخ را ارائه می‌دهند و آن‌ها برای تقسیم فعالیت بین نخ‌ها مورداستفاده قرار داده می‌شوند. نخ‌ها در یک بلوک نخ یکسان با همان مالتی پردازشگر[۳] اجرا و می‌توانند در یکدیگر هماهنگ‌سازی شوند و همان حافظه موجود را مورداستفاده قرار دهند. بخش‌های متوالی از ۳۲ Threed به‌گونه‌ای تعریف می‌شوند که قسمتی از یک بسته‌بندی باشند که در آن تمام نخ‌ها در یک دستور موازی یکسان را اجرا می‌کنند. متأسفانه وقتی نخ‌ها در یک wrap مسیرهای اجرا متفاوتی دارند، گاهی اوقات

[۱] Multi-GPU
[۲] Bult-in
[۳] SM

subpression بعضی نخ‌ها لازم می‌شود تا اجزاء کامل به دست آید باعث نقص اجرایی می‌شود [۶].

کودا انواع متفاوت مموری (حافظه) را پشتیبانی می‌کند، حافظه اصلی۱ بزرگ‌ترین حافظه است. اما تأخیر۲ بالایی دارد. اساساً برای ذخیره‌سازی داده‌های خروجی و ورودی مورداستفاده قرار داده می‌شود معمولاً عملکرد کرنل‌های کودا را مجدد می‌سازد اگر حافظه‌های دیگر مورداستفاده قرار نگیرد. هم حافظه ثابت و هم حافظه مشترک روی حافظه‌های آن چیپی۳ می‌باشند.

حافظه ثابت یک حافظه کوچک خواندنی۴ است، تأخیر پائین و دسترسی باند بالا را پشتیبانی می‌کند وقتی تمام نخ‌ها به‌طور هم‌زمان به یک مکان دسترسی دارند حافظه مشترک را می‌توان به قسمت‌های نخ اختصاص داد و درروش کاملاً موازی با سرعت‌بالا قابل‌دسترسی می‌باشند وقتی تمام tlveadsها در یک بخش thread ۵ می‌توانند حافظه مشترک خود را بخوانند و بنویسند، روش بسیار مؤثری برای نخ‌ها وجود دارد تا داده‌های ورودی خود و نتایج آن را به اشتراک‌گذارند.

برای به دست آوردن مزیت عمده از مدارهای پردازنده گرافیکی موجود ما یک ورژن جدید از مالتی-جی پی یو۶ را ساخته‌ایم. همان‌طور که روی مقدار داده‌هایی که هر پردازنده گرافیکی می‌تواند پردازش کند محدودیت وجود دارد، قسمت‌هایی از داده‌های اصلی در هر درخواست کرنل با توجه به حافظه موجود پردازش می‌شوند. علاوه بر این، برای اطمینان از اینکه هر پیکسل را بتوان درروش یکسان مورد ارزیابی قرارداد، ما یک کپی اولیه بزرگ‌تر از داده‌های تصویر داریم که در آن حاشیه‌های اضافی وجود دارد. ما این فضای اضافی را با انعکاس تصویر حاشیه‌ای پر می‌کنیم. برای اینکه باعث شویم تمام عملیات در تمام پردازنده گرافیکی های موجود اجرا شود، ما به این نیاز داریم تا حجم کل را به قسمت-هایی متناسب با تعداد پردازنده‌های گرافیکی موجود در دستگاه تقسیم‌بندی نماییم [۷].

۱-۵ تعریف مسئله و بیان سؤال‌های اصلی تحقیق

حذف نویز تصویر فرایندی بسیار مهم در عکس‌برداری پزشکی است. اینکه با حذف نویز می‌توان یکپارچگی تصویر را حفظ کرد یک موضوع بسیار مهم به‌طور خاص در عکس‌برداری از طریق

---

۱ Global memory
۲ latency
۳ ON-Chip
۴ Read-only
۵ thread block
۶ Multi-GPU

سونوگرافی و از طریق ام.آر.آی[۱] است که به خاطر وجود ساختارهای سیگنالی است که به‌ندرت بالای سطح نویز قابل‌تشخیص است.

امروزه فیلتر (NLM) ابزار غیرموضعی به‌عنوان الگوریتم پیشرفته برای حذف نویز در تصاویر ام.آر.آی موردتوجه است. که به دسته‌ای از روش‌های حذف نویز مربوط می‌شود که برای آن شدت حذف نویز در یک پیکسل به‌عنوان میانگین اندازه‌گیری شده شدت‌ها در پیکسل‌های درست انتخاب‌شده است. منطق محاسبه اندازه NLM به دو مقوله متفاوت بستگی دارد: اولاً، اندازه‌ها در میان پیکسل‌ها به شباهت در یک ناحیه از پیکسل‌ها بجای پیکسل‌های تکی تعریف می‌شود. با این روش انتخاب بافت‌های موضعی تصویر حاصل می‌شود. علی‌رغم موفقیتی که با الگوریتم NLM بسیار طبیعی خود به دست آورده است، ازنقطه‌نظر محاسباتی بسیار موردتوجه است، برای کاربردهای به‌روز مناسب نیست، حتی برای پردازش آفلاین و برای تنظیم پارامترهایش بدون واردکردن محدودیت‌های شدید در ناحیه‌های پیکسل مشکل دارد. برای فائق آمدن بر این مشکلات چندین تصحیح سازی در این پایان‌نامه پیشنهاد گردیده است که نسخه oplimized block wise، روش بر پایه ترکیب‌سازی امواج باند پائین[۲] این پیشنهاد را شامل می‌شوند [۱].

اکثر تصاویر ام.آر.آی تحت تأثیر نویز ریشن[۳] قرار می‌گیرند که از فضای K – [۴] ناشی می‌شوند که در آنجا نویز به‌صورت گاوسی است. یک استثناء قابل‌توجه توسط مارپیچی کردن‌های آرایه فازی و روش‌های جذب موازی مانند حساسیت رمزگذاری برای ام.آر.آی سریع و جذب نسبتاً موازی خودتنظیم ساز[۵] وجود دارد. در این حالت K-space تحت تأثیر نویز با توزیع Kی غیر مرکزی قرار می‌گیرد که تصاویر ام.آر.آی با نویز ریشن[۶] غیر هماهنگ را تولید می‌کند. در ارتباط با واریانس نویز غیر هماهنگ تخمین و برآورد سازی نویز موضعی مطرح‌شده است.

در طی دهه گذشته، پردازنده گرافیکی[۷] به‌عنوان پلتفرم سخت‌افزاری موردتوجه قرارگرفته‌اند که مکمل دستگاه‌های پردازش مرکزی[۸] در کامپیوترهای مدرن می‌باشند. این به خاطر هزینه پائین موردهای پردازنده گرافیکی است که حتی در اکثر کامپیوترهای شخصی وجود دارند و همین‌طور به خاطر

---

[۱] MRI
[۲] wavelet sub-bands
[۳] Rician
[۴] K-space
[۵] GRAPPA
[۱] Rician
[۲] GPU
[۳] CPU

پیشرفت‌های فنّاورانه روزافزون است که عملکرد محاسباتی را تا بیش از دو برابر سریع‌تر از پردازنده اصلی یا برای الگوریتم‌های مناسب برای موازی‌سازی بزرگ بهبود داده است. از طرف دیگر ماهیت شدید موازی تعداد زیادی از الگوریتم‌های حذف نویز با ویژگی‌های سخت‌افزاری پردازنده گرافیکی هماهنگی و سازگاری دارند که آن‌ها را ابزاری کامل برای افزایش سرعت الگوریتم تبدیل می‌سازند. درنتیجه الگوریتم‌های NLM به‌طور ویژه برای ساختارهای پردازنده گرافیکی ساخته شد.

۱-٦ ضرورت انجام تحقیق

ان ویدیا با معرفی فناوری کودا در زمینه‌های مختلف محاسباتی تحول جدیدی به وجود آورد [۴]. این فناوری اجازه دسترسی مستقیم به هسته‌های قرارگرفته بر روی کارت‌های گرافیکی را به توسعه‌دهندگان داد و درنتیجه انجام بسیاری از محاسبات عددی پیشرفته بر روی آن‌ها تسهیل شد[۶]. با استفاده از معماری کودا می‌توان برنامه‌هایی را با زبان C نوشته و سپس روی پردازنده گرافیکی اجرا نمود. درباره این معماری گفته می‌شود؛ کودا معماری است که به‌جای محدود کردن شما توسط کارایی یک سری کتابخانه، اجازه می‌دهد کار موردنظرتان را انجام دهید. درگذشته، نوشتن نرم‌افزار برای پردازنده گرافیکی به این معنی بود که به زبان پردازنده گرافیکی برنامه نوشته شود اما کودا اجازه می‌دهد با زبان-های معمول برنامه‌ای ایجاد گردد که بتواند روی پردازنده گرافیکی نیز اجرا شود. همچنین به دلیل آن‌که کودا می‌تواند نرم‌افزار را به‌صورت مستقیم روی سخت‌افزار گرافیکی کامپایل کند، کارایی به‌دست‌آمده نیز افزایش پیدا می‌کند[۷].

۱-۷ اهداف پژوهش

۱- فیلترهای NLM و مشخصات آن‌ها را معرفی می‌کند.

a. روی متدلوژی NLM خاص مطرح‌شده در این پایان‌نامه تأکید دارد. و عملکرد پردازنده گرافیکی آن را معرفی می‌کند.

۲- شامل چند آزمایش روی تصاویر MRI می‌شود تا کارایی الگوریتم در چندین زمینه را نشان دهد. مقایسه‌ها و زمان محاسبه‌ای الگوریتم‌های NLM پیشرفته پردازنده گرافیکی نیز در این بخش ارائه می‌شود. درنتیجه الگوریتم‌های NLM به‌طور ویژه برای ساختارهای پردازنده گرافیکی ساخته شد.

۳- افزایش سرعت اجرای حذف نویز با استفاده از برنامه‌نویسی بر پایه پردازنده گرافیکی

۴- استفاده از سطح بالایی از موازی‌سازی پردازش‌ها از طریق اجرای هزاران رشته پردازشی سخت‌افزاری جهت حل مشکل زمان طولانی محاسبات به‌عنوان یکی از عوامل بازدارنده و همچنین افزایش سرعت و پهنای باند.

۵- ازآنجاکه حافظه کارت‌های گرافیکی با سرعت زیادی افزایش می‌یابد، استفاده از حافظه آن‌ها به‌عنوان حافظه موقت، موجب کاهش ترافیک مطالعات در گذرگاه کارت PCI می‌شود و درنتیجه کارایی افزایش می‌یابد.

فراهم نمودن امکان تبادل ناهمگام اطلاعات بر پایه پردازنده‌های گرافیکی.

۱-۸ جنبه جدید بودن و نوآوری در طرح

۱- فیلترهای NLM و مشخصات آن‌ها را معرفی می‌کند.

۲- روی متدلوژی NLM خاص مطرح‌شده در این پایان‌نامه تأکید دارد. و عملکرد پردازنده گرافیکی آن را معرفی می‌کند.

۳- شامل چند آزمایش روی تصاویر MR می‌شود تا کارایی الگوریتم در چندین زمینه را نشان دهد. مقایسه‌ها و زمان محاسبه‌ای الگوریتم‌های NLM پیشرفته پردازنده گرافیکی نیز در این بخش ارائه می‌شود. درنتیجه الگوریتم‌های NLM به‌طور ویژه برای ساختارهای پردازنده گرافیکی ساخته شد.

۱-۹ سؤالات و فرضیه‌های پژوهش

۱- با استفاده از برنامه‌نویسی بر پایه کارت‌های گرافیکی می‌توان سرعت اجرای حذف نویزهای محلی تصاویر ام.آر.آی را افزایش داد.

۲- استفاده از برنامه‌نویسی بر پایه کارت‌های گرافیکی می‌تواند الگوریتم NLM را به‌خوبی سایر روش‌ها اجرا نماید.

۱-۱۰ پیش‌فرض‌های پژوهش

تعداد هسته‌های مجتمع در یک پردازنده گرافیکی یک عامل کلیدی است که بر کارایی پردازنده گرافیکی تأثیر می‌گذارد. در سال‌های اخیر، تعداد این هسته‌های مجتمع در یک تراشه پردازنده گرافیکی به‌سرعت در حال افزایش است. به‌عنوان‌مثال، جدیدترین سیستم NVIDIA GTX ۹۸۰ تا ۲۰۴۸ هسته کودا دارد. علاوه بر تعداد هسته‌ها، معماری پردازنده گرافیکی به‌سرعت در حال تکامل است. تولیدکنندگان پردازنده گرافیکی در حال تلاش برای مخفی سازی بیشتر مشخصات سخت‌افزاری هستند به‌طوری‌که درنهایت برنامه‌نویسان بتوانند کدهای پردازنده گرافیکی خود را راحت‌تر بنویسند. در آینده نزدیک، هسته‌های بیشتری در یک تراشه پردازنده گرافیکی مجتمع خواهند شد. هسته‌های بیشتر بدین معنی است که یک پردازنده گرافیکی از موازی‌سازی بیشتری می‌تواند پشتیبانی کند. بااین‌حال، در حال حاضر سخت‌افزار پردازنده گرافیکی در نیازهای محاسباتی کمی پیشرو است، بدین معنی که سخت‌افزار پردازنده گرافیکی ممکن است منابع محاسباتی اضافی را برای برخی از کاربردهای خاص عرضه کند. ازاین‌رو، واحد پردازش گرافیکی ممکن است راه چاره‌ای برای بهبود بیشتر و بهتر عملکرد بعضی از کاربردهای علمی مانند حذف نویزهای محلی تصاویر ام.آر.آی باشد.

۱-۱۱ ساختار پایان‌نامه

پایان‌نامه به این شرح تدوین‌یافته است، در فصل ۲، ابتدا مفاهیم و ادبیات موضوع توضیح داده‌شده و سپس به معرفی کارهای مرتبط و مشابه پرداخته می‌شود. در فصل ۳ توضیح کاملی در مورد رویکرد مورداستفاده در این تحقیق که رویکردی مبتنی بر پایه پردازش کارت‌های گرافیکی است داده می‌شود. به عبارتی فصل ۳ روش پیشنهادی را ارائه می‌دهد. در فصل ۴ به ارزیابی و تحلیل شبیه‌سازی‌ها و پیاده-سازی‌ها پرداخته و درنهایت در فصل ۵ به نتیجه‌گیری و بررسی کارهای آتی خواهیم پرداخت.

فصل دوم:
مروری بر تحقیقات انجام شده

۲-۱ مقدمه

در سال‌های اخیر، پردازش موازی و محاسبات موازی به یک موضوع مهم درزمینه‌ی علوم کامپیوتر تبدیل‌شده‌اند. هدف اصلی پردازش موازی انجام محاسبات سریع‌تر از یک پردازنده با استفاده از تعدادی پردازنده به‌صورت هم‌زمان است. این رویکرد تأثیر زیادی بر تمام فعالیت‌های وابسته به محاسبات داشته است. نیاز به راه‌حل‌های سریع‌تر و حل مسائل با اندازه‌های بزرگ‌تر در طیف وسیعی از کاربردها مانند مدل‌سازی و شبیه‌سازی دستگاه‌های بزرگ، پردازش تصویر، هوش مصنوعی و غیره مطرح است. ازاین‌رو، تکامل معماری‌های کامپیوتر به سمت تعداد هسته‌های بیشتر تأییدی بر این موضوع است که موازی‌سازی روش انتخابی برای بالا بردن سرعت یک الگوریتم است [۲].

۲-۲ تکامل محاسبات پردازنده گرافیکی

رندر[۱] صحنه‌های گرافیکی با مشخصات زیاد کار بسیار مشکلی است که ذاتاً، به موازی‌سازی بسیاری نیاز دارد. یک برنامه‌نویس گرافیکی برنامه خود را به‌صورت تک نخی می‌نویسد که فقط یک نقطه می‌کشد و پردازنده گرافیکی نمونه‌های مختلفی از این نخ را به‌صورت موازی اجرا می‌کند که درنتیجه چندین پیکسل به‌صورت موازی کشیده می‌شود. برنامه‌های گرافیکی با زبان‌های سایه زنی مثل Cg[۲] و HLSL[۳] نوشته می‌شوند. برنامه‌های محاسباتی پردازنده گرافیکی که با زبان‌های C یا C++ ، با مدل‌های محاسباتی موازی کودا یا برنامه‌های کاربردی محاسبات موازی که به‌وسیله کودا القاشده‌اند مثل DirectX[۴] یا OpenCL[۵] نوشته می‌شوند شفافیت را روی محدوده وسیعی از موازی‌سازی مقیاس می‌کنند. مقیاس‌پذیری نرم‌افزار نیز موجب شد که پردازنده‌های گرافیکی بتوانند موازی‌سازی و کارایی خود را با افزایش چگالی ترانزیستورها به‌سرعت افزایش دهند.

جدول (۲-۱) نقطه عطف توسعه تکنولوژی پردازنده‌های گرافیکی ان ویدیا که تکامل گرافیک‌های یکپارچه و محاسبات پردازنده گرافیکی را هدایت می‌کند نشان می‌دهد. تعداد ترانزیستورهای پردازنده گرافیکی هر ۱۸ ماه با افزایش تراکم نیمه‌هادی به‌صورت نمایی افزایش می‌یابد؛ تقریباً دو برابر می‌شود. از معرفی آن‌ها از سال ۲۰۰۶ هسته‌های محاسبات موازی کودا در هر پردازنده گرافیکی هر ۱۸ ماه دو برابر می‌شود [۸].

[۱] Rendering (computer graphics)
[۲] short for C for Graphics
[۳] HLSL it removed many ۳D components of a GPU language
[۴] DirectX is a collection of application programming interfaces (APIs) for handling tasks related to multimedia
[۵] Open Computing Language

جدول (۲-۱) نقطه عطف توسعه تکنولوژی پردازنده‌های گرافیکی ان ویدیا [۸]

Table 1. NVIDIA GPU technology development.

Date	Product	Transistors	CUDA cores	Technology
1997	RIVA 128	3 million	—	3D graphics accelerator
1999	GeForce 256	25 million	—	First GPU, programmed with DX7 and OpenGL
2001	GeForce 3	60 million	—	First programmable shader GPU, programmed with DX8 and OpenGL
2002	GeForce FX	125 million	—	32-bit floating-point (FP) programmable GPU with Cg programs, DX9, and OpenGL
2004	GeForce 6800	222 million	—	32-bit FP programmable scalable GPU, GPGPU Cg programs, DX9, and OpenGL
2006	GeForce 8800	681 million	128	First unified graphics and computing GPU, programmed in C with CUDA
2007	Tesla T8, C870	681 million	128	First GPU computing system programmed in C with CUDA
2008	GeForce GTX 280	1.4 billion	240	Unified graphics and computing GPU, IEEE FP, CUDA C, OpenCL, and DirectCompute
2008	Tesla T10, S1070	1.4 billion	240	GPU computing clusters, 64-bit IEEE FP, 4-Gbyte memory, CUDA C, and OpenCL
2009	Fermi	3.0 billion	512	GPU computing architecture, IEEE 754-2008 FP, 64-bit unified addressing, caching, ECC memory, CUDA C, C++, OpenCL, and DirectCompute

اوایل دهه نود پردازنده‌های گرافیکی وجود نداشتند. کنترل‌کننده‌های آرایه‌های ویدیوی گرافیکی نمایش‌های گرافیکی دوبعدی برای کامپیوترهای شخصی تولید می‌کردند تا به رابط‌های گرافیکی، سرعت بخشند. در ۱۹۹۷ ان ویدیا تک تراشه شتاب‌دهنده گرافیکی را برای بازی‌ها و تجسم سازی برنامه‌های کاربردی سه‌بعدی منتشر می‌کند. این تراشه توسط MicrosoftDirect۳D و OpenGL برنامه‌نویسی شده بود. تکامل پردازنده‌های گرافیکی مدرن شامل اضافه کردن تدریجی قابلیت برنامه‌نویسی، از پایپلاین‌های با توابع ثابت، پراسسورهای ریز برنامه‌نویسی شده، پراسسورهای قابل پیکربندی، پراسسورهای قابل برنامه‌نویسی تا پراسسورهای مقیاس‌پذیر موازی است [۸].

اولین پردازنده گرافیکی، Geforce۲۵۶ بود، که پردازنده گرافیکی تک تراشه‌ای سه‌بعدی زمان واقعی است که در سال ۱۹۹۹ معرفی شد و شامل تمام ویژگی‌های گران‌ترین ایستگاه‌های کاری سه‌بعدی خط لوله گرافیکی آن زمان بود. این پردازنده شامل تبدیل رأس ۳۰ بیتی ممیز شناور قابل پیکربندی، پراسسورهای تولید روشنایی، خط لوله Pixel-Fragment[۱] صحیح قابل پیکربندی بود که توسط برنامه‌های کاربردی OpenGL و DirectX۷ برنامه‌نویسی شده بود. اولین پردازنده‌های گرافیکی از حساب ممیز شناور برای محاسبه رئوس و هندسه سه‌بعدی استفاده می‌کردند، سپس از آن برای به دست آوردن روشنایی پیکسل و مقادیر رنگ استفاده کردند تا بتوانند

[۱] a *fragment* is the data necessary to generate a single ***pixel***'s worth of a drawing primitive in the frame buffer

محدوده صحنه‌های با پویایی زیاد را مدیریت و برنامه‌نویسی را ساده کنند. زمانی که سایه زن‌های قابل برنامه‌نویسی ظهور پیدا کردند. پردازنده‌های گرافیکی قابل برنامه‌نویسی‌تر و انعطاف‌پذیرتر شدند. در ۲۰۰۲، اولین پردازنده رأس قابل برنامه‌نویسی معرفی شد که برنامه‌های سایه زنی رأس را با خط لوله Pixel-Fragment ۳۰ بیتی ممیز شناور قابل برنامه‌نویسی اجرا می‌کرد و با OpenGL و DirectX۸ برنامه‌نویسی شده بود. ATI Radeon ۹۷۰۰ در سال ۲۰۰۲ معرفی شد و شامل یک پردازنده Pixel-Fragment ۰۴ بیتی قابل برنامه‌نویسی با OpenGL و DX۹ بود. Geforce FX و Geforce ۶۸۰۰ شامل پردازنده‌ای Pixel-Fragment ممیز شناور ۳۰ بیتی و پردازنده‌های رأس قابل برنامه‌نویسی بودند که با Cg و DX۹ و OpenGL برنامه‌نویسی شده بودند. این پردازنده‌ها چند نخی بودند و یک نخ را می‌ساختند و همچنین یک برنامه نخی برای هر رأس و هر Pixel-Fragment اجرا می‌کردند. معماری هسته پردازنده مقیاس‌پذیر Geforce ۶۸۰۰ پیاده‌سازی پردازنده‌های گرافیکی چندگانه با تعداد هسته‌های پردازنده متفاوت را آسان می‌کند. توسعه زبان Cg برای برنامه‌نویسی پردازنده گرافیکی، مدل برنامه‌نویسی موازی مقیاس‌پذیر را برای رئوس ممیز شناور قابل برنامه‌نویسی و پردازنده‌های Pixel-Fragment در Geforce FX و Geforce ۶۸۰۰و پردازنده‌های گرافیکی دوبعدی فراهم کرد. برنامه Cg به برنامه C تک نخی که یک رأس یا یک پیکسل را می‌کشد شباهت دارد. پردازنده گرافیکی چند نخی نخ‌های مستقلی ایجاد می‌کند که برنامه سایه زنی را برای کشیدن هر رأس و FragmentPixel اجرا می‌کند. علاوه بر رندر کردن گرافیک‌ها در زمان واقعی، برنامه نویسان از Cg برای محاسبه شبیه‌سازی‌های فیزیکی و محاسبات با اهداف همه‌منظوره روی واحد پردازنده گرافیکی استفاده می‌کنند. برنامه‌های محاسباتی با اهداف همه‌منظوره روی واحد پردازنده گرافیکی اولیه به این طریق کارایی بالایی کسب کردند اما به‌این‌علت که برنامه نویسان مجبور بودند محاسبات غیر گرافیکی را با برنامه‌های کاربردی گرافیکی مثل OpenGL بیان کنند نوشتن آن‌ها سخت بود [۹].

Geforce ۸۸۰۰ در سال ۲۰۰۲ معرفی شد و شامل اولین گرافیک‌های یکپارچه و معماری محاسبات پردازنده گرافیکی بود و علاوه بر اینکه با DX۱۰ و OpenGL قابل برنامه‌نویسی بود با زبان C با مدل محاسباتی کودا نیز قابل برنامه‌نویسی بود. هسته‌های پردازنده یکپارچه آن، رأس geometary[۱] و نخ‌های سایه زن پیکسل را برای برنامه‌های گرافیکی DX۱۰ اجرا می‌کردند و همچنین نخ‌های محاسباتی را برای برنامه‌های کودا C اجرا می‌کردند. سخت‌افزار چند نخی، Geforce ۸۸۰۰ را قادر ساخت تا ۱۲۲۱۱ نخ را هم‌زمان در ۱۲۱ هسته پردازنده اجرا کند. ان ویدیا معماری مقیاس‌پذیر را در خانواده‌ای از پردازنده‌های گرافیکی Geforce با تعداد هسته‌های پردازنده متفاوت برای هر بخش بازار منتشر کرد. Geforce ۸۸۰۰ اولین پردازنده گرافیکی بود که از پردازنده‌های نخی اسکالر به‌جای پردازنده‌های برداری استفاده می‌کرد و با زبان اسکالر استانداری مثل C انطباق داشت و به مدیریت

۱ eometry is a branch of mathematics concerned with questions of shape

رجیسترهای برداری نیازی نداشت و همچنین عملیات برداری را برنامه‌نویسی کرد. این تراشه دستوراتی برای پشتیبانی از C و زبان‌های همه‌منظوره دیگر مثل حساب صحیح، حساب ممیز شناور IEEE ۷۵۴ و دستورات دسترسی به حافظه (Load/Store) را به‌وسیله آدرس‌های بایتی اضافه کرد. همچنین سخت‌افزار و دستورات را برای پشتیبانی از محاسبات موازی، ارتباطات و هم‌زمانی (شامل هم‌زمانی آرایه‌های نخی، حافظه اشتراکی و سدهای سریع) فراهم کرد [۱۰].

در سال ۲۰۰۲ ان ویدیا در پاسخ به تقاضاها برای سیستم‌های محاسبات پردازنده گرافیکی؛ کارت‌های Tesla C۸۷۰، D۸۷۰ و S۸۷۰ و سیستم‌های محاسبات پردازنده گرافیکی، Rackmount شامل یک، دو و چهار پردازنده گرافیکی را معرفی کرد. پردازنده گرافیکی T۸ بر پایه پردازنده‌های گرافیکی Geforce ۸۸۰۰ بود و برای محاسبات موازی پیکربندی‌شده بود. دومین نسل سیستم‌های محاسباتی پردازنده گرافیکی، C۱۰۶۰ و Tesla S ۱۰۷۰ بود که در سال ۲۰۰۱ معرفی شد و از پردازنده‌های گرافیکی T۱۰ استفاده می‌کردند و بر پایه Geforce GTX ۲۸۰ بودند . T۱۰ شامل ۲۴۰ هسته پردازنده، نرخ ممیز شناور تک دقتی حداکثر یک تریلیون در هر ثانیه، معماری ممیز شناور دقت مضاعف ۲۴ بیتی IEEE ۷۵۴-۲۰۰۸ و GB ۴ حافظه DRAM است. امروزه سیستم‌های Tesla S۱۰۷۰ شامل هزاران پردازنده گرافیکی هستند که به‌طور گسترده در تولید و تحقیقات سیستم‌های محاسباتی با کارایی بالا استفاده می‌شوند [۱۱].

ان ویدیا نسل سوم معماری محاسبات پردازنده گرافیکی فرمی را در سال ۲۰۰۲ معرفی کرد. IEEE ۷۵۴-۲۰۰۸ فرمی را پیاده‌سازی کرد و به‌طور چشمگیری کارایی دقت مضاعف را افزایش داد. آن کد تصحیح خطا حفاظت از حافظه، آدرس‌دهی یکپارچه ۲۴ بیتی، سلسله مراتب حافظه کش و دستورات برای C ، C++ ، Fortran ، OpenCL و DirectCompute[۱] را برای محاسبات مقیاس وسیع پردازنده گرافیکی اضافه کرد.

اکوسیستم محاسبات پردازنده گرافیکی به‌وسیله توسعه بیش از ۱۱۰ میلیون پردازنده گرافیکی سازگار با کودا به‌سرعت توسعه پیدا کرد. محققان و توسعه‌دهندگان با شوق فراوان کودا و محاسبات پردازنده گرافیکی را برای محدوده وسیعی از برنامه‌های کاربردی پذیرفته‌اند. منتشر شدن صدها مقاله تکنیکی، پایان‌نامه‌های درسی برنامه‌نویسی موازی و آموزش برنامه‌نویسی کودا در بیش از ۳۰۰ دانشگاه گواه این مسئله است. حوزه کودا بیش از ۱۰۰۰ لینک را برای برنامه‌های کاربردی پردازنده گرافیکی، برنامه‌ها و مقالات تکنیکی ارائه کرده است. در کنفرانس تکنولوژی پردازنده گرافیکی ۲۱پوستر ، در سال ۲۰۰۲ تحقیقاتی منتشر شد [۱۲].

زبان‌های محاسبات پردازنده گرافیکی؛ کودا C ، کودا C++ ، Portland Group(PGGI) ،کودا Fortran ، Direct Compute و OpenCL است. بسته‌های ریاضی پردازنده گرافیکی شامل Math Works Mathlab

۱ Microsoft’s DirectCompute API for GPU Computing is supported on NVIDIA’s DX۱۰ and DX۱۱ class

، National Instrument Labview، Scicomp SciFinance وPy کودا است. ان ویدیا محیط توسعه چند جنبه‌ای موازی، اشکال‌زدایی و آنالیز کننده‌های مجتمع شده با مایکروسافت ویژوال استودیو را ایجاد کرده است.کتابخانه پردازنده گرافیکی شامل کتابخانه‌های بهره‌وری C++، جبر خطی متراکم، جبر خطی پراکنده، FFTs ، پردازش تصویر و ویدیو و داده‌های موازی اولیه است. صاحبان سیستم‌های کامپیوتری سیستم‌های با کمک پردازنده مجتمع شده باCPU+ پردازنده گرافیکی را در سرورهای rackmount و پیکربندی‌های کلاستر استفاده می‌کنند [۱۳].

۲-۳ معماری محاسباتی تسلا

شرکت ان ویدیا هم‌اکنون یکی از کارخانه‌های پیشتاز در تکنولوژی‌های گرافیکی است. ان ویدیا، پردازنده گرافیکی های با بیشترین زمینه‌های متقاضی محاسباتی مثل بازی، پردازش‌های گرافیکی حرفه‌ای و محاسبات با کارایی بالا را منتشر می‌کند. از نوامبر۲۰۰۲پردازنده‌های گرافیکی ان ویدیا بر پایه گرافیک‌های یکپارچه‌شده تسلا و معماری محاسبات ارائه شدند و از فوریه ۲۰۰۲ با محیط برنامه‌نویسی کودا تولید شدند تا برنامه‌نویسی خیلی هسته‌ای را ساده کنند. در راهنمای برنامه‌نویسی رسمی معماری تسلا به‌صورت مختصر به شیوه زیر توصیف‌شده است: یک مجموعه چندپردازنده‌های تک دستوری چند نخی با حافظه اشتراکی روی تراشه. چندپردازنده‌ها بر اساس یک اصطلاح دارای معماری SIMT هستند. SIMT اشاره به روشی دارد که یک پردازنده جریانی داده‌ها را سازمان‌دهی می‌کند.

۲-۳-۱ چندپردازنده‌های جریانی

معماری تسلا از آرایه‌ای از چندپردازنده‌ای جریانی چند نخی ساخته‌شده است. یک چندپردازنده شامل هشت پردازنده اسکالر، دو واحد تابعی مخصوص برای غیر جبری‌ها، یک واحد دستور چند نخی و یک حافظه روی تراشه‌ای است. یک‌چند پردازنده جریانی نخ‌های هم‌زمان در سخت‌افزار را با سربار زمانی صفر ساخته، مدیریت و اجرا می‌کند [۱۴].

۲-٤ حافظه‌های پردازنده گرافیکی

در یک پردازنده گرافیکی ما می‌توانیم دو نوع حافظه مجزا را محلی کنیم:

حافظه دستگاه و روی تراشه‌ای. حافظه‌های روی تراشه‌ای به‌صورت زیر هستند:

- یکسری رجیسترهای ۳۲ بیتی در هر پردازنده اسکالر
- یک حافظه چک نویس موازی (حافظه اشتراکی) در هرچند پردازنده جریانی. زمان دسترسی به حافظه

اشتراکی با کش L۱ روی پردازنده اصلی قابل‌مقایسه است.

ـ یک کش فقط خواندنی دائمی که توسط همه پردازنده‌های اسکالر به اشتراک گذاشته‌شده است تا خواندن از ناحیه دائمی در حافظه دستگاه را سرعت بخشد.

ـ یک کش بافت فقط خواندنی که توسط همه پردازنده‌های اسکالر به اشتراک گذاشته‌شده است تا خواندن از ناحیه بافت در حافظه دستگاه افزایش یابد.

ـ حافظه دستگاه یک حافظه DRAM[۱] با سرعت و تأخیر زیاد است و ابعاد آن از حافظه روی تراشه بیشتر است (به‌طور مثال صدها برابر کندتر و میلیون‌ها بار بزرگ‌تر هستند.)

حافظه دستگاه به چهار بخش تقسیم‌شده است:

۱- ناحیه غیر کش سراسر قابل‌خواندن و نوشتن

۲- ناحیه غیر کش محلی قابل‌خواندن و نوشتن

۳- ناحیه کش ثابت فقط خواندنی

۴- ناحیه کش بافت فقط خواندنی

در اینجا در مورد یکسری اصطلاحات حافظه صحبت می‌کنیم. حافظه سراسر و دستگاه آشکارا هم‌معنی نیستند. سراسری به الگوی دسترسی دلالت دارد که فضای خاصی است و قسمتی از حافظه دستگاه پردازنده گرافیکی است. به‌طور مشابه حافظه محلی و اشتراکی مفهوم‌های یکسانی نیستند. حافظه محلی قسمتی از حافظه دستگاه است که کند است درحالی‌که حافظه اشتراکی روی تراشه است و سریع است. حافظه محلی به‌وسیله کامپایلر استفاده می‌شود تا هر چیزی را که توسعه‌دهنده آن را برای یک نخ محلی در نظر می‌گیرد اما به بعضی دلایل در رجیستر پردازنده اسکالر؛ جایی که نخ اجرا می‌شود؛ جا نمی‌شود را حفظ کند (برای مثال، ساختارهای بزرگ که رجیستری زیادی را مصرف می‌کنند).

برخلاف مدل حافظه بیشتر پردازنده‌های اصلی، هیچ مکانیزمی برای خودکار کردن کش بین RAM پردازنده گرافیکی و کش اشتراکی وجود ندارد. پردازنده‌های گرافیکی در طول اجرا نمی‌توانند مستقیماً به حافظه میزبان دسترسی پیدا کنند. در عوض، داده به‌طور واضح از قبل بین حافظه دستگاه و حافظه میزبان انتقال می‌یابد و اجرا پردازنده گرافیکی ادامه می‌یابد. ازآنجایی‌که مدیریت دستی حافظه برای اجرا لازم است (مکانیزم صفحه‌بندی وجود ندارد) و به‌این‌علت که در زمان اجرا پردازنده‌های گرافیکی نمی‌توانند داده را بین پردازنده‌های گرافیکی و پردازنده‌های اصلی انتقال دهند برنامه نویسان نیاز دارند که برنامه‌های کاربردی خود را تغییر دهند به‌طوری‌که همه داده‌های مربوط در زمان نیاز درون پردازنده گرافیکی قرارگرفته باشند. مجموعه

[۱] Dynamic random-access memory

داده‌هایی که خیلی بزرگ هستند و در یک پردازنده گرافیکی جا نمی‌شوند به چندین انتقال بین پردازنده گرافیکی و پردازنده اصلی نیاز دارند و باعث به وجود آمدن حباب در زمان اجرا می‌شوند. با محاسبات موازی سنتی، استفاده از multi- پردازنده‌های گرافیکی منابع اضافی را فراهم کرد که به‌طور بالقوه به فراخوانی پردازنده گرافیکی کمتری نیاز است و اجازه می‌دهد که الگوریتم‌ها ساده‌تر شوند [۱۴].

۲-٤-۱ نمونه SIMT[1]

SIMT یک نمونه جدید است که برای مدیریت صحیح مقادیر بزرگ نخ‌های اجرایی روی پردازنده‌های گرافیکی بر پایه تسلا معرفی‌شده است. یک تفاوت کلیدی بین SIMT و SIMD[2] در این است که SIMT بردار پهن ندارد و یک تک دستور استخراج‌شده و به‌وسیله عنصرهای پردازشی چندگانه اجرا می‌شود تا در هر بار بهره‌برداری کامل از هسته‌ها پشتیبانی کند. در مقابل، معماری SIMD وقتی‌که اندازه ورودی کوچک‌تر از ابعاد بردار است استفاده می‌شود. SIMT تا حدی شبیه معماری SIMD است. اگرچه این دو هیچ‌وقت نباید باهم اشتباه گرفته شوند اما هر دو آن‌ها اصول پردازش هم‌زمان المان‌های داده چندگانه با یک دستور یکسان را به اشتراک می‌گذارند .یک دستگاه SIMT قادر است که کد موازی سطح نخ را زمانی که نخ‌های مستقل انشعاب می‌یابند، پردازش کند درحالی‌که یک دستگاه SIMD به مسیرهای اجرا یکسانی نیاز دارد (نمی‌تواند چندین دستور انشعاب را اجرا کند) [۱۴].

۲-۵ معماری مقیاس‌پذیر کودا

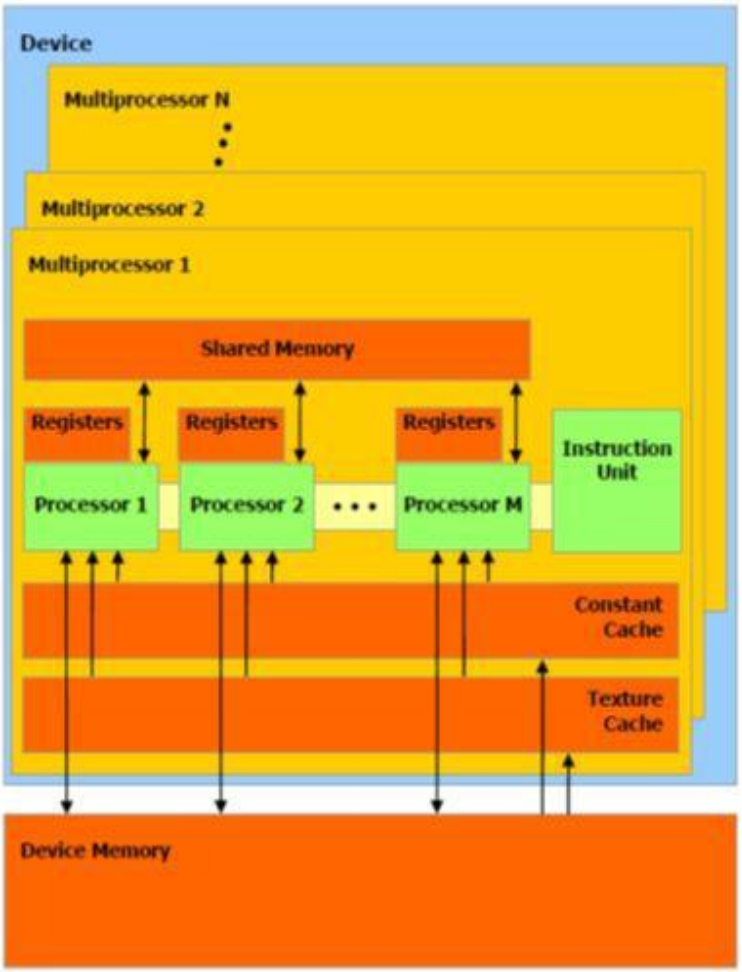
به‌این‌علت که بیشتر زبان‌ها برای یک نخ ترتیبی طراحی‌شده بودند، کودا این مدل را حفظ می‌کند و آن را با مجموعه حداقلی از تجردها برای تشریح موازی‌سازی گسترش می‌دهد. این به برنامه نویسان اجازه می‌دهد که روی مسائل مهم موازی‌سازی تمرکز کنند؛ اینکه چگونه الگوریتم‌های موازی کارآمد را با استفاده از یک‌زبان آشنا طراحی کنند. با طراحی آن، کودا توسعه برنامه‌های موازی مقیاس‌پذیر را که می‌توانند در میان چندین هزار نخ هم‌زمان و هزاران پردازنده اجرا شوند فراهم می‌کند. یک برنامه کودا کامپایل شده روی هراندازه‌ای از پردازنده گرافیکی می‌تواند اجرا شود؛ زمانی که پردازنده گرافیکی از پردازنده‌ها و نخ‌های بیشتری تشکیل‌شده باشد می‌تواند به‌طور خودکار از موازی‌سازی بیشتری استفاده کند. یک برنامه کودا در یک برنامه میزبان درست‌شده است، که

[1] ingle instruction, multiple thread
[2] Single instruction, multiple data

شامل یک یا چندین نخ ترتیبی است که روی یک پردازنده اصلی اجرا می‌شود و یک یا چندین هسته موازی برای اجرا روی پردازنده گرافیکی محاسباتی موازی است [۱۵].

۲-٦ معماری دستگاه کودا

معماری دستگاه کودا ان ویدیا از مجموعه‌ای از پراسسورهای چندهسته‌ای و چند نخی که چندپردازنده‌ای جریانی نامیده می‌شوند تشکیل‌شده است. شکل (۲-۱) معماری سخت‌افزار دستگاه کودا را نشان می‌دهد. یک دستگاه از تعداد بی‌شماری چندپردازنده‌ای جریانی تشکیل‌شده است. این چندپردازنده‌ای جریانی شامل تعدادی پردازنده یا هسته (تعریف‌ها متفاوت است) هستند. هرچند پردازنده یک واحد دستور مشترک برای همه هسته‌های خودش دارد.



شکل (۲-۱) معماری دستگاه کودا [۱۵]

۲-۶-۱ نخ‌ها، بلاک‌ها و گریدها: تطبیق الگوریتم‌ها با مدل کودا

کودا از مفهوم سویچ‌های تک چرخه‌ای استفاده می‌کند تا تأخیر مسیر داده و دسترسی به حافظه را پنهان کند. وقتی از پردازنده‌های گرافیکی سری ان ویدیا G۸۰ استفاده می‌کنیم، نخ‌ها میان ۱۲ چندپردازنده مدیریت می‌شوند، که هرکدام شامل ۱ هسته SIMD هستند. متد مدیریت اجرای کودا تقسیم رودهایی از نخ‌ها در میان بلاک‌ها است. گریدها از مجموعه‌ای از بلاک‌ها تشکیل‌شده‌اند. نخ‌ها می‌توانند مکان خودشان را در یک بلاک و مکان بلاک درون گرید را با المان‌های داده ذاتی که به‌وسیله کودا مقداردهی اولیه شده‌اند مشخص کنند. نخ‌های درون یک بلاک می‌توانند باهم از طریق توابع سدی که توسط کودا فراهم‌شده است هم‌زمان شوند؛ اما این امکان وجود ندارد که نخ‌های درون بلاک‌های مختلف مستقیماً باهم هم‌زمان شوند یا باهم ارتباط برقرار کنند. نخ‌های بلاک‌های مختلف متعلق به یک گرید می‌توانند از طریق عملیات اتمیک، در فضای حافظه سراسری که با بقیه نخ‌ها به اشتراک گذاشته‌اند هماهنگ شوند. گریدهای که به‌صورت ترتیبی به هسته وابسته هستند می‌توانند از طریق مانع‌های سراسری هم‌زمان و از طریق حافظه اشتراکی سراسری هماهنگ شوند. برنامه‌های کاربردی که به‌خوبی به این مدل نگاشت شوند به علت درجه بالا موازی‌سازی سطح داده توانایی موفقیت بالایی با multi- پردازنده‌های گرافیکی دارند. زمانی که برنامه‌های کاربردی توانایی تقسیم به بلاک‌های مستقل را داشته باشند این مدل اجازه می‌دهد که اجرا روی چندین پردازنده گرافیکی نیز به‌خوبی انجام شود البته این حالت همیشه امکان‌پذیر نیست و برای اینکه این امر امکان‌پذیر باشد یک بلاک نباید روی بلاک دیگر تأثیر گذارد [۱۵].

۲-۶-۲ نخ‌ها

کودا از تعدادی از عملیات اتمیک پشتیبانی نمی‌کند اما این عملیات تنها روی مجموعه‌ای از پردازنده‌های گرافیکی در دسترس هستند. استفاده‌ی متناوب از عملیات اتمیک موازی‌سازی که پردازنده گرافیکی فراهم می‌کند را محدود می‌کند. این عملیات اتمیک مکانیزم های هم‌زمانی بین تردهای درون بلاک‌های متفاوت است.

برنامه‌نویسی کودا از مدل SPMD به‌عنوان پایه هم‌زمانی استفاده می‌کند، آن‌ها به هر جریان داده به‌عنوان یک نخ اشاره می‌کنند و نمونه خودشان را SPMT می‌نامند. این به این معناست که هر نخ کودا مسئول یک جریان مستقل از کنترل برنامه است. ان ویدیا تصمیم می‌گیرد که از مدل برنامه‌نویسی SPMT استفاده کند تا برنامه نویسان موازی که تجربه نوشتن برنامه‌های چند نخی را دارند احساس راحتی با کودا داشته باشند. اگرچه ازنظر تکنیکی نخ‌های کودا از هم مستقل هستند، در واقعیت کارایی هر برنامه کودا به‌طور شدیدی به گروه‌های نخی که دستورات یکسان را در یک مرحله اجرا می‌کنند وابسته است [۱۵].

برخلاف مدل SPMT[1] کودا، سخت‌افزار پردازنده گرافیکی ان ویدیا با استفاده از واحدهای چند پردازشی SPMD[2]ساخته‌شده است. نخ‌ها به گروه‌های ۳۲بیتی که بسته نامیده می‌شوند گروه‌بندی می‌شوند و به SP ها نگاشت می‌شوند. یک بسته واحد پایه‌ای زمان‌بندی است. همه‌ی ۳۲بیت یک بسته هرچند داده آن‌ها متفاوت است باید دستور یکسانی را اجرا کنند. یک نخ به یک SP نگاشت می‌شود. بسته‌ها به‌وسیله MIU ها ساخته، مدیریت و زمان‌بندی می‌شوند. تعداد نخ‌ها در هر بسته معمولاً دو برابر تعداد SP ها است. تعداد نخ‌هایی که به یک بسته تعلق دارند با یک آدرس برنامه یکسان شروع می‌شوند و برای انشعاب و اجرا به‌صورت مستقل عمل می‌کنند. چندپردازنده‌ها هشت واحد تابعی (که هسته نامیده می‌شوند) دارند که بیشتر عملیات را در چهار سیکل تمام می‌کنند. در سیکل اول، هشت نخ اول (نخ‌های ۲-۰) داده خود را وارد خط لوله می‌کنند. این روند به‌وسیله یک سویچ متن که هشت نخ بعدی (نخ‌های ۱۵-۱) را فعال می‌کند ادامه می‌یابد. در سیکل دوم، این نخ‌ها (نخ‌های ۱۵-۱) داده‌ی خود را وارد خط لوله می‌کنند. گروه‌های سوم و چهارم نیز به همین ترتیب دنبال می‌شوند. در سیکل پنجم، نخ‌های اول نتایج خود را به دست آورده‌اند و آماده اجرا دستور بعدی هستند. اگر نخ‌های کافی برای پر کردن این ۳۲ جای خالی دربسته وجود نداشته باشد این سیکل‌ها به هدر می‌روند.

MIU یک بسته را که آماده اجرا است انتخاب می‌کند و دستور بعدی را به نخ‌های فعال هر بسته می‌فرستد .یک بسته در یک‌زمان یک دستور مشترک را اجرا می‌کند، بنابراین کارایی کامل زمانی به دست می‌آید که همه-ی نخ‌های یک بسته روی مسیر اجرایی توافق داشته باشند. اگرچه، نخ‌ها آزاد هستند که به‌صورت متفاوت اجرا شوند، اما وقتی این واقعه رخ دهد این ریسک وجود دارد که کارایی به‌طورجدی آسیب ببیند. علت آن این است که ناسازگاری روی جریان کنترل، زمان‌بند نخ را مجبور می‌کند که اجرای آن‌ها را سریالی کند.

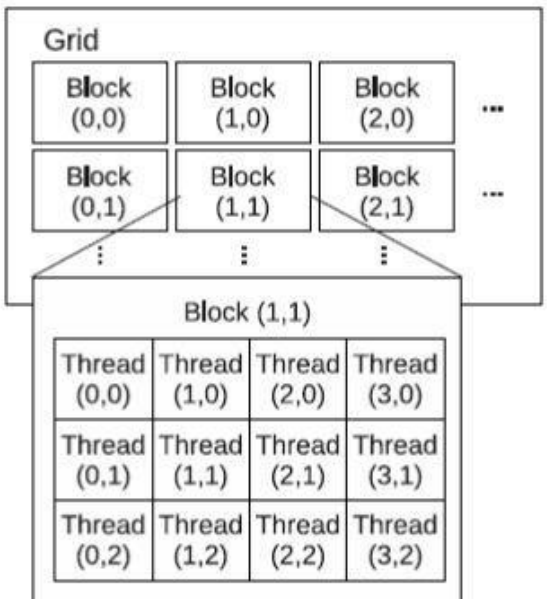
مسلماً، زمانی که بسته به‌صورت مستقل اجرا می‌شوند، سریالی شدن مسئله‌ای است که ممکن است فقط برای نخ‌های یک بسته ۳۰ یکسان رخ دهد. واگرایی انشعاب‌ها تنها علتی نیست که ممکن است منجر به سریالی شدن نخ‌ها شود. اگر یک دستور به‌وسیله یک بسته اجراشده باشد، بدون توجه به اتمیک بودن آن نوشتن در مکان یکسانی در حافظه اشتراکی برای یک یا بیشتر نخ‌های درون یک بسته به‌صورت سریالی انجام می‌شود. به این اثر جانبی تناقض بانک گفته می‌شود. گاهی اوقات نخ‌های درون یک بسته به یک جمله شرطی می‌رسند و مسیر اجرایی متفاوتی می‌گیرند. در این حالت، همان‌طور که بیان شد جریان دستورات از هم دور می‌شود و بعضی نخ‌ها نیاز دارند که دستورات درون شرط‌ها را اجرا کنند درصورتی‌که بقیه نخ‌ها به اجرا این دستورات نیازی ندارند .برای مقابله با این موضوع، دستورات درون قسمت شرطی روی چندپردازنده‌ها اجرا می‌شوند، اما نخ‌هایی که نباید پنهان شوند تا زمانی که جریان کنترل دوباره همگرا شود بیکار می‌مانند.

[1] single-program-multiple-exeuction-time
[2] single program, multiple data

۲-۶-۳ بلاک‌ها و گریدها

در کودا، بلاک واحد قابل زمان‌بندی است که می‌تواند به یک چندپردازنده تخصیص داده شود. یک بلاک شامل تعداد صحیحی بسته است و در هر زمان حداکثر به یک چندپردازنده تخصیص داده می‌شود. اگرچه بسته‌ها برحسب توان عملیاتی بااهمیت هستند، آن‌ها به‌طور مؤثر برای مدل برنامه‌نویسی شفاف هستند. در عوض، کودا نخ‌ها را به ساختارهای(بلاک‌های)یک، دو و سه‌بعدی مدل می‌کند که درواقع واحد قابل زمان‌بندی برای اجرا یک چندپردازنده‌ها هستند. اگر همه‌ی نخ‌های یک بلاک منتظر یک خواندن یا نوشتن از حافظه با تأخیر زیاد باشند، سپس کودا ممکن است بلاک دیگری را زمان‌بندی کند تا روی چندپردازنده یکسانی اجرا شود. به‌هرحال، کودا فایل رجیستر یا کش اشتراکی را در زمان تعویض یک بلاک مبادله نمی‌کند بنابراین اگر یک بلاک جدید درزمانی که بلاک دیگر منتظر است اجرا شود آن بلاک می‌تواند با منابعی که هنوز در دسترس هستند کار کند.

زمانی که یک برنامه کودا که کرنل نامیده می‌شود روی پردازنده گرافیکی اجرا می‌شود هر نخ مشخصه‌ی مشخصی دارد که به‌طور پویا تعیین می‌شود. این مقادیر، هماهنگ‌کننده‌های بلاک خودشان در گرید و هماهنگ‌کننده‌های نخ خودشان درون بلاک هستند. یک تمرین عمومی برای نگاشت نخ‌ها به مجموعه‌های مسئله این است که یک نخ داشته باشیم که مسئول هر المان درون داده خروجی باشد. برای انجام این کار، ابعاد بلاک‌ها و نخ‌ها معمولاً ابعاد مجموعه داده خروجی را نشان می‌دهند .در شکل (۲-۲) مفهوم نخ، بلاک و گرید به تصویر کشیده شده است [۱۵].



شکل (۲-۲) گریدها، بلاک‌ها و نخ‌ها [۱۵]

وقتی یک تابع کودا که به‌عنوان هسته شناخته می‌شود از برنامه اصلی فراخوانی می‌شود، با صدها نخ هم‌زمان شروع می‌شود. هرچند پردازنده به‌طور مستقل نخ‌ها و بلاک نخ‌ها را مدیریت می‌کند. هرچند پردازنده هم‌زمان چندین بلاک نخ را به‌خوبی بسته‌های یک بلاک پردازش کند. هرچند پردازنده به‌وسیله پردازش SIMT محدودشده است اما چندپردازنده‌های مختلف یک دستگاه کودا ممکن است در یک کرنل مسیر اجرای متفاوتی داشته باشند.

۲-۷ کاربردهای کودا

۲-۷-۱ زمینه : فیلم و سرگرمی

نام نرم‌افزار: Weta Digital PantaRay

توسعه‌دهنده: Weta Digital

در حوزه‌ی فیلم و سرگرمی، شرکت Weta Digital[۱] یکی از نخستین شرکت‌هایی است که از قدرت پردازنده گرافیکی برای رندر تصاویر استفاده کرده است. مهم‌ترین دلیل حرکت این شرکت به این سمت، وجود محیط‌های بسیار پیچیده و بسیار زیاد کامپیوتری در فیلم آواتار بود که رندر آن‌ها یکی از مهم‌ترین چالش‌های شرکت به شمار می‌آمد. به همین دلیل، با همکاری یکی از متخصصان شرکت ان ویدیا و انجام تحقیقات روی پلتفرم اختصاصی Weta Digital VFX Pipeline ، نرم‌افزار برای استفاده از فناوری کودا و پردازش موازی سنگین روی پردازنده گرافیکی بهینه‌سازی شد و یک موتور پیش‌پردازش برای انجام فرایند ردیابی پرتو به آن اضافه شد. با استفاده از این موتور پردازش جدید، سرعت فرایند ردیابی پرتو در مقابل حالت‌های سنتی استفاده از پردازنده اصلی چیزی حدود ۲۵ برابر افزایش‌یافته است. همچنین، در مقایسه با فناوری‌های سابق، سرعت این فرایند چیزی حدود صد برابر افزایش‌یافته است که به میزان بسیار زیادی در کاهش زمان و هزینه تولید تأثیر داشته است [۱۶].

۲-۷-۲ زمینه :پردازش‌های تصویری

نام نرم‌افزار : مختلف

توسعه‌دهنده : مختلف

از کارایی‌های فناوری کودا درزمینه‌ی فناوری‌های بصری کامپیوتری بیش از هر زمینه‌ی دیگری

[۱] **Weta Digital** is a digital visual effects company based in Wellington

استفاده‌شده است. نرم‌افزارهای مختلفی در این زمینه تولیدشده‌اند و سرعت محاسبات در این زمینه به میزان بسیار زیادی افزایش‌یافته است. در ادامه به چند نرم‌افزار که برای استفاده از امکانات کودا در زمینه‌های مختلف توسعه داده‌شده‌اند، می‌پردازیم.

FurryBall ۱-۲-۷-۲

نرم‌افزار FurryBall نخستین رندرکننده بی‌درنگ مبتنی بر پردازنده گرافیکی است که به‌طور مستقیم در مایا پیاده‌سازی شده است. با استفاده از این نرم‌افزار، می‌توان رندر مدل‌های سه‌بعدی را به‌صورت بی‌درنگ و بدون تأخیر به همراه بافت، سایه‌ها، انعکاس‌ها، باند تداخل رنگ و عمق میدان دید در پنجره اصلی مایا دید و ویرایش کرد. همچنین، می‌توان با استفاده از قدرت پردازنده گرافیکی خروجی‌هایی بدون تأخیرهای بلندمدت و با سرعتی معادل سی تا سیصد برابر حالت رندر با پردازنده اصلی ایجاد کرد.

Ruins ۲-۲-۷-۲

این نرم‌افزار یک پلاگین افکت خرد شدن در مایاست که با دقت و سرعت‌بالا، خرد شدن اجسام در مایا را شبیه‌سازی می‌کند. در نرم‌افزار مایا، پیاده‌سازی افکت خرد شدن اجسام کار بسیار مشکل و کندی است که با استفاده از قدرت پردازنده گرافیکی و کودا به همراه PhysX ، ایجاد این افکت با سرعتی بالاتر، واقعی‌تر و با سهولت بیشتر انجام می‌پذیرد.

Badaboom ۳-۲-۷-۲

یکی از بهترین کاربردهایی که برای استفاده از فناوری کودا مطرح‌شده است، تبدیل فرمت‌های ویدیویی با سرعتی بسیار بالاتر از گذشته است. نرم‌افزار Badaboom یک مبدل فرمت تصویری ساده است که با استفاده از قدرت پردازشی پردازنده گرافیکی و فناوری کودا می‌تواند عملیات تبدیل پَروَنجاهای ویدیویی را با سرعتی معادل بیست برابر گذشته(حالت استفاده از) پردازنده اصلی انجام دهد.

vReveal ٤-۲-۷-۲

نرم‌افزار vReveal یک ابزار رایگان برای اصلاح تصاویر ویدیویی ضبط‌شده با دستگاه‌های موبایل است که روی کارت‌های گرافیکی ان ویدیا با استفاده از کودا، پنج برابر سریع‌تر عمل کرده و تصاویری پایدارتر، روشن‌تر و حرفه‌ای‌تر به ارمغان میاورد. در این نرم‌افزار، سرعت اعمال تغییرات به ویدیوهای باکیفیت معمولی با سرعتی باورنکردنی و به‌صورت بی‌درنگ انجام می‌پذیرد و با انتخاب یک گزینه، کاربر می‌تواند فایل اصلاح‌شده

را در کنار فایل اصلی به‌صورت هم‌زمان مشاهده کرده و میزان تغییرات اعمال‌شده را موردبررسی قرار دهد [۱۶].

۲-۸ فیلتر ان.ال.ام[۱]

فرض کنید Ω حجم سه‌بعدی یک تصویر و X_i واکسل‌های مربوط است. پس X_i€Ω است. پس ما داریم $U_i=(U_i^1 \ldots\ldots\ldots U_i^2)$ شدت چند مقوله‌ای بررسی MR در واکسل Ki، با این فرض که مقوله‌های C ارائه می‌شود. به‌صورت آنالوگی ما شدت واکسل X_i را بعد از حذف نویز از طریق U_i داریم. در یک فرآیند سه‌بعدی فول فیلتر NLM [۲] شدت بازیافتی در واکسل X_i ژنریک یک متوسط اندازه‌گیری شده که (kernelconvolution نامیده می‌شود) شدت‌هایی در واکسل‌ها است که حجم جستجوی سه‌بعدی V_i Ω مربوط می‌شود که در واکسل X_i متمرکزشده است:

$$\tilde{u}_i^c = \sum_{x_j \in V_i} w^c(x_i, y_j) u_j^c, c = ۱, \ldots, C, \qquad \text{فرمول (۱-۲)}$$

در این معادله $w^c(x_i\ x_j)$ بار اختصاص داده‌شده به U^c_j در حذف نویز از واکسل X_i است. اگر بخواهیم به‌طور دقیق‌تر توضیح دهیم بار یک برآورد و تخمین از شباهت بین شدت‌های دو قسمت مجاور سه‌بعدی D_iΩ و D_j Ω است که به ترتیب در واکسل‌های X_i و X_j متمرکزشده است:

$$w^c(x_i, x_j) \in [۰,۱],\ \sum_{x_j \in V_i} w^c(x_i, x_j) = ۱, \quad c = ۱, \ldots, C. \qquad \text{فرمول(۲-۲)}$$

تعریف فیلتر NLM کلاسیک هیچ فرضی در مورد حجم جستجو ایجاد نمی‌کند فقط نشان می‌دهد که هر واکسل می‌تواند به واکسل‌های دیگر مرتبط شود. اما در ارتباط با دلایل محاسباتی معمولاً این فرض وجود دارد که قسمت مجاور U_i و D_i به‌صورت مکعب‌هایی با اندازه $(۲d_x+۱)(۲d_y+۱)(۲d_z+۱)$ و $(۲v_x+۱)(۲v_y+۱)(۲v_z+۱)$ یا $(V_X، V_y، V_z)$ و $(d_x، d_y، d_z)$ که به ترتیب شعاع‌های پنجره V_i و D_i است که در ابتدا هر جهت فضایی x،y،z می‌باشند.

ما مطرح می‌کنیم که در ساختار دوبعدی NLM متوسط بار بالای واکسل‌هایی پدید می‌آید که به یک قسمت یکسان مربوط می‌شوند، بنابراین حذف نویز به‌صورت قسمت به قسمت انجام می‌شود و پردازش هر قسمت به اطلاعات روی قسمت دیگر مربوط نمی‌شود. اضافه بر این تشکیل یک مقوله، پیچیدگی (convolution) به‌صورت

[۱] NLM

قسمت به قسمت محاسبه می‌شود (یعنی T_1،T_2،PD،FLAIR و غیره) و نتیجه تحت تأثیر مقوله‌های باقی‌مانده قرار نمی‌گیرد.

شباهت بین دو واکسل X_i و X_j با شاخصی که بر اساس شدت چند مقوله‌ای در قسمت‌های D_i و D_j است بیان می‌شود. با توجه به [۹] فاصله (۲) مقیاس قابل‌قبول برای تخمین زدن شباهت پنجره‌های تصویر در مسیر بافت است. بنابراین، می‌توان فاصله (Euclidean) اقلیه سی با احتمال اندازه‌گیری گاوسی

$\|\square(D_i)-\square(D_j)\|_2^2$ بین مسیرهای $U(D_i)$ و $U(D_j)$را برآورد کرد. اندازه‌گیری‌های مربوط به فاصله درجه دوم (quadratic) در [۲] ارائه‌شده و در حالت معمول چند مقوله‌ی به‌صورت [۲۰] ارائه‌شده.

$$w(x_i,x_j)=\frac{1}{z_i}\exp\left(-\frac{1}{C}\sum_{c=1}^{C}\frac{\|u^c(D_i)-u^c(D_j)\|_2^2}{(h^c)^2}\right)$$ فرمول(۲-۳)

در این معادله Z_i ثابت نرمال‌سازی است تا اطمینان را به دست دهد که h^c.....h^1 و پارامترهای فیلترینگ چند مقوله‌ای می‌باشند که مسیریابی تابع‌نمایی را کنترل می‌کنند. در حقیقت برای هر واکسل در حجم، فاصله بین شدت‌های مجاور(D) U^c و $U^c(D_j)$ برای تمام واکسل های nj موجود در Vi و برای هر مقوله c باید محاسبه شود. با نشان دادن اندازه تصویر سه‌بعدی با N، پیچیدگی فیلتر ترتیب $((2dz+1)(2dy+1)(2vx+1)(2vz+1)$ $O(NC(2vz+1)$ است. برای تصویر ایزوتروپی می‌توان این فرض را داشت که:

$V_n=V_{y=}V_z=V$ و $dn=dy=dz=d$

۲-۸-۱ نویز ریشن[1]

همان‌طور که در قسمت مقدمه بحث گردید، نویز اکثر تصاویر MR پیرو تابع شدت Rician است، بنابراین یک متدلوژی بر پایه گاوسی متوسط بارگذاری شده در (۱) سو گرایی دارد که به خاطر عدم تقارن در توزیع Rician است. من جان و همکاران (۱۹) و ویست – دایسل و همکاران (۲۵) ورژن های هماهنگ‌سازی شده فیلتر NLM را مطرح کردند که می‌توانند این سو گرایی (bia) ناشی از تقریب گذاری گاوسی را حذف نمایند. از این پژوهش ما طرح تصحیح سازی با (۲۵) را مورداستفاده قرار می‌دهیم.

این بر پایه لحظه درجه دوم (seccnd order moment) توزیع Rician است.

$$E[X^2]=\mu^2+2\sigma^2$$ فرمول(۲-۴)

در این معادله $\partial 2$ واریانس نویز گاوسی در فضای k قبل از تقصیر شکل فوریه (Founer tlansform) است. در

[1] Rician

ورژن blockwise شدت تصحیح‌شده Rician، c.....۱=c و Uc از معادله زیر به دست می‌آید.

$$\tilde{u}_i^c = \sqrt{\max(\sum_{x_j \in v_i} w^c(x_i, x_j)\,(\tilde{u}_j^c)^{۲}, ۰)}\,, \quad c = ۱, \ldots, C$$ فرمول(۲-۵)

با استفاده نادرست ما شدت U^c برآورد سازی شده مهم با فرض نویز گاوسی و هم تصحیح سازی Rician را نشان می‌دهیم.

فیلتر NLM چندگزینه‌ای در مالتی جی پی یو[۱] ما الگوریتم‌های NLM مالتی-جی پی یو را با استفاده از ابزار کودا برای جی پی یوهای ان ویدیا ساخته‌ایم. کودا یک API شکل کاربرد معمول را می‌یابد و کنترل روی اینکه چگونه اقدام در سخت‌افزار پردازنده گرافیکی محاسبه شود، ارائه می‌دهد.

در کودا سیستم از یک هاست (سی پی یو) و یک یا چند دستگاه دیگر تشکیل می‌شود. که خیلی زیاد با پردازشگرها موازی می‌باشند. این هاست می‌تواند داده‌های کاربردی را بین هاست و حافظه دستگاه انتقال دهد، و تعداد متغیری از عملیات (که کرنل نامیده می‌شوند) برای اجرا روی دستگاه فعال و تحریک سازد. کرنل‌ها ویژگی مهم موازی‌سازی داده‌ای را نشان می‌دهند، باعث می‌شوند. داده‌ها به‌طور هم‌زمان توسط تعداد زیادی از threqdsها اجرا می‌شوند.

Threds‌هایی که همان کرنل را اجرا می‌کنند در چندین سطح شبکه‌ای از بخش‌های نخ سازمان‌دهی می‌شوند. هر بخش نخ تا ۱۰۲۴ نخ دارد که ویژگی‌های دستگاه بستگی دارد. نمونه‌های درون دستگاهی (Bult-in) مانند (thread ldx،threadldy،Threadldz) و (block ldx،block ldy،blockldz) شبکه نخ و پیکربندی‌های شاخص بلوک به هر نخ را ارائه می‌دهند. و آن‌ها برای تقسیم فعالیت بین نخ‌ها مورداستفاده قرار داده می‌شوند. نخ‌های در یک بلوک نخ یکسان با همان مالتی پردازشگر (SM) اجرا می‌شوند و می‌توانند در یکدیگر هماهنگ‌سازی شوند و همان حافظه موجود را مورداستفاده قرار دهند. بخش‌های متوالی از ۳۲ نخ به‌گونه‌ای تعریف می‌شوند که قسمتی از یک بسته‌بندی باشند که در آن تمام نsها در یک موازی دستور یکسان را اجرا می‌کنند. متأسفانه وقتی نخ‌ها در یک wrap مسیرهای اجرا متفاوت را دارند، گاهی اوقات subpression بعضی نخsها لازم می‌شود تا اجزاء کامل به دست آید باعث نقص اجرایی می‌شود.

کودا انواع متفاوت ممووری (حافظه) را پشتیبانی می‌کند، حافظه اصلی[۲] بزرگ‌ترین حافظه است. اما تأخیر[۳] بالایی دارد. اساساً آن برای ذخیره‌سازی داده‌های خروجی و ورودی مورداستفاده قرار داده می‌شود معمولاً آن عملکرد کرنل‌های کودا را مجدد می‌سازد اگر حافظه‌های دیگر مورداستفاده قرار نگیرد. هم حافظه ثابت و هم حافظه

[۱] Multi-GPU
[۱] Global memory
[۲] latency

مشترک روی حافظه‌های آن چیپی[۱] می‌باشند.

حافظه ثابت یک حافظه کوچک خواندنی[۲] است، از تأخیر پائین و دسترسی باند بالا را پشتیبانی می‌کند وقتی تمام نخ‌ها به‌طور هم‌زمان به یک مکان دسترسی دارند حافظه مشترک را می‌توان به قسمت‌های نخ اختصاص داد و درروش کاملاً موازی با سرعت‌بالا قابل‌دسترسی می‌باشند وقتی تمام tlveadsها در یک بخش نخ (Thread block) می‌توانند حافظه مشترک خود را بخوانند و بنویسند، روش بسیار مؤثری برای نخ‌های وجود دارد تا داده‌های ورودی خود و نتایج آن را به اشتراک‌گذارند.

برای به دست آوردن مزیت عمده از مدارهای پردازنده گرافیکی موجود ما یک ورژن جدید از مالتی-چی پی یو[۳] را ساخته‌ایم. همان‌طور که روی مقدار از داده‌هایی که هر پردازنده گرافیکی می‌تواند پردازش کند محدودیت وجود دارد، قسمت‌هایی از داده‌های اصلی در هر درخواست کرنل با توجه به حافظه موجود پردازش می‌شوند. اضافه بر این، برای اطمینان از اینکه هر پیکسل را بتوان درروش یکسان مورد ارزیابی قرار داد، ما یک کپی اولیه بزرگ‌تر از داده‌های تصویر داریم که در آن حاشیه‌های اضافی وجود دارد. ما این فضای اضافی را با انعکاس تصویر حاشیه‌ای پر می‌کنیم. برای اینکه باعث شدیم تمام عملیات در تمام پردازنده گرافیکی های موجود اجرا شود، ما به این نیاز داریم تا حجم کل را به قسمت‌هایی متناسب با تعداد پردازنده‌های گرافیکی موجود در دستگاه تقسیم‌بندی نماییم.

برای تضمین دقت الگوریتم‌ها، ما داده‌های فرعی بزرگ‌تر را (Subdata) هماهنگ‌سازی نمی‌کنیم به‌اندازه‌ای که پارتیشن داده‌ای و بعضی حاشیه‌های اضافی را داشته باشند که با داده‌های ورودی از پارتیشن‌های مجاور پر می‌شوند. برای حجم Ω مشخص ۳D full داده‌هایی $N=(N_Y،N_Y،N_x)$ وقتی مقدار داده‌ها بیش از بیش از mg،.....G،.....g=۱ حافظه اصلی هر پردازشگر باشد، شعاع‌های s،dوv را در نظر بگیرند و حافظه اضافی لازم برای تخصیص ساختارهای کرنل آنی را داشته باشند، و بتوانند نتایج کرنل را جمع‌آوری کنند و از پردازنده گرافیکی به قسمت پردازنده گرافیکی کپی نمایند. با مشخص کردن حجم کلی داده‌ها برای پردازنده‌های گرافیکی با M تعداد کلی شکافت‌ها (N+€)/M است به‌طوری‌که € داده‌های اضافی است که ردیف‌های آن‌ها با حاشیه‌های گسترش‌یافته از هر تصویر شکافته شده پر می‌شود.

اضافه بر این، داده‌های مربوط به هر شکافت به زیر ساختارهای G تقسیم می‌شوند. بنابراین $\Omega^{g\,c}_{ij}$

داده‌های فرعی (subdata) است که متعلق به قسمت i در شکافتj مقوله داده‌های تصویر c می‌باشند که با پردازنده گرافیکی g پردازش می‌شوند. به‌علاوه یک‌فاز ترکیب‌سازی انجام می‌شود تا داده‌های شکافته شده بازیافت

[۱] ON-Chip
[۲] Read-only
[۳] Multi-GPU

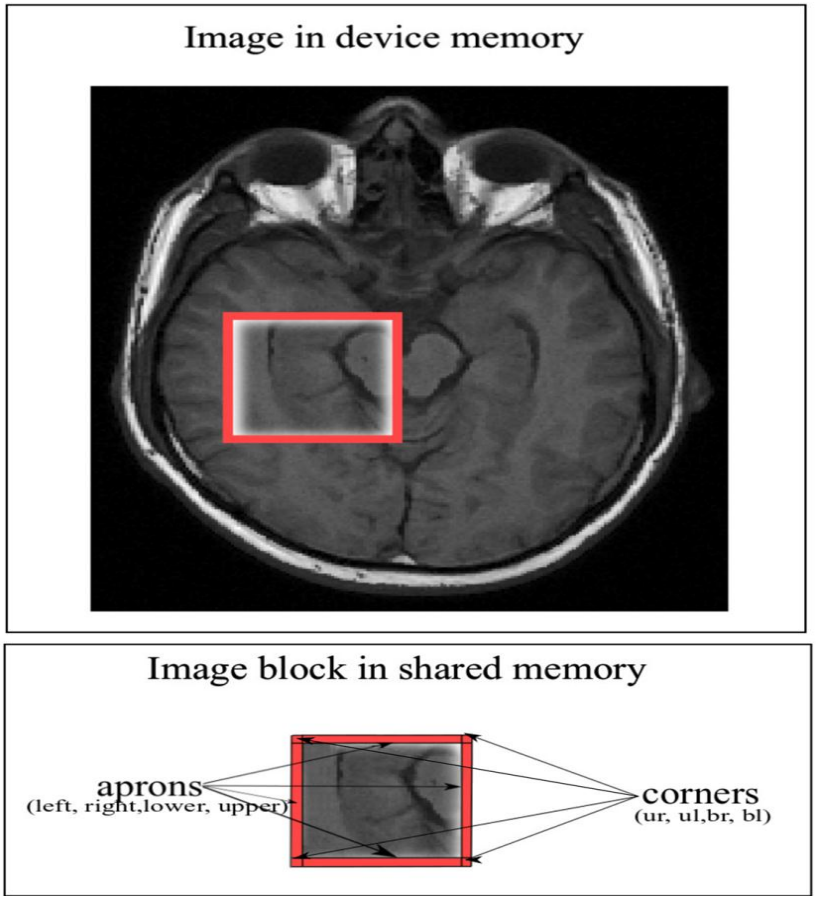
شده Ω را از قسمت‌های تصویر $\Omega^{g\,c}_{i\,j}$ گردآوری سازد. به‌طور خاص این فاز وجود حاشیه‌های اضافی را تحت اقدام قرار می‌دهد، که برای تضمین تصحیح سازی حذف نویز لازم بوده است.

به بیان ساده‌تر، فرض ما این است که داده‌ها و شاخص‌های شکافت i،j مثلاً برنامه‌ریز شده تا توسط یک پردازنده گرافیکی خاص،G،.....و g=۱ پردازش شود. سپس در تمام پژوهش ما از علامت اختصاری $\Omega^{c}_{i\,j}$ استفاده می‌کنیم.

یک روش ساده برای اجرا فیلتر در کودا این است که یک بلوک (block) از داده‌های تصویر را در آرایه حافظه مشترک بارگذاری کنیم که به‌صورت دینامیکی اختصاص داده‌شده است. هر بلوک تصویر با بلوک نخ هم سطر از پردازش می‌شود که فیلتر را روی پخش تصویر اجرایی می‌کند و مقادیر بازیافتی را به ساختار خروجی از حافظه دستگاه به ساختار پردازنده اصلی می‌نویسد.

برای هر کرنل بیشتر پیکسل‌ها در حاشیه بلوک حافظه مشترک به پیکسل‌هایی بستگی دارند که در حافظه مشترک بلوک بارگذاری نمی‌شوند. بنابراین، برای اجرا صحیح فیلتر، لازم است دسته‌های اضافی از پیکسل‌های بلوک در حافظه مشترک بارگذاری شود. اندازه این (apron) دسته به قسمت که بستگی دارد: در حالت برآورد سازی تطبیقی واریانس (الگوریتم۲) شعاع واریانس $a_v= (S_x،S_y،S_z)$ به (waprons) دسته‌ها داده می‌شود.

برای محاسبه حقیقی فیلتر (الگوریتم۳) اندازه دسته‌ها (aprons) مقدار پنجره (window) و شعاع‌های مشابه $a=(d_N+VY+d+V_x+،،،)$ است. بنابراین نخ‌های دارای threadldx.y□ay در مرز بالایی و پائینی هر بلوک بارگذاری خواهد شد و نخ‌های دارای threadldx.y□ax در مرزهای راست و چپ بارگذاری خواهند شد. شکل(۴-۲) aprons های (دسته‌های) اضافه‌شده بلوک‌ها را نشان می‌دهد.



شکل(۲ -۳) دسته‌های (apr ons) اضافه‌شده بلوک‌ها [۲۰]

اضافه بر این، نخ‌ها واکسل‌های گوشه‌ای بارگذاری خواهند شد. (به شکل۲-۳ مراجعه شود).به دلایل عملیاتی ،پیشنهاد می‌شود از شعاع‌های نسبتاً کوچک استفاده شود در غیر این صورت در محاسبه فیلتر تعداد زیادی نخs سرگردان وجود خواهد داشت. هر وقت اشغال سازی ظرفیت بیش از محدوده حافظه باشد، مقدار a_{dz} به‌تناسب کاهش می‌یابد.

ما دو نمونه متفاوت از فیلتر حذف نویز یک فیلتر تک مقوله‌ای و فیلتر چند مقوله‌ای را فعال می‌سازیم. در مورد دوم، سه تصویر متفاوت از یک بیمار برای الگوریتم فراهم می‌شود و یک پیکربندی متفاوت کرنل مورداستفاده قرار داده می‌شود. هدف این است که بلوک‌های کوچک‌تر ایجاد شود و مقدار حافظه مشترک محدود شود تا تمام داده‌های تصویری که از این سه مقوله می‌آید پوشش داده شود مشکل محدودیت حافظه برطرف شود. ما یک پیکربندی اندازه بلوک (۱،۸،۱۴) برای روش چند مقوله‌ای و (۱و۱۴و۱۴) را برای دیگر موارد مورد استفاده قرار می‌دهیم.

در حقیقت که در سه تابع اصلی نشان داده‌شده در الگوریتم ۱ سازمان‌دهی می‌شود.

۱-آغازسازی و مرحله شکافت

۲-محاسبه حقیقی واریانس تطبیقی (الگوریتم ۲)

۳-محاسبه سیگنال فیلتر شده NLM مطابق با گزینه‌های تنظیم‌شده با پارامترهای داده‌شده توسط کاربر (الگوریتم ۳)

الگوریتم‌های ۳-۱ ورژن چند مقوله‌ای را دارند دارای معادل عملیاتی دو ورژنی می‌باشند که ازلحاظ مقدار داده‌ای استفاده‌شده برای محاسبه بارها تفاوت دارند و به خاطر وجود سه تصویر برای هر بیمار تفاوت دارند. الگوریتم ۱ درخواست عملکرد با پارامترهای (۱/۰) ورودی/خروجی لازم همراه با درخواست به تخمین و برآورد سازی تطبیقی (حقیقتی) واریانس نویز و به محاسبه فیلتر حقیقتی Ω توضیح می‌دهد. فیلتر حذف نویز به‌صورت قسمت به قسمت برای تمام واکسل‌ها و پنجره و حجم متناظر محاسبه می‌شود. پارامترهای ۱/۰ ورودی /خروجی با چند (FLAGS) فلگ فعال می‌شوند که تنظیمات آن‌ها نوع نویز تأثیرگذار روی تصاویر، تعداد مقوله‌های تصاویر و ابعاد حجم و تعیین اندازه پنجره‌های مربوطه را تنظیم می‌کند:

ADAPTIVESIGMA: واریانس نویز ثابت ۱: تخمین تطبیقی موضعی واریانس

RICIAN: تقریب گذاری گاوسی برای نویز ۱: تصحیح سازی RICIAN. (معادله۳)

VOLUME: فیلترینگ ۲D(دوبعدی) ۱: فیلترینگ۳D

SIGMA_BLOCK_RADIUS , NLM_BLOCK_RADIUS: شعاع جستجوی پنجره $((V_x،V_y،V_z))$.

اضافه بر این، بعضی ساختارهای ساپورت کننده به‌صورت بافرهای دستگاه و بافرهای هاست برای هر مقوله را و شکافت j داده‌های فرعی i تعریف می‌شوند.

Ω^c_{ij}: بافرهای دستگاه برای بارگذاری داده‌ها است.

$(\Omega^c_{ij})^2$: بافرهای دستگاه برای بارگذاری تخمین واریانس نویز است.

$\square\Omega^c_{ij}$: بافرهای هاست دارای داده‌های فرعی(subdata) بازیابی شده است.

Algorithm ۱ NLM_GPU

$(\Omega^{۱}, \dots, \Omega^{c}, (\sigma^{۱})^{۲}, \dots, (\sigma^{C})^{۲}, N_x, N_y, N_z, \tilde{\Omega}^{۱}, \dots, \tilde{\Omega}^{C}$, $SIGMA\_BLOCK\_RADIUS$, NLM_BLOCK_RADIUS, NLM_WINDOW_RADIUS, ADAPTIVE_SIGMA, VOLUME, SIN_CLE_COMPONENT, RICIAN)

For i=۱ to N_x **do**

Generate data splits $\Omega_{i,j}^{۱}, \dots, \Omega_{i,j}^{C}$ and schedule them on G GPUs.
If ADAPTIVE_SIGMA **then**
For each split j of subdata i **do**
allocate and initialize host-side input data, device buffer and streams for asynchronous command execution
call **COMPUTE_ADAPTIVE_SIGMA**
$(\Omega_{i,j}^{۱}, \dots, \Omega_{i,j}^{C}, (\sigma_{i,j}^{۱})^{۲}, \dots., (\sigma_{i,j}^{C})^{۲}, SIGMA\_BLOCK\_RADIUS,\ NLM\_WINDOW\_RADIUS)$
end for
for each split j of subdata i **do**
call **NLM_KERNEL** $(\Omega_{i,j}^{۱}, \dots, \Omega_{i,j}^{C}, (\sigma_{i,j}^{۱})^{۲}, \dots., (\sigma_{i,j}^{C})^{۲}, N_x, N_y, N_z, \tilde{\Omega}_{i,j}^{۱}, \dots \tilde{\Omega}_{i,j}^{C}$ SIGMA_BLOCK_RADIUS, NLM_BLOCK_RADIUS, NLM_WINDOW_RADIUS, ADAPTIVE_SIGMA VOLUME, SIN_GLE_COMPONENT, RICIAN)
end for
end if
collect the restored data $\tilde{\Omega}^{۱}, \dots., \tilde{\Omega}^{C}$.
End for

شکل (۲-۴) : آغاز سازی و مرحله شکافت [۲۰]

ALGORITHM ۲ COMPUTE_ADAPTIVE_SIGMA $(\Omega_{i,j}^{۱}, \dots, \Omega_{i,j}^{C}, (\sigma_{i,j}^{۱})^{۲}, \dots, (\sigma_{i,j}^{C})^{۲}$, SIGMA_BLOCK_RADIUS, NLM_WINDOW_RADIUS)

for all the voxel x_i estimate the local variance **do**
for each CUDA block $b^i = (b_x^i, b_y^i, ۰)$ **do**
for each image $\Omega_{i,j}^{c}$ and each component c **do**
load the block of the image and corresponding aprons to the shared memory
Syncthreads ();
Compute $(\sigma_{i,j}^{۱})^{۲}, \dots, (\sigma_{i,j}^{C})^{۲}$ (Section ۲.۲)
end for
end for
end for

شکل (۲-۵): محاسبه حقیقی واریانس تطبیقی [۲۰]

Algorithm ۳ NLM_KERNEL ($\Omega_{i,j}^{1}, \dots, \Omega_{i,j}^{C}, (\sigma_{i,j}^{1})^{2}, \dots, (\sigma_{i,j}^{C})^{2}$, SIGMA_BLOCK_RADIUS, NLM_BLOCK_RADIUS, NLM_WINDOW_RADIUS, ADAPTIVE_SIGMA, VOLUME< SINGLE_COMPONENT, RICIAN)

for all the voxel x_i **do**
  for each CUDA block $b^i = (b_x^i , b_y^i , ۰)$ **do**
    for each image $\Omega_{i,j}^{c}$ **do**
      load the block of the image and corresponding aprons to the shared memory
      Syncthreads ();
      compute the restored intensity $\tilde{u}_i^c$ (Eq. (۱))
      if RICIAN **then**
        apply Rician correction to $\tilde{u}_i^c$ (Eq. (۳))
      end if
      normalize the restored value $\tilde{u}_i^c$ and copy it to the global memory
    end for
  end for
end for

شکل (۲-۶): محاسبه سیگنال فیلتر شدهNLM مطابق با گزینه‌های تنظیم‌شده با پارامترهای داده‌شده توسط کاربر [۲۰]

فصل سوم:
روش کار

۳-۱ مقدمه

بخش عمده‌ای از تعهدات محاسباتی پردازنده گرافیکی در بسیاری از مشکلات کاربردی ساختارهای موازی نهفته شده است.در این حالت، قصد داریم با گذراندن از این فصل به چگونگی اجرای کدهای موازی در پردازنده گرافیکی با استفاده از کودا C بپردازیم.

۳-۲ اهداف این فصل

در طی این فصل از پایان‌نامه، موارد زیر را می‌آموزید:

یکی از راه‌های بنیادی برای نمایش دادن کودا که موازات است را می‌آموزید.

در مورد شیوه‌ای که از طریق آن کودا C به فراخوانی رشته‌ها می‌پردازد را می‌آموزید. مکانیزم به‌کارگیری رشته‌های مختلف برای برقراری ارتباط با یکدیگر را می‌آموزید. مکانیزمی برای هم‌زمان‌سازی اجرای موازی رشته‌های مختلف را می‌آموزید.

۳-۳ برنامه‌نویسی کودا موازی

پیش‌ازاین چگونگی اجرای آسان تابع استاندارد C را بر روی یک دستگاه دیدیم. با اضافه کردن توصیف‌کننده__ global__ (توابع سراسری) به تابع و با فراخوانی آن به‌عنوان یک دستورالعمل پرانتز زاویه‌دار خاص [] ((براکت ها) متغیرهایی را مشخص می‌کنند که قصد داریم آن‌ها را از سیستم‌های زمان اجرا عبور دهیم) تابع را بر روی پردازنده گرافیکی اجرا کردیم. اگرچه این اجرا بسیار ساده و ناکارآمد بود چراکه عوامل مهندسی سخت‌افزار ان ویدیا نسبت به عملکرد صدها پردازنده‌ی محاسباتی موازی، در حال بهینه‌سازی پردازنده‌های گرافیکی می‌باشند.

بااین‌حال، تاکنون فقط اقدام به ساخت هسته‌ای کرده‌ایم که به‌صورت ردیفی بر روی پردازنده گرافیکی اجرا می‌شود. در این فصل، می‌بینیم که برای راه‌اندازی یک هسته به چه صورت دستگاه محاسبات خود را به‌صورت موازی انجام می‌دهد.

۳-۳-۱ تقسیم بلوک‌های موازی

ما در فصل گذشته به نحوه‌ی راه‌اندازی کد موازی بر روی پردازنده گرافیکی به‌صورت گذرا پرداختیم. این کار با هدایت و کنترل سیستم زمان اجرای کودا در بسیاری از نسخه‌های موازی کرنل برای راه‌اندازی انجام می‌شود. ما به این نسخه‌های موازی بلوک میگوییم. زمان اجرای کودا به این

بلوک‌ها اجازه تقسیم به رشته‌ها را می‌دهد. به خاطر داشته باشید که آرگومان اول در براکت‌های (علامت‌های کوچک‌تر و بزرگ‌تر) زاویه‌ای در زمان راه‌اندازی بلوک‌های موازی متعدد از شماره ۱ تا تعداد نهایی بلوک‌هایی که ما قصد راه‌اندازی آن‌ها را داریم، تغییر داده می‌شود. برای مثال، ما در حین بررسی جمع برداری،از طریق فراخوانی زیر به راه‌اندازی یک بلوک برای هر یک از عناصر موجود در برداری با اندازه N دست می‌زنیم:

```
add<<< N,۱>>>( dev_a, dev_b, dev_c )
```

درواقع، پارامتر دوم در داخل براکت‌های زاویه‌ای نشان‌دهنده تعداد رشته‌ها در هر بلوکی است که ما از جانب خود خواهان ایجاد آن در زمان اجرای کوداهستیم. تا این مرحله، ما تنها یک رشته را در هر بلوک راه‌اندازی کرده‌ایم. در مثال قبلی به راه‌اندازی مواردی به شرح زیر اقدام شد [۱۸].

فرمول(۳-۱) رشته‌های موازی = N بلوک ۱ /رشته × Nبلوک

پس درواقع می‌توانیم بلوک‌های N/۲ را با دو رشته در هر بلوک و بلوک‌های N/۴ را با چهار رشته در هر بلوک و ... راه‌اندازی کنیم. بیایید بار دیگر مثال جمع برداری مجهز شده با این اطلاعات جدید را در مورد قابلیت‌های CODA C مرور کنیم [۱۸].

۳-۳-۲مجموع برداری: ردوکس[۱]

ما برای به انجام رساندن وظیفه‌ای مشابه آنچه در فصل قبل انجام دادیم، تلاش می‌کنیم. به این معنا که ما می‌خواهیم دو بردار ورودی را گرفته و جمع آن‌ها را در یک بردار خروجی سوم ذخیره کنیم. بااین‌حال، در این زمان برای تحقق این امر بجای بلوک‌ها از رشته‌ها استفاده می‌کنیم .شاید برایتان این سؤال پیش‌آمده باشد که مزیت استفاده از رشته‌ها به‌جای بلوک‌ها در چیست؟ خب، در حال حاضر هیچ مزیتی که ارزش بحث را داشته باشد، وجود ندارد. اما رشته‌های موازی موجود در یک بلوک توانایی انجام کارهایی را دارند که از عهده بلوک‌های موازی برنمی‌آید [۱۸].

[۱] REDUX

۳-۳-۲-۱ مجموع برداری پردازنده گرافیکی استفاده‌کننده از رشته‌ها

ما کار را با پرداختن به دو تغییر قابل‌توجه در هنگام حرکت از بلوک‌های موازی به رشته‌های موازی آغاز می‌کنیم. فراخوانی هسته از جایی شروع به تغییر می‌کند که بلوک‌های N از یک نمونه رشته راه‌اندازی می‌شوند:

(dev _ a, dev _ b, dev _ c)<<<>> را به نسخه‌ای که همه رشته‌های N را در یک بلوک راه‌اندازی می‌کند، اضافه کنید:

add<<<>>(dev _ a, dev _ b, dev _ c)

تنها دیگر تغییرات پیش‌آمده در این روش، ما را قادر به شاخص بندی داده‌ها می‌کنند. پیش‌ازاین ما توانستیم داده‌های ورودی و خروجی را با استفاده از شاخص بلوکی در درون‌هسته فهرست بندی کنیم. در اینجا وجود این جمله نباید تعجب‌برانگیز باشد. حال که ما تنها از یک بلوک برخورداریم بایست داده‌ها را با استفاده از شاخص رشته‌ای فهرست بندی کنیم. این دو تغییر تنها تغییراتی هستند که به تغییر از پیاده‌سازی بلوک‌های موازی به پیاده‌سازی رشته‌های موازی نیاز دارند. برای تکمیل کار در اینجا لازم است که لیست کامل منبع همراه با خطوط تغییر به‌صورت پررنگ نشان داده شوند:

```
#include
define N 10
global void add( int *a, int *b, int *c) {
int tid =threadIdx.x;
[tid] < N [cltid] = [altid] +b[tid];
int main(void) {
int a[N], b[N). C(N);
int dev_a, dev_b, dev_c;
// allocate the memory on the GPu
HANDLE_ERROR( cudaMalloc (vold.*)dev_a, N.sizeot (int) );
HANDLE_ERROR( cuciaMalloc( (void) &dev_b, N. sizeof(int) ):
HANDLE_ERROR( cuciaMalloc( (void) &dev_c. N. sizeof(int) )
// F111 che arrays a' and b' on the CPU
 for (int i=0; i<n; i++) {
a[i] = 1; b[i] = 1;
// copy the arrays ta' and 'b' to the GPU
HANDLE_ERROR( cudamencpy( dev_a.N. sizeof(int).cudamemcpyHostToDevice));
```

```
HANDLE_ERROR( cudaNemcpy( dev_b.N. sizeof(int), cudaMemcpy HostToDevice } );
Add<<<1,N>>> (dev_a, dev_b, dev_e );
```

شکل(۳-۱) فهرست بندی شاخص بلوکی در درون‌هسته داده‌های ورودی و خروجی

خیلی ساده است، مگر نه؟ در بخش بعدی، ما با یکی از محدودیت‌های این رویکرد فقط رشته‌ای مواجه می‌شویم و البته بعداً درخواهیم یافت که چرا تقسیم بلوک‌ها به اجزای موازی دیگر بدون زحمت نیست.

۳-۳-۲-۲مجموعه یک بردار بلندتر پردازنده گرافیکی

ما در فصل گذشته متوجه شدیم که سخت‌افزار تعداد بلوک‌ها در یک‌راه اندازی واحد را تا ۶۵.۵۳۵ کاهش می‌دهد. همچنین، سخت‌افزار تعداد رشته‌های موجود در هر بلوک که با آن قادر به راه‌اندازی هسته است را کم می‌کند. به‌طور خاص، این تعداد نمی‌تواند از مقدار مشخص‌شده توسط فیلد maxThreadsPerBlock(حداکثر رشته‌های موجود در هر بلوک) مربوط به ساختار خصوصیات دستگاه مشاهده‌شده در فصل ۲ فراتر برود. در حال حاضر برای بسیاری از پردازنده‌های گرافیکی در دسترس، این محدودیت به میزان ۵۱۲ رشته در هر بلوک است بنابراین این سؤال مطرح می‌شود که چگونه می‌توان با استفاده از یک رویکرد مبتنی بر رشته به‌اضافه کردن دو بردار با اندازه‌های بزرگ‌تر از ۵۱۲ مبادرت ورزید؟ ما برای تحقق این امر ملزم به استفاده ترکیبی از رشته‌ها و بلوک‌ها هستیم. درست مثل قبل در اینجا نیز به دو تغییر نیاز است: ما باید محاسبه شاخص در هسته و همچنین خود راه‌اندازی هسته را تغییر دهیم. حال که ما از تعداد متعددی بلوک و رشته برخورداریم، نمایه‌سازی مربوطه شبیه به روش استاندارد برای تبدیل فضای شاخص دوبعدی به یک فضای خطی آغاز خواهد شد.

برای انجام این کار از یک متغیر جدید ساخته‌شده به نام blockDim استفاده می‌شود. این متغیر برای همه بلوک‌ها ثابت است و تعداد رشته‌های موجود در طول هر بعد از بلوک را ذخیره می‌کند و ازآنجاکه ما در حال استفاده از یک بلوک تک‌بعدی هستیم تنها به x.blockDim رجوع می‌کنیم. اگر به یاد داشته باشید gridDim یک مقدار مشابه را ذخیره می‌کرد ولی درعین‌حال تعداد بلوک‌های موجود در امتداد هر یک از ابعاد کل شبکه را نیز ذخیره می‌کرد. علاوه بر این، gridDim دوبعدی است درحالی‌که blockDim درواقع سه‌بعدی است. به‌عبارت‌دیگر، زمان اجرای کودا به شما اجازه راه‌اندازی یک شبکه دوبعدی از بلوک‌ها که در آن هر بلوک آرایه‌ای سه‌بعدی از رشته‌هاست را

می‌دهد. بله، ابعاد بسیاری وجود دارند و بعید است که شما مرتباً به پنج درجه آزادی نمایه‌سازی فراهم‌شده نیاز پیدا کنید. اما در صورت تمایل در دسترس می‌باشند.

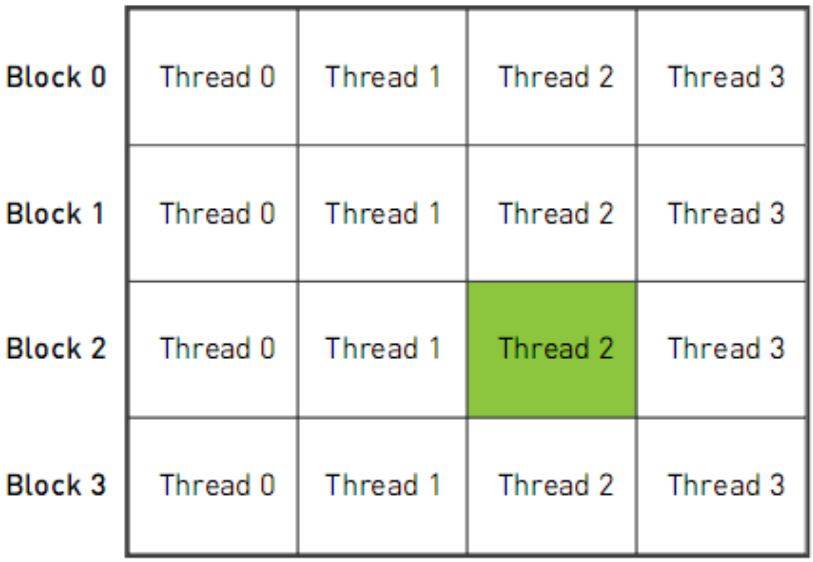
درواقع نمایه‌سازی داده‌ها در یک آرایه خطی با کمک گرفتن از وظیفه قبلی کاملاً حسی است. اگر شما با این ایده مخالفید بهتر است بدانید که ممکن است به شما کمک کند تا در مورد مجموعه بلوک‌های رشته‌ها همانند آرایه دوبعدی پیکسل‌ها به‌صورت فضایی فکر کنید. ما در شکل ۲ـ۱ این ترتیب را به تصویر کشیده‌ایم. اگر رشته‌ها نشان‌دهنده ستون‌ها و بلوک‌ها نشان‌دهنده ردیف‌ها باشند، می‌توانیم با در نظر گرفتن محصول شاخص بلوک همراه با تعداد رشته‌ها در هر بلوک و اضافه کردن شاخص رشته‌ای به داخل بلوک به یک شاخص منحصربه‌فرد دست‌یابیم. این در مورد روش مورداستفاده ما در طولی کردن شاخص تصویر دوبعدی در مثال مجموعه (Set Julia) نیز صدق می‌کند.

در اینجا، DIM همان بعد بلوکی است (که در رشته‌ها اندازه‌گیری می‌شود)، Y شاخص بلوکی و X شاخص رشته‌ای داخل بلوک است. ازاین‌رو ما به شرح ذیل به شاخص دست میابیم:

```
int tid = threadIdx.x + blockIdx.x * blockdim.x;
```

شکل(۳-۲): تعریف بلوک و نخ‌ها

تغییر دیگر راه‌اندازی خود هسته است. ما هنوز به رشته‌های موازی N جهت راه‌اندازی نیاز داریم اما خواهان راه‌اندازی آن‌ها در سرتاسر بلوک‌های چندگانه هستیم و بنابراین محدودیت ۵۱۲ رشته‌ای تحمیل‌شده را مدنظر قرار نمی‌دهیم. یک‌راه حل، تنظیم دلخواه اندازه بلوک به تعداد ثابتی از رشته‌ها است. بگذارید در این مثال از ۱۲۱ رشته در هر بلوک استفاده کنیم. سپس تنها کاری که می‌توانیم انجام دهیم راه‌اندازی N/۱۲۸ جهت به دست آوردن تعداد کلی رشته‌های در حال اجرا است. مسئله پیچیده در اینجا این است که N/۱۲۸ درواقع یک تقسیم عدد صحیح است و این حاکی از آن است که در صورت ۱۲۲ بودن تعداد، N/۱۲۸ برابر با صفر خواهد بود و اگر ما اقدام به راه‌اندازی کنیم درواقع دست به محاسبه هیچ‌چیزی نزده‌ایم. درواقع، ما تعداد بسیار کمی از رشته‌ها را درزمانی که N مضرب دقیقی از ۱۲۱ نیست راه‌اندازی خواهیم کرد و این خوب نیست. در حقیقت ما قصد گرد کردن افزایشی این تقسیم را داریم [۱۹].



شکل (۳-۳) ترتیب دوبعدی مجموعه‌ای از بلوک‌ها و رشته‌ها و رشته‌های صفر [۱۹]

ترفند مرسومی برای تحقق این امر در تقسیم عدد صحیح آن‌هم بدون فراخوانی ceil() وجود دارد. ما عمل $(N + ۱۲۷) / ۱۲۸$ را بجای N/۱۲۸ محاسبه می‌کنیم. درهرصورت یا شما منظور ما را در مورد اینکه کوچک‌ترین مضرب ۱۲۱ بزرگ‌تر یا مساوی N محاسبه می‌شود متوجه خواهید شد و یا همین حال باید وقتی را برای فکر کردن به این واقعیت جهت متقاعد کردن خود اختصاص دهید. ما در هر بلوک ۱۲۱ رشته را انتخاب کرده‌ایم و به همین دلیل از روش زیر برای راه‌اندازی هسته استفاده می‌کنیم:

```
Add<<< (N+127)/128, 128 >>>( dev_a, dev_b, dev_c);
```

شکل(۳-۴) : راه‌اندازی N/۱۲۸ جهت به دست آوردن تعداد کلی رشته‌های در حال اجرا

به دلیل تغییر صورت گرفته در تقسیم است که از راه‌اندازی رشته‌های کافی مطمئن می‌شویم، درواقع هنگامی‌که N مضرب دقیق ۱۲۱ نیست می‌توانیم رشته‌های بسیاری را راه‌اندازی کنیم. اما یک چاره ساده‌تر هم برای این مشکل وجود دارد که هسته ما پیش‌تر به آن پرداخته است. ما باید این موضوع را بررسی کنیم که آیا افست یک‌رشته پیش از استفاده از آن برای دسترسی به آرایه‌های خروجی و ورودی درواقع بین ۰ و N هست یا خیر:

```
if (tid < N)
c[tid] = a [tid] + b[tid];
```

شکل (۳-۵) : بررسی آرایه‌های خروجی و ورودی بین ۰ و N

بنابراین هنگامی‌که شاخص ما انتهای آرایه را فرا جهش می‌کند مثل زمانی که یک‌جا مضرب ۱۲۱ فعال می‌شود، ما به‌طور خودکار از انجام محاسبه خودداری می‌کنیم. مهم‌تر آنکه ما از خواندن و نوشتن حافظه در انتهای آرایه‌مان خودداری می‌کنیم. مجموع بردارهای به‌طور دلخواه بلند پردازنده گرافیکی زمانی که ما برای اولین بار به بحث در مورد راه‌اندازی بلوک‌های موازی در پردازنده گرافیکی پرداختیم به‌طور کامل آمادگی ارائه آن را نداشتیم. علاوه بر محدودیت در تعداد رشته‌ها، محدودیت‌های سخت‌افزاری هم در تعداد بلوک‌ها وجود دارد (اگرچه خیلی بیشتر از محدودیت‌های رشته‌ای است). همان‌طور که قبلاً ذکر شد هیچ‌یک از ابعاد شبکه بلوک‌ها نمی‌توانند از ۲۵،۵۳۵ فراتر روند. بنابراین این موضوع مشکلی را با اجرای جاری جمع برداری پدید می‌آورد. اگر به‌منظور اضافه کردن بردارهایمان به راه‌اندازی بلوک‌های N/۱۲۸ بپردازیم در زمان فراتر رفتن بردارمان از ۶۵،۵۳۵* ۱۲۸=۸،۳۸۸،۴۸۰ عنصر خواهیم توانست مشکلات راه‌اندازی را با موفقیت پشت سر بگذاریم. مقدار این عدد به نظر زیاد می‌آید ولی با ظرفیت حافظه جاری بین GB۱ و GB ۴، پردازنده‌های گرافیکی فوق پیشرفته (end-high) می‌توانند مقیاس اطلاعاتی گسترده‌تری را نسبت به بردارهای ۸ میلیون عنصر در خود جای دهند. خوشبختانه، راه‌حل این موضوع بسیار ساده است. ما برای اولین بار تغییری را در هسته ایجاد می‌کنیم.

این امر تداعی‌کننده‌ی نسخه اصلی جمع برداری است! درواقع، شما باید آن را با پیاده‌سازی واحد پردازنده مرکزی (پردازنده اصلی) فصل قبل مقایسه کنید:

```
void add( int *a, int *b, int *c ) {
int tid = 0;
// this is CPU zero, so we start at zero
while (tid < N) {
C[tid] = a [tid] + b[tid] ;
 tid += 1;
 // we have one CPU, so we increment by one
}
}
```

شکل(۳-۶) : در اینجا ما از حلقه ()while جهت انجام تکرار از طریق داده‌ها استفاده می‌کنیم

به یاد بیاورید که ما به‌جای افزایش شاخص آرایه تا ۱، ادعای وجود یک نسخه چندهسته‌ای یا دارای چند واحد پردازنده مرکزی را می‌کنیم که می‌تواند توسط تعدادی از پردازنده‌های مورداستفاده ما در اینجا توسعه یابد. ما در حال حاضر از اصول مشابهی در نسخه پردازنده گرافیکی پیروی می‌کنیم. برای اجرای پردازنده گرافیکی باید تعدادی از رشته‌های موازی راه‌اندازی شده جهت کار به‌عنوان تعدادی از

پردازنده‌ها مدنظر قرار گیرند. گرچه ممکن است پردازنده گرافیکی واقعی نسبت به این مورد از واحدهای پردازشی کمتری (و یا بیشتری) برخوردار باشد ولی هر رشته ازلحاظ منطقی به‌صورت موازی اجرا می‌شود و سپس به سخت‌افزار اجازه می‌دهد تا اجرای واقعی را زمان‌بندی کند. افتراق (Decoupling)[1]موازی‌سازی از روش واقعی اجرای سخت‌افزاری یکی از مشکلات اساسی است که کودا C از سر راه طراح نرم‌افزاری برمی‌دارد. با در نظر گرفتن اینکه سخت‌افزار فعلی ان ویدیاقابلیت انتقال به هر نقطه‌ای در محدوده بین ۱ و ۴۱۰ واحد حسابی در هر تراشه را دارد می‌توان به‌راحتی هر چه بیشتر کار دست‌یافت! حال که ما با ضوابط و اصول پنهان‌شده در فراسوی این پیاده‌سازی آشنا شدیم تنها نیاز است که نحوه تعیین ارزش شاخص اولیه برای هر رشته موازی و نحوه افزایش آن را درک کنیم. ما می‌خواهیم که هر یک از رشته‌های موازی بر مبنای یک شاخص داده متفاوت شروع به کار کند بنابراین تنها کاری که نیاز است گرفتن رشته‌ها و بلوکه کردن شاخص‌ها و به‌صورت طولی درآوردن آن‌ها طبق آن چیزی است که در بخش "مجموع بردار طولانی‌تر پردازنده گرافیکی" دیدیم. هر رشته در شاخصی به شرح زیر شروع بکار می‌کند:

```
int tid =threadIdx.x + blockIdx .x * blockdim.x;
```

شکل (۳-۷)

به یاد بیاورید که ما به‌جای افزایش شاخص آرایه تا ۱، ادعای وجود یک نسخه چندهسته‌ای یا دارای چند واحد پردازنده مرکزی را می‌کنیم که می‌تواند توسط تعدادی از پردازنده‌های مورداستفاده ما در اینجا توسعه یابد. ما در حال حاضر از اصول مشابهی در نسخه پردازنده گرافیکی پیروی می‌کنیم. برای اجرای پردازنده گرافیکی باید تعدادی از رشته‌های موازی راه‌اندازی شده جهت کار به‌عنوان تعدادی از پردازنده‌ها مدنظر قرار گیرند. گرچه ممکن است پردازنده گرافیکی واقعی نسبت به این مورد از واحدهای پردازشی کمتری (و یا بیشتری) برخوردار باشد. ازآنجاکه ما علاقه به کپی و پیست کردن (چسباندن) هستیم از ۱۲۱ بلوک که هر یک حاوی ۱۲۱ رشته می‌باشند استفاده خواهیم کرد:

```
Add<<< 128, 128 >>>( dev_a, dev_b, dev_c);
```

شکل (۳-۸)

شما باید برای تنظیم این مقادیر احساس راحتی کنید حتی اگر آن‌ها مناسب به نظر می‌رسیدند البته به‌شرط آنکه این مقادیر در چهارچوب محدودیت‌های تشریح شده از سوی ما باشند. بعدها در همین پایان‌نامه در مورد پیامدهای عملکرد بالقوه این انتخاب‌ها بحث خواهیم کرد اما در حال حاضر برای

[1] is proposed for HetNets for improving system performance

انتخاب ۱۲۱ رشته در هر بلوک و ۱۲۱ بلوک به همین میزان بسنده می‌کنیم. حال می‌توانیم بردارهایی با طول دلخواه که به مقدار پردازنده گرافیکی روی RAM محدودشده‌اند را اضافه کنیم. در اینجا لیست کل منابع موجود است.

۳-۳-۲-۳ موج‌دار کردن (ایجاد حلقه‌های موجی در) پردازنده گرافیکی با استفاده از رشته‌ها

مطابق فصل گذشته، از طریق ارائه یک مثال جالب که نشان‌دهنده برخی از رشته‌های مورداستفاده ما است، جمع برداری را نشان می‌دهیم و دوباره از قدرت محاسباتی پردازنده گرافیکی برای تولید تصاویر به‌صورت رویه‌ای استفاده می‌کنیم. اما نگران نباشید چراکه ما همه کدهای نامربوط انیمیشنی را به شکل توابع کمکی بسته‌بندی می‌کنیم بنابراین شما مجبور به پیدا کردن مهارت گرافیکی یا انیمیشنی نخواهید بود.

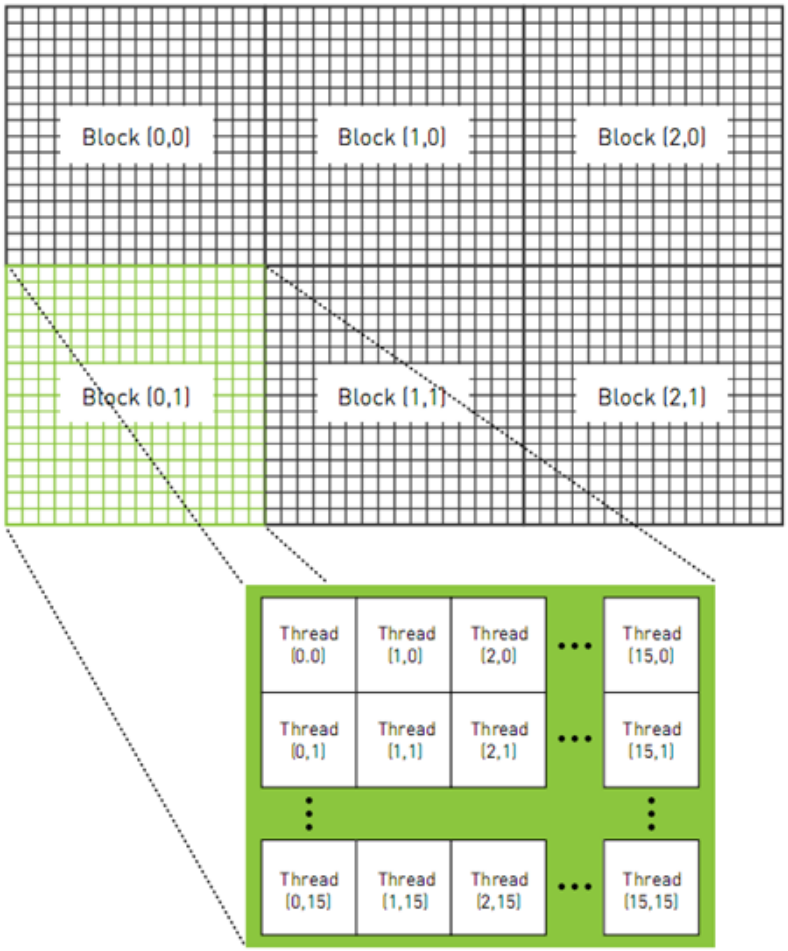
بخش اعظمی از پیچیدگی ()main در کالس۱ کمکی CPUAnimBitmap پنهان‌شده است. شما در آینده نزدیک متوجه خواهید شد که ما دوباره از الگوی انجام کودا ()Malloc که کار اجرای کد دستگاهی استفاده‌کننده از حافظه اختصاصی و سپس تمیز کردن با کودا ()Free را بر عهده دارد، برخورداریم. این در حال حاضر قدیمی به نظر می‌رسد. در این مثال، ما شیوه‌ای را که توسط آن قادر به تکمیل مرحله میانی یعنی " مرحله اجرای کد دستگاهی استفاده‌کننده از حافظه اختصاصی هستیم " راکمی پیچیده کرده‌ایم. ما شیوه ()exit_and_anim را به‌عنوان یک اشاره‌گر تابع به frame_generate ()موردقبول قراردادیم. این تابع هر بار که می‌خواهد یک فریم جدید انیمیشنی را تولید کند توسط کالس فراخوانی می‌شود [۲۰].

```
void generate_frame ( DataBlock *d, int ticks ) {
dim3 blocks (DIM /16, DIM/16);
 dim 3 threads (16,16);
 kernel<<< blocks, threads >>>(d=>dev_bit map, ticks);
HANDLE_ERROR( cudaMemcpy( d->bitmap->get_ptr();
                                d=>dev_bitmap.
                                D=>bitmap=>image_size (),
                                cudaMemcpyDeviceToHost ) );
```

شکل (۳-۹) : راه‌اندازی بلوک‌های DIM/۱۶ * DIM/۱۶ جهت حصول یک‌رشته در هر پیکسل

[۱] The third parameter specifies the amount of shared memory per block to be dynamically allocated

اگرچه این تابع تنها شامل چهار خط می‌شود ولی همه آن‌ها دربرگیرنده مفاهیم مهم کودا C هستند. ما در وهله اول دو متغیر دوبعدی، بلوک‌ها و رشته‌ها را اعلام کرده بودیم. بنا به آنچه کنوانسیون نام‌گذاری با زحمت بسیار آشکار ساخته است، بلوک‌های متغیر نشان‌دهنده تعداد بلوک‌های موازی می‌باشند که ما در شبکه راه‌اندازی کرده‌ایم. رشته‌های متغیر نیز نشان‌دهنده تعداد رشته‌هایی هستند که ما در هر بلوک راه‌اندازی خواهیم کرد. ازآنجاکه ما در حال تولید یک تصویر هستیم، می‌توانیم از نمایه‌سازی دوبعدی به نحوی استفاده کنیم که هر رشته از یک شاخص منحصربه‌فرد (X,Y) که به‌راحتی قابل مطابقت با یک پیکسل در تصویر خروجی باشد، برخوردار گردد. ما استفاده از بلوک‌های حاوی آرایه ۱۲×۱۲ رشته‌ها را برمی‌گزینیم. اگر تصویر دارای پیکسل‌های DIM * DIM باشد، به راه‌اندازی بلوک‌های DIM/۱۶ * DIM/۱۶ جهت حصول یک‌رشته در هر پیکسل نیاز است. شکل (۳-۳) نحوه‌ای که این بلوک و پیکربندی رشته در یک تصویر (به‌طور مضحک) کوچک با عرض ۴۱ پیکسل و ارتفاع ۳۲ پیکسل به نظر می‌رسند را نشان می‌دهد [۲۰].



شکل(۳-۱۰) یک سلسله‌مراتب دوبعدی از بلوک‌ها و رشته‌های مورداستفاده در پردازش یک تصویر ۳۲ * ۴۸ پیکسلی با استفاده از یک‌رشته در هر پیکسل [۲۰]

ممکن است در صورت انجام هرگونه برنامه‌نویسی چند رشته‌ای واحد پردازنده مرکزی، سؤال مطرح شود که چرا می‌خواهیم این تعداد بسیار زیاد از رشته‌ها را راه‌اندازی کنیم. به‌عنوان‌مثال، این روش به‌منظور رندر کردن یک انیمیشن ۱۹۲۰×۱۰۸۰ باکیفیت و وضوح‌بالا شروع به ایجاد بیش از ۲ میلیون رشته می‌کند. اگرچه ما به‌طورمعمول به ایجاد و برنامه‌ریزی زمانی این تعداد فراوان از رشته‌ها در پردازنده گرافیکی مبادرت می‌ورزیم ولی رؤیای ایجاد این تعداد رشته در پردازنده اصلی بعید به نظر می‌رسد. ازآنجاکه زمان‌بندی و مدیریت رشته‌ای پردازنده اصلی در نرم‌افزار انجام می‌شود، نمی‌تواند به‌سادگی به تعداد رشته‌هایی مقیاس بندی شود که پردازنده گرافیکی مقیاس بندی می‌شود. ازآنجاکه ما صرفاً به ایجاد یک‌رشته برای هر عنصر اطلاعاتی دست می‌زنیم که قصد پردازش اجرا داریم، برنامه‌نویسی موازی در پردازنده گرافیکی می‌تواند به‌مراتب ساده‌تر از برنامه‌نویسی موازی در پردازنده اصلی باشد. پس از اعلام متغیرهای نگه‌دارنده ابعاد راه‌اندازی، کرنل مقادیر پیکسلی را به‌سادگی محاسبه می‌کند.

کرنل به دو قطعه از اطلاعاتی که ما به‌عنوان پارامتر می‌پذیریم، نیاز دارد. در ابتدا، به یک اشاره‌گر به حافظه دستگاهی دارای پیکسل‌های خروجی نیاز است. این یک متغیر جهانی است که حافظه خود را به Main()اختصاص داده است. اما متغیر تنها برای کد میزبان "جهانی" است، بنابراین ما باید آن را به‌عنوان پارامتری جهت اطمینان از اینکه زمان اجرای کودا آن را برای کد دستگاهی قابل‌دسترس می‌سازد مورد تائید قرار دهیم. در مرحله بعد، هسته ما نیاز به دانستن زمان جاری انیمیشن دارد پس می‌تواند فریم درستی را تولید کند. زمان جاری یا همان تیک‌ها از کد زیر ساختاری موجود در CPUAnimBitmap به تابع frame_generate()انتقال می‌یابند و بنابراین می‌توانیم به‌راحتی آن را به کرنل نیز انتقال دهیم. حالا نوبت به کد کرنل می‌رسد:

```
_global_void kernel ( unsigned char *ptr, int ticks ) {
// map from threadIdx/BlockIdx to pixel position
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadlaxy + blockIdx.y.blockDim.y:
 int offset=x+y* blockDim.x. gridDim.x;
// now calculate the value at that position
float fx =x - DIM/2;
float fyy - DIM/2;
floatd = sqrtf( fx fx + fyfy);
unsigned char grey = (unsigned char) (128.0f + 127.0f •
            cos (d/10.0f - ticks/7.0f) /
```

```
(d/10.01 + 1.0f));
ptr (offset*4 + 0] = grey;
ptr (offset *4 + 1] = grey;
ptr (offset*4 + 2) = grey;
ptr (offset *4 + 3] - 255;
```

به شرط اول در کرنل از همه مهمتر هستند.

```
int x =threadIdx.x + blockIdx.x * blockDim. X;
int y=threadIdx.y+ blockIdx.y* blockDim. y;
int offset = X + Y* blockDim.x * gridDim.x;
```

شکل (۳-۱۱) : اشاره‌گر به حافظه دستگاهی دارای پیکسل‌های خروجی

در این خطوط هر رشته به شاخص خود در بلوک و شاخص بلوک خود در شبکه دست می‌یابد و آن‌ها را به یک شاخص منحصربه‌فرد (y,x) در تصویر ترجمه می‌کند. بنابراین، هنگامی‌که رشته در نمایه (۵،۳) بلوک (۱،۱۲) شروع به اجرا می‌کند، می‌داند که ۱۲ بلوک کامل در سمت چپ و ۱ بلوک کامل دیگر در بالایش وجود دارند. در داخل این بلوک، رشته (۵،۳) از سه رشته در سمت چپ و پنج رشته در بالای خود برخوردار است. ازآنجاکه در هر بلوک ۱۲رشته وجود دارد، این معنا تداعی می‌شود که رشته مورد سؤال از موارد زیر برخوردار است:

۳رشته + ۱۲ بلوک × رشته / بلوک ۱۶=۱۹۵ رشته در سمت چپ آن

۵رشته + ۸ بلوک × رشته / بلوک ۱۶=۱۲۸ در بالای آن رشته

این محاسبه با محاسبه x وy در دو خط اول یکسان است و نحوه‌ای را که شاخص‌های رشته و بلوک به مختصات تصویر ترسیم می‌شوند را به ما نشان می‌دهد. سپس ما به‌منظور انتقال یک افست به بافر خروجی به‌راحتی مقادیر x وy را به‌صورت طولی درمی‌آوریم که بازهم مشابه آن چیزی است که ما در بخش‌های " مجموعه یک بردار بلندتر "پردازنده گرافیکی " و "مجموعه بردارهای به‌طور دلخواهانه بلند پردازنده گرافیکی " انجام دادیم. ما می‌دانیم که رشته باید کدام پیکسل (y,x) در تصویر را موردمحاسبه قرار دهد. همچنین ما زمانی را که به محاسبه این مقدار نیاز است را می‌دانیم و می‌توانیم هر تابع (t,y,x) را محاسبه و این مقدار را در بافر۱ خروجی ذخیره کنیم. در این حالت، تابع

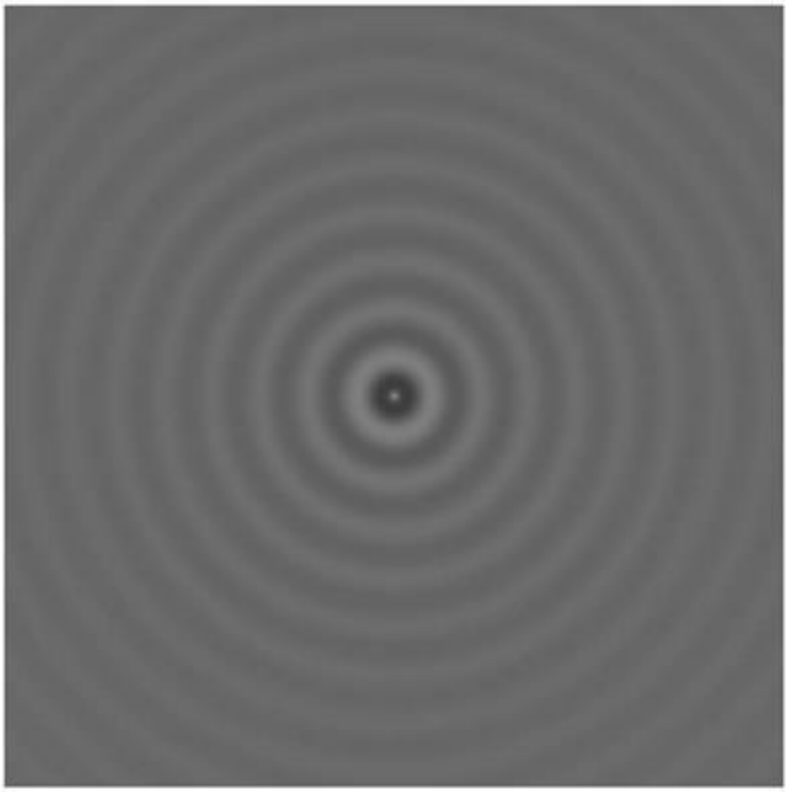
[۱] Buffer

یک "موج" سینوسی با زمان‌های مختلف را تولید می‌کند.

```
float fx = x - DIM/ 2;
float fy = Y - DIM/ 2;
float d = sqrtf( fx * fx + Ey + fy);
unsigned char grey - (unsigned char) (128.0f + 127.0f *
        cos (d/10.0f - ticks/7.0f) / (d/10.0f + 1.0f));
```

شکل (۳-۱۲) : مجموعه یک بردار بلندتر پردازنده گرافیکی

ما توصیه می‌کنیم که شما بیش‌ازحد به این محاسبات چندان دقیق دل نبندید. این در اصل فقط یک تابع دوبعدی زمان است که باعث به وجود آمدن یک اثر موجدار خوب در هنگام انیمیشن‌سازی می‌شود. تصویر یک فریم باید چیزی شبیه به شکل (۳ـ۳) باشد [۲۰].



شکل(۳-۱۲) تصویری از یک مثال موج پردازنده گرافیکی [۲۰]

حافظه اشتراکی و هم‌زمان‌سازی تاکنون، انگیزه تقسیم بلوک‌ها به رشته‌ها، صرفاً یکی از محدودیت‌های سخت‌افزاری بشمار آمده است که برای تعدادی از بلوک‌ها در هنگام پیشروی کار وجود داشته است. این‌یک انگیزه نسبتاً ضعیف است زیرا این کار می‌تواند به‌راحتی به‌وسیله زمان

اجرای کودا در پشت‌صحنه‌ها انجام شود. خوشبختانه، دلیل دیگری هم وجود دارند که ممکن است یک فرد را راغب به تقسیم یک بلوک به رشته‌ها کند. کودا C منطقه‌ای از حافظه را در دسترس قرار می‌دهد که ما به آن حافظه مشترک (دینامیک) میگوییم. این منطقه از حافظه علاوه بر آن، پسوند دیگری به زبان C را به ارمغان می‌آورد که مشابه __device__ و __ global_ است. به‌عنوان یک برنامه‌نویس، شما می‌توانید اعلان‌های متغیر خود را با کلمه کلیدی کودا C یعنی مشترک به‌گونه‌ای تغییر دهید که این متغیر در حافظه مشترک جای بگیرد. اما مکان موردنظر کجاست؟

کامپایلر کودا C در قبال متغیرهای موجود در حافظه مشترک رفتاری متفاوت از متغیرهای معمولی را بروز می‌دهد و یک کپی از متغیر را برای هر بلوک راه‌اندازی شده در پردازنده گرافیکی فراهم می‌آورد. هر رشته در آن بلوک به اشتراک حافظه مشغول می‌شود اما رشته‌ها قادر به دیدن یا تغییر کپی این متغیر دیده‌شده در داخل بلوک‌های دیگر نیستند. این فراهم‌کننده یک ابزار عالی است که توسط آن، رشته‌های موجود در یک بلوک می‌توانند با یکدیگر در امر انجام محاسبات ارتباط و همکاری به عمل بیاورند. علاوه بر این، بافرهای حافظه مشترک بجای اقامت در DRAM اف-تراشه ، به‌صورت فیزیکی بر روی پردازنده گرافیکی مقیم می‌شوند. به همین دلیل، زمان تأخیر دسترسی به حافظه مشترک به‌مراتب کمتر از بافرهای معمولی است به‌طوری‌که نقش حافظه مشترک به‌عنوان یک کش یا دفتر یادداشت تحت مدیریت نرم‌افزار در هر بلوک را پررنگ‌تر کرده است. چشم‌انداز ارتباط بین رشته‌ها باید شمارا نیز همانند ما بسیار هیجان‌زده کند. اما هیچ‌چیز در زندگی بی‌بها نیست و ارتباط بین‌رشته‌ای نیز از این قاعده مستثنا نیست. اگر ما انتظار برقراری ارتباط بین رشته‌ها را داریم مسلماً به مکانیزمی جهت هماهنگ‌سازی رشته‌ها نیز نیازمندیم. به‌عنوان‌مثال، اگر رشته A مقداری را برای حافظه مشترک می‌نویسد و ما از رشته Bمی‌خواهیم که با این مقدار کاری انجام دهد، نمی‌توانیم تا زمان تکمیل کار نوشتن از روی رشته A، رشته B را وادار به شروع کار خود کنیم. ما بدون هم‌زمان‌سازی باعث ایجاد شرایط مسابقه‌ای شده‌ایم که در آن صحت نتایج اجرایی به جزئیات غیرقطعی سخت‌افزاری بستگی دارد. بیایید به نمونه‌ای که از این ویژگی‌ها استفاده می‌کند، نگاهی بیندازیم [۲۰].

۳-۳-۲-٤ ضرب نقطه‌ای

حال حاضر باید نگاهی هم به ضرب نقطه‌ای بیندازیم (که گاهی اوقات به آن ضرب داخلی هم گفته می‌شود). ما ابتدا در مورد اینکه اصل ضرب نقطه‌ای چیست آن‌هم فقط در مواردی که شما با ریاضیات برداری ناآشنا هستید و یا جزء مسائلی است که اخیراً به آن پرداخته‌شده است دست به یک بررسی اجمالی می‌زنیم. محاسبه از دو مرحله تشکیل‌شده است. در مرحله اول، عناصر متناظر دو بردار

ورودی را ضرب می‌کنیم. این بسیار به جمع برداری شبیه است اما بجای جمع از ضرب بهره می‌جوییم. بااین‌حال بجای ذخیره‌سازی بعدی این مقادیر در یک بردار خروجی سوم، می‌توانیم همه آن‌ها را به‌منظور تولید یک خروجی عددی واحد جمع کنیم. به‌عنوان‌مثال، اگر ضرب نقطه‌ای دو بردار چهارعنصر را صورت دهیم به فرمول (۳-۲) خواهیم رسید.

$$(x_1, x_2, x_3, x_4).(y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4 \qquad \text{(فرمول ۳-۲)}$$

شاید الگوریتمی که ما تمایل به استفاده از آن را داریم در حال آشکار شدن باشد. ما می‌توانیم اولین گام را دقیقاً به همان صورتی برداریم که در جمع برداری برداشتیم. هر رشته یک جفت از ورودی‌های متناظرها ضرب می‌کند و سپس به‌سوی جفت بعدی خود حرکت می‌کند. ازآنجاکه نتیجه باید جمع تمامی این ضربه‌ای جفتی باشد. هر رشته یک جمع جاری (sum running) از جفت‌های افزوده‌شده توسط خود را نگهداری می‌کند. درست مثل مثال جمع، رشته‌ها برای اطمینان از حفظ و از دست ندادن هر عنصر و جلوگیری از ضرب مجدد یک جفت به افزایش شاخص‌هایشان از طریق تعداد کل رشته‌ها می‌پردازند. این اولین مرحله از روال معمول ضرب نقطه‌ای است:

```
#include
#define imin(a,b) (a<b?a:b)
const int N = 33* 1024;
const int threads PerBlock = 256;
_global void dot ( float *a, float *b, float *c) {
_shared_float cache (threadsPerBlock);
int tid - threadIdx.x + blockIdx .x * blockdim.x;
int cacheIndex = threadIdx.x;
float temp = 0; while (tid < N) {
temp += a[tid] *b[tid]; tid += blockDim.x * gridDim.x;
// set the ca che values cache
 [cacheIndex] = temp;
```

شکل (۳-۱۴) ضرب مجدد یک جفت به افزایش شاخص‌هایشان از طریق تعداد کل رشته‌ها

همان‌طور که می‌بینید ما یک بافر حافظه مشترک به نام حافظه موقت (cache) را معرفی کرده‌ایم. این بافر برای ذخیره جمع جاری هر رشته استفاده می‌شود. به‌زودی به دلیل انجام این کار پی می‌برید اما فعلاً فقط به بررسی مکانیک‌هایی می‌پردازیم که با آن می‌توانیم کار را تکمیل کنیم. شناسایی و

اعلام یک متغیر برای اقامت در حافظه مشترک از اهمیت چندانی برخوردار نیست و بیشتر به شیوه‌ای شباهت و بستگی دارد که شما به‌وسیله آن‌یک متغیر را در استاندارد C به‌عنوان استاتیک و یا فرار اعلام می‌کنید:

```
int tid = threadIdx.x + blockIdx .x * blockdim.x;
int cacheIndex = threadIdx .x;
```

شکل (۳-۱۵) استفاده از بافر برای ذخیره جمع جاری هر رشته

ما آرایه اندازه ThreadsPerBlock[1] را اعلام می‌کنیم بنابراین هر رشته در بلوک دارای یک محل برای ذخیره نتیجه موقت خود هست. به خاطر بیاورید که وقتی ما حافظه را به‌صورت جهانی اختصاص می‌دهیم به‌اندازه کافی برای هر رشته اجراکننده هسته یا ThreadsPerBlock که تعداد کل بلوک‌ها را زمان‌بندی می‌کند حافظه اختصاص می‌دهیم. اما ازآنجاکه کامپایلر یک کپی از متغیرهای مشترک را برای هر بلوک ایجاد می‌کند، تنها به تخصیص کافی حافظه نیاز است به‌نحوی‌که هر رشته در بلوک از یک ورودی برخوردار باشد. پس از تخصیص حافظه مشترک درست مثل گذشته نوبت به محاسبه شاخص داده‌ها می‌رسد:

```
// set the ca che values
 cache [cacheIndex] = temp;
```

شکل (۳-۱۶)

حال دیگر باید محاسبات سه‌بعدی متغیر آشنا به نظر برسند. ما تنها شاخص‌های بلوکی و رشته‌ای را برای انتقال افست جهانی به آرایه‌های ورودی با یکدیگر ترکیب می‌کنیم. افست انتقال‌یافته به حافظه کش مشترک صرفاً به‌عنوان شاخص رشته‌ای عمل می‌کند و دیگر به‌جای دادن شاخص بلوکی در این افست نیازی نیست زیرا هر بلوک از کپی اختصاصی حافظه مشترک خود برخوردار هست. درنهایت ما به روشن‌سازی بافر حافظه مشترک می‌پردازیم به‌طوری‌که بعداً نگرانی در مورد اینکه آیا یک ورودی خاص دارای داده‌های معتبر ذخیره‌شده در اینجا هست یا خیر، داشته باشیم، حتی قادر به جمع کورکورانه کل آرایه‌ها باشیم.

اگر اندازه بردارهای ورودی، مضربی از تعداد رشته‌ها در هر بلوک نباشد ممکن است که همه ورودی‌ها مورداستفاده قرار نگیرند. در این حالت، آخرین بلوک شامل برخی رشته‌ها است که هیچ کاری انجام نمی‌دهند و بنابراین مقادیر را نمی‌نویسند.

[1] In general you want to size your blocks/grid to match your data and simultaneously maximize occupancy

هر رشته یک جمع جاری حاصل‌ضرب ورودی‌های متناظر را در a و b محاسبه می‌کند. پس از رسیدن به انتهای آرایه، هر رشته به ذخیره جمع موقت خود در بافر مشترک مشغول می‌شود.

```
float temp = 0;
while (tid < N) {
temp += a [tid] *b[tid]; tid += blockDim.x * gridDim.x;
// set the ca che values
cache (cacheIndex] = temp;
```

شکل (۳-۱۷) ذخیره جمع موقت هر رشته در بافر مشترک

در این نقطه از الگوریتم به جمع تمامی مقادیر موقت جای‌داده شده در کش نیاز است. برای انجام این کار نیز به برخی از رشته‌ها برای خواندن مقادیر ذخیره‌شده احتیاج است. بااین‌حال همان‌طور که اشاره شد، این‌یک عمل خطرناک است. ما به روشی نیازمندیم که همه نوشته‌های انجام‌شده در کش آرایه مشترک [ ] را پیش از اینکه کسی برای خواندن آن‌ها تلاش کند به‌طور کامل تضمین کند. خوشبختانه چنین روشی وجود دارد.

این فراخوانی تضمین می‌کند که هر رشته در بلوک، دستورهای قبل از __ syncthreads () را پیش از اجرای دستور بعدی توسط سخت‌افزار بر روی هر رشته به اتمام می‌رساند. این دقیقاً همان چیزی است که ما به آن نیاز داریم ما اکنون می‌دانیم که وقتی رشته اول پس از __ syncthreads () به اجرای دستور اول می‌پردازد هر رشته دیگر در بلوک نیز به پایان اجرای خود تا مرز __ syncthreads () می‌رسد. حال که از پر شدن کش موقت اطمینان حاصل کرده‌ایم، می‌توانیم مقادیر مربوطه را در آن جمع بزنیم. ما روند کلی گرفتن یک آرایه ورودی و انجام بعضی از محاسبات تولیدکننده آرایه کوچک‌تر نتایج یک کاهش را فراخوانی می‌کنیم. کاهش‌ها اغلب در محاسبه موازی که به نام‌گذاری آن‌ها منجر می‌شود، به وجود می‌آیند. ساده‌ترین راه برای تکمیل این کاهش، تکرار یک‌رشته در حافظه مشترک و محاسبه جمع جاری است. این امر متناسب با طول آرایه زمان‌بر است. بااین‌حال، ازاینجایی‌که ما دارای صدها رشته در دسترس برای انجام کارهایمان هستیم، انجام کاهش به‌طور موازی صورت می‌پذیرد و به زمانی متناسب با لگاریتم طول آرایه هم نیاز است. در ابتدا، کد زیر به‌صورت حلقوی به نظر می‌رسد و ما در یک‌لحظه به شکستن آن مبادرت می‌ورزیم. ایده کلی این است که هر رشته دو مقدار را در کش [ ] اضافه می‌کند و نتیجه را به‌صورت ذخیره‌شده به کش [ ] بازمی‌گرداند. ازآنجاکه هر رشته دو ورودی را به یکی ترکیب می‌کند، این مرحله با نصف تعدادی که کار با آن آغازشده به پایان می‌رسد. در مرحله بعد، ما همان کار را در نیمه‌ی باقیمانده انجام

می‌دهیم. ما به همین طریق برای مراحل (threadsPerBlock) log۲ ادامه می‌دهیم تا زمانی که جمع هر ورودی در کش [ ] حاصل آید. برای مثال، در هر بلوک از ۲۵۲ رشته استفاده می‌شود پس این روند باید ۱ تکرار داشته باشد تا ۲۵۲ ورودی در کش [ ] به جمعی واحد کاهش یابد [۲۰].

۳-۳-۲-۵ بیت مپ۱ حافظه مشترک۲

ما به نمونه‌هایی بازمی‌گردیم که از حافظه مشترک استفاده می‌کنند و برای اطمینان از آماده بودن داده‌ها پیش از ادامه کار، () __ syncthreads را بکار می‌بندند. وقتی حرف از سرعت می‌شود ممکن است به انجام یک کار خطرناک وسوسه شوید و () __ syncthreads را حذف کنید. در حال حاضر به یک مثال گرافیکی نیازمند به() __ syncthreads برای دانستن میزان صحت، رجوع می‌کنیم. ما به شما تصاویری هرچند اندک از خروجی درنظر گرفته‌شده می‌دهیم و خروجی را درزمانی که بدون () __ syncthreads قابل‌اجرا هستند نشان می‌دهیم.

بدنه‌ی ()main با مثال مجموعه ژولیای پردازنده گرافیکی یکسان است، هرچند که در این زمان رشته‌های متعدد در هر بلوک راه‌اندازی می‌شوند:

```
#include "cuda.h" #include #include "../common/cpu_bitmap.h"
#define DIM 1024 #define PI 3.1415926535897932f
int main(void) {
CPUBitmap bitmap ( DIM, DIM ); unsigned char *dev_bitmap;
HANDLE_ERROR( cudaMalloc( (void**) &dev_bitmap,
bitmap.image_size () ) );
dim 3 grids (DIM/16, DIM/16);
dim 3 threads (16,16);
kernele <<<rids, threads>>> ( dev_bitmap);
HANDLE_ERROR( cudaMemcpy (bitmap.get_ptr(0, dev_bitmap.
bitmap.image_size (),cudaMemcpyDeviceToHost ) );
bitmap.display_and_exit();
cudaFree ( dev_bitmap );
```

شکل (۳-۱۸) استفاده از حافظه مشترک با مجموعه ژولیای پردازنده گرافیکی

۱ For image processing applications you would acces the **bitmap's** memory, copy this to the **کودا** device

۲ shared memory

همان‌طور که در مثال مجموعه ژولیا نشان داده شد، هر رشته یک مقدار پیکسلی را برای محل یک خروجی واحد محاسبه خواهد کرد. اولین چیزی که هر رشته انجام می‌دهد محاسبه محل X و Y خود در تصویر خروجی است. این محاسبه با محاسبه سه‌بعدی در مثال جمع برداری یکسان است، اگرچه که این بار ما آن را به‌صورت دوبعدی محاسبه می کنیم:

```
_global void kernel ( unsigned char *ptr) {
// map from threadIdx/blockIdx to pixel position
 Int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y + blockDim.y;
int offset = x + y + blockDim.x * gridDim.x;
```

از آنجایی که ما از یک بافر حافظه مشترک برای کم کردن محاسباتمان استفاده می کنیم، باید گفت که هر رشته در پلوت ۱۶ ۴۱۶ ما دارای یک ورودی است.

```
shared | float shared [16] [16];
```

سپس هر رشته یک مقدار را برای ذخیره شدن در این بافر محاسبه می کند

```
// now calculate the value at that position
const float period = 128.0f;
shared threadIdx.x) threadIdx,y) =255 + (sinf (x* 2.0f* PI / period) + 1.0f) +
(sinf ( 2.0f* PI/ period) + 1.0f) / 4.0;
```

شکل (۳-۱۹) جمع برداری یکسان

و دست‌آخر، این مقادیر برگردانده شده به پیکسل ذخیره می‌شوند و ترتیب x و y معکوس می‌شود:

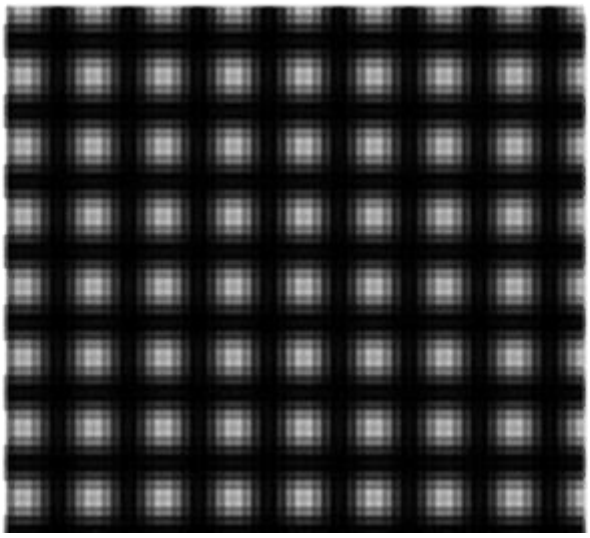
```
ptr [offset *4 + 0] = 0;
ptr [ffset *4 + 1]= shared [15-threadIdx.x] [15-threadı dx.y]; ptr (offset *4 + 2) = 0;
ptr (offset*4 + 3) = 255;
```

شکل (۳-۲۰) مقادیر برگردانده شده به پیکسل ذخیره می‌شوند و ترتیب x و y معکوس می‌شود

مسلماً، این محاسبات تا حدودی خودسرانه هستند. ما صرفاً با چیزهایی روبرو می‌شویم که شبکه‌ای از حباب‌های کروی سبز را ترسیم می‌کنند. بنابراین پس از کامپایلر و اجرای این کرنل، تصویری شبیه

به آنچه در شکل (۳-۷) نشان داده‌شده را به‌عنوان خروجی خواهیم داشت. حال در اینجا چه اتفاقی می‌افتد؟ ممکن است در حین بروز حدس‌هایی در ذهن شما از نحوه تنظیم مثال، یک نقطه هم‌زمان‌سازی مهم از دست برود. وقتی یک‌رشته اقدام به ذخیره مقدار محاسبه‌شده به پیکسل آن‌هم به‌صورت مشترک می‌کند، این احتمال وجود دارد که رشته مسئول نوشتن مقدار مشترک هنوز کار نوشتن را تمام نکرده باشد. تنها راه تضمین اتفاق نیفتادن این مسئله، استفاده از __ syncthreads ()است. بنابراین، نتیجه یک تصویر خراب از حباب‌های سبز هست. هرچند که این پایان جهان است ولی ممکن است نرم‌افزار، مقادیر مهم‌تری را موردمحاسبه قرار دهد. در عوض، ما به‌اضافه کردن یک نقطه هم‌زمان‌سازی بین نوشتن برای حافظه مشترک و خواندن متعاقب آن نیاز داریم [۲۰].

```
shared .th eadi .* th: 2e adIdx,y) = 255 = (sinf x* 2.25*?I/ period) + 2.25) = (sinf (y*
2,045*I/ eriod) + 1.5) / 4.0;
--Syncthreads ();
to set = +1 = 0;
pt = (of !set = 4 + 1 = shared i5- threadIdx . * y - 615،thiye adIdx );
t = [of !set = +2 = 0; ptr to set *4 + 3) = 255;
```



شکل (۳-۲۱) تصویر بعد از اضافه کردن هم‌زمان‌سازی مناسب [۲۰]

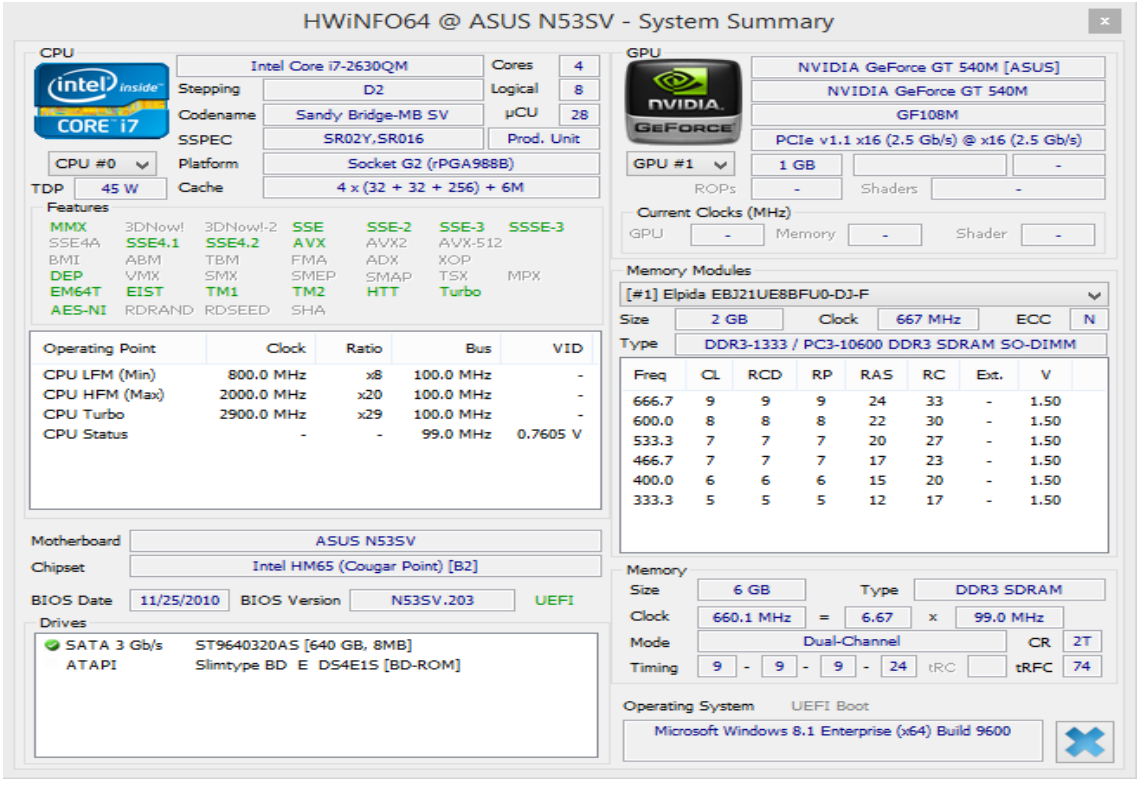
فصل چهارم:
نتایج (یافته های تحقیق)

٤-١ مقدمه

در این بخش نتایج آزمایش‌هایی را ارائه می‌دهیم که هدف آن‌ها ارزیابی عملکرد الگوریتم NLM بوده است در ارتباط با این هدف ما تصاویر MR نمونه را قبل و بعد از حذف نویز مورداستفاده قرار می‌دهیم. ولی این تنها هدف این مقاله برای ارزیابی دقت NLM تحت تنظیمات متعدد (یعنی نویز تطبیقی، تابع شدت نویز) نیست، بلکه ما روی زمان محاسباتی عملکرد پردازنده گرافیکی و افزایش سرعت آن نسبت به عملکرد پردازنده اصلی تأکید خواهیم داشت.

٤-٢ مشخصات سیستم و کارت گرافیک

جهت پیاده‌سازی و شبیه‌سازی از سیستمی با مادربرد مدل N٥٣S ASUS و قدرت پردازنده Intel iV Coreبا میزان کش ۶ مگابایت که در نوع خود بالاترین مدل از پردازنده‌های چندهسته‌ای تولیدشده توسط کمپانی Intel است و کارت گرافیکی NVIDIA GT GEFORCE ۵۴۰ M که دارای حافظه ۱۶ گیگابایت رم DDR۳ است به همراه سیستم‌عامل Microsoft Windows ٨.١ Enterprise (x۶۴) Build ۹۶۰۰ بهره گرفته‌شده است. در شکل (۴-۱) مشخصات سخت‌افزاری سیستم مورداستفاده نشان داده‌شده است.



شکل (۴-۱) مشخصات سخت‌افزاری سیستم مورداستفاده

٤-٣ ابزار شبیه‌سازی

برای شبیه‌سازی و پیاده‌سازی این الگوریتم در پردازنده پردازنده گرافیکی از نرم‌افزار Microsoft Visual Studio Ultimate ۲۰۱۰ همراه با ان ویدیا کودا Toolkit ۷.۰ استفاده‌شده است.

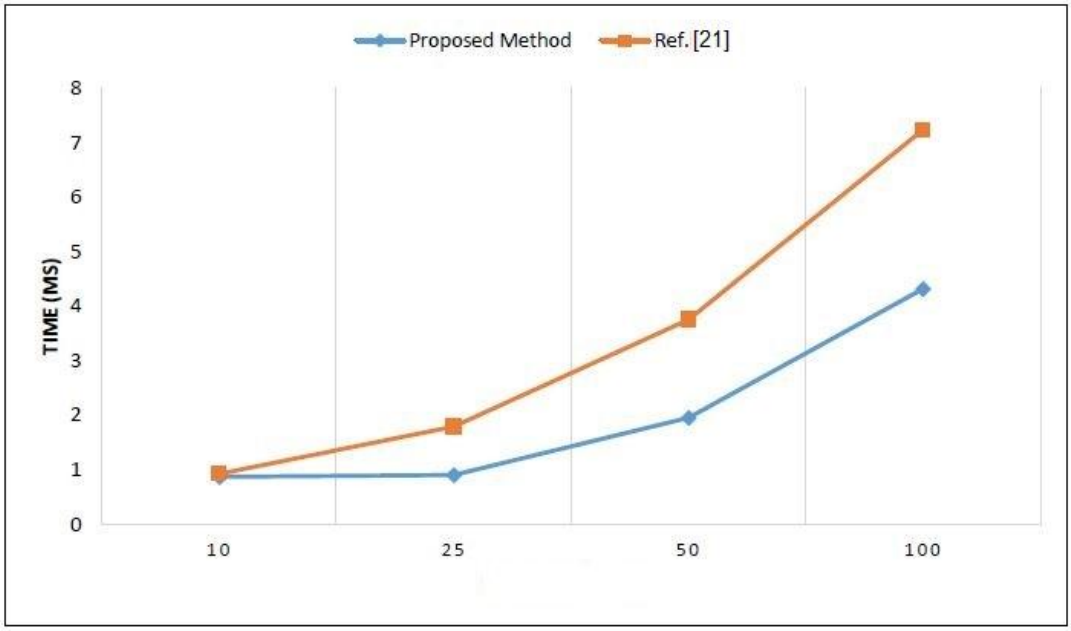
٤-٤ نحوه پیاده‌سازی و شبیه‌سازی الگوریتمNLM

در این قسمت برنامه با Visual C++ تدارک دیده‌شده است تا الگوریتم NLM را با استفاده از امکانات موازی‌سازی کودا و کاربرد مفاهیم حذف نویز حل کند. این برنامه دارای این قابلیت است که داده‌های ورودی خود را به شکل مجموعه داده از تصویر bitmap شده کاربر می‌گیرد.

به‌عنوان تصاویر نمونه، ما دو آزمایش متفاوت را موردتوجه قراردادیم. در آزمایش اول یک Phantomag مورداستفاده قرار گرفت. سه مقوله متفاوت دیگر در این بررسی مورداستفاده قرار گرفت که T1w, T2w and PDw می‌باشند. مدل اول دارای پیکسل‌های (۲۵۶*۲۵۶*۱۵۰) ایزوتروپیک نزدیک (۰.۹۳۷۵*۰.۹۳۷۵*۱) mm سیگنال‌های شبیه‌سازی‌شده به شکل ((۵۱۰/۱۵ms TR/TE) T1w و PDW، (۱۵۹۰ms TR/TE) /۱۸۶۷) T2w قسمت‌های محوری با ضمانت ۴ میلی‌متر برای رتیب اسپین اکو۱ معمول و ۱ میلی‌متر TR=۹.۹ms، زاویه فلیب=۱۰ درجه، TE=۳.۵ms برای ترتیب ۳D T1w FFE ارائه شد. سپس ما ۳۷ قسمت ۴ میلی‌متر را مورداستفاده قراردادیم.

درواقع اجرای الگوریتم حذف نویز با تصویر معمولی فاحشی در زمان اجرا ایجاد نمی‌کند، اما تصویر دارای پیکسل‌های (۲۵۶*۲۵۶*۱۵۰) ایزوتروپیک نزدیک (۰.۹۳۷۵*۰.۹۳۷۵*۱) ، اختلاف‌زمان بین این دو واحد بیشتر شده و پردازنده گرافیکی موفق‌تر عمل می‌نماید.

[۱] spin echo



شکل (۴-۲) مقایسه زمان اجرای الگوریتم NLM روش پیشنهادی و روش مرجع [۲۱]

برای رسیدن به درک بهتری در رابطه با عملکرد واحد پردازش گرافیکی، روش پیشنهادی با روش به کار گرفته‌شده در مرجع [۲۱] مقایسه شد. همان‌طور که در شکل (۴-۲) مشخص است، در تصویر معمولی تفاوت چندانی بین زمان اجرای روش پیشنهادی و الگوریتم پیاده‌سازی شده در مرجع [۲۱] وجود ندارد اما رفته‌رفته با بزرگ‌تر شدن ابعاد تصویر اختلاف‌زمان بیشتر شده و روش پیشنهادی عملکرد بهتری از خود نشان می‌دهد.

آزمایش دوم بررسی موضوعی واقعی است که از قسمت‌های محوری TLW، T۲ و PDW تشکیل می‌شود که کل حجم مغز را پوشش می‌دهد با استفاده از اسکنر Philips Intera ۱.۵T MR به‌دست‌آمده که دارای ترتیب اسپین اکو معمول دوبعدی[۱] با (۵۱۰/۱۵ ms TR/TE) T۱w و PDW، T۲w (۱۸۶۷/۱۵–۹۰ ms TR/TE) با رزولوشن in-plane با دقت mm ۰.۹۳۷۵* ۰.۹۳۷۵ (ماتریس جذب۲۵۶*۲۵۶) و ضخامت ۱.۲۰ میلی‌متر می‌یابد. مجموعه داده‌ای حقیقتی دارای ۱۳۰*۱۵۷*۱۸۹ پیکسل است.

جدول‌های (۴-۱، ۴-۲، ۴-۳ و ۴-۴) زمان اجرا پردازنده گرافیکی و سرعت گرفتن متناسب تحت تنظیمات متفاوت در حجم دوبعدی[۲] یا سه‌بعدی[۳] اندازه بلوک d (۱ یا ۲) و پنجره جستجو v (۳ یا ۵) را گزارش می‌دهند. برای هر مورد ۸ آزمایش در مدل‌های نویز متفاوت (گاوسی،Rician، یکنواخت

[۱] ۲D
[۲] ۲D
[۳] ۳D

فضایی، ازلحاظ فضایی تطبیقی با پنجره واریانس مساوی با (۱ یا ۲) انجام می‌گردید. دامنه زمان‌های محاسباتی پردازنده اصلی و پردازنده گرافیکی و سرعت‌های متناظر نشان داده می‌شود.

به‌طور خاص، ستون‌های هفتم و پنجم نشان می‌دهند که تا چه اندازه ورژن‌های موازی از پردازنده اصلی سریع‌تر می‌باشند. به دلایل عملیاتی مربوط به‌اندازه حافظه مشترک موجود ما اندازه آپرون‌ها[۱] avفیلتر الگوریتم مقوله‌ای مربوط می‌شوند. جدول‌های(۴-۳و ۴-۴) به ورژن الگوریتم سه مقوله‌ای (PDwوT۱w, T۲w) مربوط می‌باشند.

در ارتباط با یک مقوله‌ای از جدول (۴-۱) مجموعه داده (phontomg) فاکتورهایی افزایش سرعت حداقل به‌دست‌آمده با یک پردازنده گرافیکی و دو پردازنده گرافیکی به ترتیب *۵۰۳ و *۷۵۸ بالغ گردید. زیرمجموعه داده‌ای حقیقی (جدول ۴-۲) مقادیر به ترتیب به *۳۴۰ و *۵۵۱ کاهش یافت این فرآیند به خاطر اورهد[۲] زیادتر ناشی از فرآیندهای اتلاف زمانی می‌شود همان‌طور که شکافت داده‌ای، تخصیص دهی حافظه کرنل، انتقال داده‌ای بین کرنل و حافظه موضعی[۳] درخواست از کرنل و جمع‌آوری داده‌ها مطرح است، این اورهد حافظه وقتی با اهمیت می‌شود که تصاویر با اندازه کوچک‌تر پردازش می‌شود.

جدول (۴-۱) فاکتورهایی افزایش سرعت حداقل به‌دست‌آمده مجموعه داده (phontomg) با یک پردازنده گرافیکی و دو پردازنده گرافیکی

Vol.	d	v	CPU Time	2 GPUs Time	1 GPU Time	2 GPUs Speed-up	1 GPU Speed-up
3D	2	5	5080-6628	3.8-4	7.6-7.9	3119-1686	659-847
3D	1	5	1561-1624	1.3-1.4	2.6-2.7	1174-1243	599-619
3D	2	3	2241-2870	1.4-1.5	3.3-3.4	1590-1984	677-845
3D	1	3	515-545	0.36-0.42	0.64-0.69	1283-1447	781-819
2D	2	5	174-282	0.14-0.29	0.28-0.55	1005-1271	519-640
2D	1	5	70-145	0.067-0.14	0.13-0.26	931-1335	503-690
2D	2	3	71-141	0.051-0.11	0.12-0.23	1082-1745	507-778
2D	1	3	29-61	0.028-0.062	0.043-0.084	758-1417	567-896

جدول (۴-۲) الگوریتم NLM تک مقوای در مجموعه داده‌ای حقیقتی (با اندازه ۱۳۰*۱۵۷*۱۳۹) برای پیکربندی‌های حجم متفاوت، (۳D،۲D، ستون ۱)، (d=۱,۲ ستون ۲)، (v=۳,۵ ستون ۳). دامنه

[۱] aprons
[۲] overhead
[۳] local memory

زمان‌های پردازنده اصلی، پردازنده گرافیکیs ۲، پردازنده گرافیکی ۱ (به ثانیه به ترتیب ستون‌های۶-۴) و افزایش سرعت متناظر (ستون‌های ۷و۸) برای مدل‌های نویز متفاوت را نشان می‌دهد.

جدول (۴-۲) عملکرد الگوریتم NLM تک مقوله در مجموعه داده‌ای حقیقتی

Vol.	d	v	CPU Time	۲ GPUs Time	۱ GPU Time	۲ GPUs Speed-up	۱ GPU Speed-up
۳D	۲	٥	۸۸۲۹-۹٥۲۱	۷.۹-۸.۳	۱۹-۲۰	۱۰۹۱-۱۱٦٤	٤٤٦-٤۷۲
۳D	۱	٥	۲٦٦۷-۲۷٥٥	۲.۷-۲.۹	٥.۳-٥.٦	۹٦٤-۱۰۲۱	٤۹۳-٤۷۲
۳D	۲	۳	۳٦۹٥-٤۷۲۱	۳-۳.۲	٦-٦.۳	۱۱۸۹-۱٤۸۲	٦۰٤-۷٥٦
۳D	۱	۳	۸٥۲-۸۸٤	۰.۸-۱	۱.۳-۱.٥	۹٤٤-۱۰۸۲	٦۲٦-٦٦٤
۲D	۲	٥	۲۷۸-٤٤۸	۰.۲۸-۰.٥۸	۰.٦٥-۱.۳	۷۸۱-۱۰۱۹	۳٤٥-٤۳۰
۲D	۱	٥	۱۲۱-۲۲٦	۰.۱٥-۰.۳۱	۰.۲۷-۰.٥٤	٦۰۱-۱۰٤٤	۳٤۰-٥٦۲
۲D	۲	۳	۱۱۳-۲۳۱	۰.۱۲-۰.۲٥	۰.۲۲-۰.٤٤	۷٥٥-۱۲۸۲	٤۲۲-٦۸۹
۲D	۱	۳	٤٦-۹۷	۰.۰٦٥-۰.۲	۰.۰۸۹-۰.۲	٥٥۱-۹٤۸	٥۰۷-٦۹۷

جدول (۴-۳) عملکرد الگوریتم چند مقوله‌ای NLM در مجموعه داده‌ای (۳۷*۲۵۶*۲۵۶) برای پیکربندی‌های حجم متفاوت (۲D و ۳D) (دوبعدی و سه‌بعدی) ، (ستون ۱) ۲و۱=d، (ستون ۲)، ۵و۳=v (ستون ۳) دامنه زمان‌های پردازنده اصلی،پردازنده گرافیکی S ۲، پردازنده گرافیکی ۱ به ترتیب (ستون‌های ۶-۴) و افزایش سرعت متناظر (ستون‌های ۷ و ۸ برای نویزهای متفاوت را نشان می‌دهد.

جدول (۴-۳) عملکرد الگوریتم چند مقوله‌ای NLM در مجموعه داده‌ای (۳۷*۲۵۶*۲۵۶)

Vol.	d	v	CPU Time	۲ GPUs Time	۱ GPU Time	۲ GPUs Speed-up	۱ GPU Speed-up
۳D	۲	۵	۱۰.۸٤٦-۱۱.۳۲۵	٤۲-٤۳	٦۳-٦۵	۲۵۵-۲۷۰	۱۷۰-۱۷۹
۳D	۱	۵	۳۲۲۱-٤۱۲۷	۱٤-۱٤	۲۰-۲۱	۲۲۹-۲۹۰	۱۵۳-۱۹۳
۳D	۲	۳	٤۷۳۳-۵٦۹۱	۱۷-۱۸	٦.۱-٦.٦	۲٦٦-۳۲۷	۱۸۸-۲۳۰
۳D	۱	۳	۱۰۷۵-۱۱۰۲	٤.۳-٤.۸	۲.۲-٤	۲۳۱-۲۵۲	۱٦۷-۱۷۸
۲D	۲	۵	۳۵۹-۵۵۳	۱.۵-۲.۵	۱-۱.۹	۲۲۰-۲٤۹	۱٤۰-۱٦٦
۲D	۱	۵	۱۳۷-۲۷۵	۰.٦۳-۱.۲	۰.۸۲-۱.٦	۱۸۷-۲۷۸	۱۱٦-۱۸۸
۲D	۲	۳	۱٤٦-۲۲٦	۰.۵۹-۱.۱	۰.۸۲-۱.٦	۲۱۸-۳۰۵	۱٤٦-۲۱۸
۲D	۱	۳	۵٦-۱۱۵	۰.۲۷-۰.۵۲	۰.۳۷-۰.۷۳	۱۷۲-۲۷۸	۱۲۱-۲۰۸

جدول (۴-۴) عملکرد الگوریتم NLM چند مقوای در مجموعه داده‌ای حقیقتی (اندازه ۱۳۰*۱۵۷*۱۸۹) برای پیکربندی‌های حجم متفاوت (۲D,۳D) (ستون ۱) ۲و۱=d (ستون ۲)، ۵و۳=v (ستون ۳) دامنه زمان‌های پردازنده اصلی،پردازنده گرافیکی S ۲، پردازنده گرافیکی ۱ به ترتیب (ستون‌های ۶-۴) و افزایش سرعت متناظر (ستون‌های ۷ و ۸) برای مدل‌های نویزهای متفاوت را نشان می‌دهد.

جدول (۴-۴) عملکرد الگوریتم NLM چند مقوای در مجموعه داده‌ای حقیقتی (اندازه ۱۳۰*۱۵۷*۱۸۹)

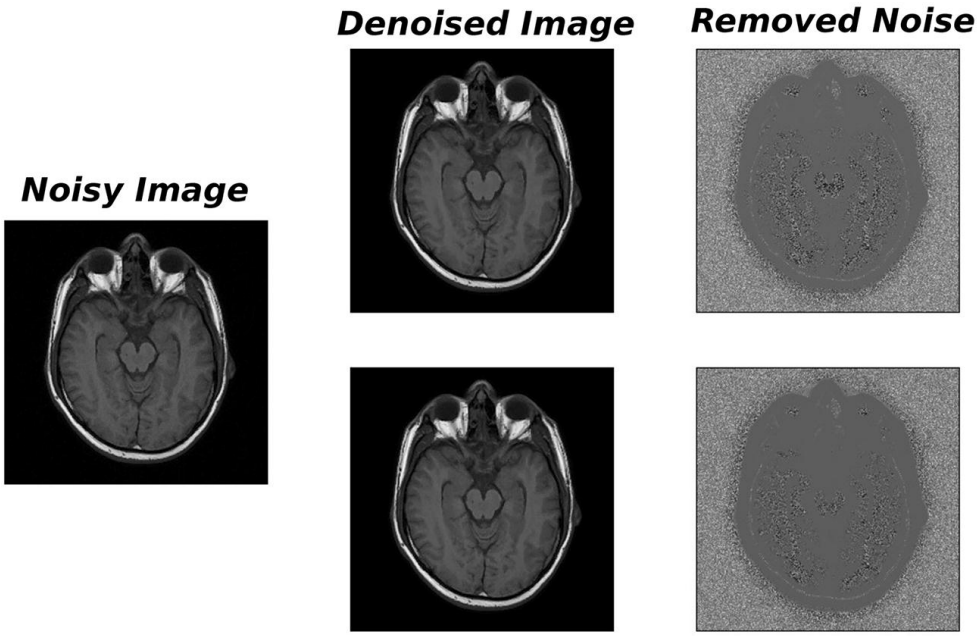
Vol.	d	v	CPU Time	۲ GPUs Time	۱ GPU Time	۲ GPUs Speed-up	۱ GPU Speed-up
۳D	۲	۵	۱۸.۵۳٦-۱۸.۹۵٤	۱۰۸-۱۱۱	۱٦۲-۱٦٦	۱۷۰-۱۷۳	۱۱٤-۱۱۵
۳D	۱	۵	۵۳۹۳-٦۰۷۷	۳٤-۳٦	۵۱-۵۳	۱۵۳-۱۷۵	۱۰۳-۱۱٦
۳D	۲	۳	۷٦٦۸-۸۸۲٦	۳۷-۳۹	۵٦-۵۸	۲۰۱-۲۳٦	۱۳٤-۱۵٦
۳D	۱	۳	۱۷۳۵-۱۷۸٦	۹.۳-۱۰	۱٤-۱۵	۱۷۳-۱۸۹	۱۲۰-۱۲۷
۲D	۲	۵	۵٦۸-۸۷۲	۳.۵-٦	۵.۲-۸.۸	۱٤٤-۱۷٤	۹۸-۱۱٦
۲D	۱	۵	۲۱۸-٤۲۱	۱.۵-۲.۸	۲.۲-٤	۱۱۹-۱۸۷	۸۵-۱۲۷
۲D	۲	۳	۲۳۰-۳۵٤	۱.۳-۲.۳	۱.۸-۳.۲	۱۵۷-۲۲۸	۱۱۱-۱۵۸
۲D	۱	۳	۸۸-۱۷۷	۰.۵۹-۱.۲	۰.۸-۱.۵	۱۲۱-۲۰۲	۹۳-۱۵۱

متوسط افزایش سرعت برای مجموعه داده‌ای (Phantomag) ۱۳۰۲X برای پردازنده گرافیکی s ۲ و ۶۷۰X برابر برای پردازنده گرافیکی ۱ است. که فاکتور ۱.۹۴ در نزدیک به مقدار ایده آل ۲ است. این فاکتور تقریباً برای مجموعه داده‌ای حقیقی۱ ۱.۸۶ یکسان است.

در ارتباط با سه مقوله (جدول‌های ۴-۳ و۴-۴) فاکتورهای حداقلی افزایش سرعت برای پردازنده گرافیکی ۱ و پردازنده گرافیکی 2S به تربیت در مجموعه داده‌ای Phantomag *۱۷۶ و ۱۱۶ است. در مجموعه داده‌ای حقیقی، مقادیر به ترتیب *۸۵ و *۱۱۹ می‌شود. مجموعه داده‌ای Phantomag متوسط افزایش سرعت *۱۷۶ برای پردازنده گرافیکی و *۲۴۴ را برای پردازنده گرافیکی S ۲ دارد. مقادیر برای مجموعه داده‌های حقیقی به ترتیب *۱۱۷ و*۱۷۱ می‌شود. کاهش عامل‌های افزایش سرعت برای الگوریتم سه مقوله‌ای به خاطر تعداد بالاتر شکافت‌های داده‌ای (۴ برابر بیش از الگوریتم تک مقوله‌ای) و درخواست‌های کرنل بعدی و عملیات مربوط است.

در کل ما مطرح می‌کنیم که وقتی فیلترینگ سه‌بعدی۲ مورداستفاده قرار داده می‌شود سرعت به‌طور قابل‌توجه افزایش پیدا می‌کند، به‌طور متوسط با اندازه پنجره d افزایش پیدا می‌کند. و توسط فاکتورها کمتر تأثیرپذیر است، به‌عنوان‌مثال شکل (۴-۳)، تصاویر بازیابی شده و حذف نویز برای مجموعه داده‌ای Phantomag تحت پیکربندی زیر را نشان می‌دهد: حجم ۳D، یک مقوله‌ای و چند مقوله‌ای d=۱، v=۳، نویز ریشن۳ و واریانس یکنواخت را نشان می‌دهد.

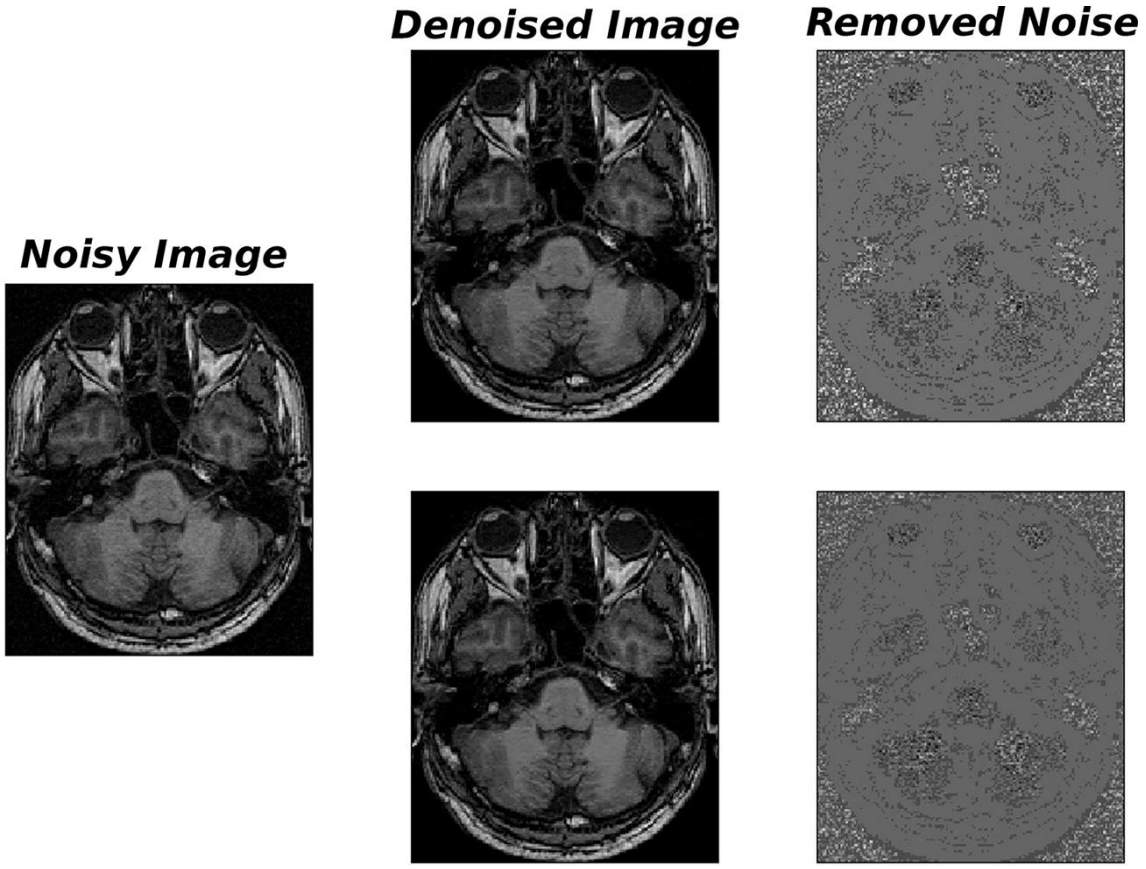
[۱] Real data set
[۱۳] D
[۲] Rician



شکل (۴-۳) مثال‌هایی از فیلتر NLM به‌کاربرده شده در قسمت بیست و چهارم آزمایش Phantomag را نشان می‌دهد. سمت چپ : تصویر نویز دار اصلی قسمت بالایی تصویر با حذف نویز و حذف نویز متناظر تحت پیکربندی سه‌بعدی (۳D) تک مقوله‌ای نویز راشن (Racian) واریانس یکنواخت فضایی v=۳ ،d=۱ را نشان می‌دهد.

مثال‌هایی از فیلتر NLM به‌کاربرده شده در قسمت بیست چهارم آزمایش داده‌ای حقیقی را نشان می‌دهد. قسمت چپ: تصویر نویز دار اصلی، پانل بالایی: تصویر نویزگیری شده و حذف کامل پذیر متناظر تحت پیکربندی سه‌بعدی۱ تک مقوله‌ای، نویز راشن۲، واریانس یکنواخت فضایی d=۱، v=۳ پانل پائین: تصویر نویزگیری شده و نویز کاملاً حذف‌شده متناظر با پیکربندی سه‌بعدی، چند مقوله‌ای، نویز راشن واریانس یکنواخت فضایی d=۱، v=۳ است.

[۱] ۳D
[۲] Rician



شکل(۴-۴) به مجموعه داده‌ای حقیقی تحت همان پیکربندی اشاره دارد. شکل‌ها مؤثر و مفید بودن NLM را تائید می‌کنند.

اکنون عملکرد الگوریتم خود با تعدادی از نمونه‌های مدرن یعنی کومو۳ همکاران [۱] و لیو همکاران [۱۱] مقایسه می‌کنیم. برای اینکه انصاف را رعایت کرده باشیم الگوریتم خود را در پیکربندی‌های داده‌ای شبیه آن‌هایی که در مقالات بالا ذکرشده استفاده‌شده اجرا می‌کنیم. بنابراین، الگوریتم NLM تک مقوله‌ای خود با واریانس نویز یکنواخت را مدنظر قراردادیم. همین‌طور دستگاه‌های سخت‌افزاری که روی آن‌ها الگویتم را اجرا کردیم را مورد مقایسه قرار می‌دهیم. در کومو و همکاران [۶] پردازشگر NVIDIA TESLA S۲۰۵۹ مورداستفاده قرار داده شد که شامل چهار بخش GP پردازنده گرافیکی بود که هرکدام دارای حافظه رم (GB ۳) سه گیگابایتی ۴۴۸ هسته پردازشگر بود که با ۱۰۵۱ گیگاهرتزی کار می‌کرد و توانایی محاسبه ۲.۰ گیگاهرتزی را داشت. جدول (۴-۵) زمان محاسباتی پردازنده گرافیکی الگوریتم NLM V=۵، پیکربندی اندازه بلوک (۱،۱۶،۱۶) و حجم سه‌بعدی۱ را با الگوریتم‌های فول آن رولینگ۲ و آن رولینگ جزئی (۶) در چندین اندازه تصویری را مورد مقایسه قرار می‌دهد. این جدول نشان می‌دهد که الگوریتم ما حداقل دو برابر تقریباً در تمام موارد سریع‌تر است. مشخصات سخت‌افزاری

[۱] ۳D
[۲] full unrolling

قابلیت محاسبه کردن مشابه مدار دارند. آزمایش‌ها توسط لی و همکاران[۱۱] را می‌توان روی کارت GeForce GTX ۶۶۰ Ti انجام داد که مجهز به ۱۳۴۴ هسته پردازشگر و GB ۲ حافظه است و دارای توانایی محاسباتی 3D است ما همین شباهت و پنجره‌های جستجو (v=۵ و d=۵،۴،۳،۲،۱) پیکربندی اندازه بلوک (۱،۱۶،۱۶) و حجم سه‌بعدی را در نظر گرفتیم.

زمان محاسباتی (نه ثانیه) الگوریتم NLM تک مقوله‌ای ما در مقایسه با الگوریتم آن رولینگ کامل و جزئی کامو و همکاران را نشان می‌دهد. پیکربندی استفاده‌شده D=۱، V=۵ پیکربندی اندازه بلوک (۱،۱۶،۱۶) و حجم سه‌بعدی است.

جدول (۴-۵) زمان محاسباتی (به ثانیه) الگوریتم NLM تک مقوله‌ای ما در مقایسه با لی و همکاران [۷] پیرامون مجموعه‌های داده‌ای، دستگاه سونوگرافی جنین و کاموتید عروق و پیکربندی‌های متفاوت d و v را نشان می‌دهد.

Dataset size	Our algorithm		Partial unrolling algorithm [۷]		Full unrolling algorithm [۷]	
	۱ GPU	۲ GPUs	۱ GPU	۲ GPUs	۱ GPU	۲ GPUs
٦٤٣	۰.٥٦	۰.٥٥	۰.۹۹	۰.٤۹	۰.٥٦	۰.۱۲
۱۲۸۲?٦٤	۱.۳۰	۰.۷۹	۲.۳۱	۱.۱٥	۲.۳۳	۰.۸۷
۱۲۸۳	۲.٦۷	۱.٦۳	۱.٦۳	۲.۳۱	٤.٤٦	۱.۹٤
۲٥٦۲?۱۲۸	۸.۰۱	٤.۸۱	۱٦.۹	۸.٤٤	۱۷.۹	۷.۷۳
۲٥٦۳	۱٦.۲٥	۹.۷٦	۳۳.۷۹	۱٦.۹	۳٤.۹	۱٦.۳
٥۱۲۲?۱۲۸	۳۹.۱۲	۲٦.۰۷	٦٥.۲	۳۲.٦	۷۱.٤	۳۰.۹
٥۱۲۲?۲٥٦	٦٤-۷۳	۳۸.۸۹	۱۳۱	٦٥.۲	۱٤۰	٦٥
٥۱۲۳	۱۲۹.٦۲	۷۷.۹۰	۲٦٤	۱۳۱	۲۷٦	۱۳۳

Configuration used is d=-1, v=5, block size configuration (16, 16, 1) and 3D volume

جدول (۴-۶) زمان محاسباتی (به ثانیه) الگوریتم NLM تک مقوله‌ای ما در مقایسه با لی و همکاران [۱۱] پیرامون مجموعه‌های داده‌ای، دستگاه سونوگرافی جنین و کاموتید عروق و پیکربندی‌های متفاوت d و v را نشان می‌دهد.

d	v	Ultrasound fetus				Carotid artery dataset			
		Our algorithm		Li et al.[۱۱]		Our algorithm		Li et al.[۱۱]	
		Stp_S	۱ GPU	Stp_S	۱ GPU	Stp_S	۱ GPU	Stp_S	۱ GPU
۱	٥	۱	۲۸.۷۳	۲	۲٦.٦٦	۱	۳۸.۰۱	۲	۳۳.٥۸
۲	٥	۱	۸٤.۱۸	۳	٥۱.۱٥	۱	۱۱۲.۲۷	۳	٥۷.۲۲
۳	٥	۱	۱٥٤.۷٤	٤	۷٥.٥٤	۱	۲۰۷.۲۳	٤	۹۰.٦٤
٤	٥	۱	۲٦٦.۸٦	٥	۹۹.۱۱	۱	۳٥۸.۲۷	٥	۱۲۷.۹٦
٥	٥	۱	۱٥۸.٥۹	٦	۱۷۷.۳٤	۱	۲۲٥.۸۷	٦	۳٥٦.۱۹

یادآوری می‌سازیم که لی و همکاران [۱۱] پارامتر STP-S را مطرح کردند که فاصله بلوک‌های اصلی را

نشان می‌دهد (در نمونه ما این ۱ است) انتخاب آن‌ها به‌صورت دراماتیک تعداد کلی مسیرهایی که لازم است در فیلتر ازلحاظ دقت حذف نویز پردازش شود را نشان می‌دهد. دو پایگاه داده‌ای متفاوت موردتوجه قرار گرفت سونوگرافی جنین، که دارای رزولوشن ۳۳۸*۲۴۷*۴۲۸ پیکسل است و کاروتید عروق که دارای و رزولوشن ۳۳۸*۲۴۷*۳۹۶ است. پیکربندی‌های نتایج به‌دست‌آمده در جدول (۴-۶) نشان داده می‌شود. ما مشاهده کردیم که الگوریتم لی و همکاران [۱۱] نسبت به الگوریتم ما در افزایش d سریع‌تر است (اما در ارتباط با D=۵ این‌گونه نیست.)

اما یادآوری می‌سازیم که الگوریتم ما بلوک‌های اصلی با رزولوشن کامل را مورداستفاده قرار می‌دهد درحالی‌که شماره آن‌ها توسط stp-s کاهش دارد. برای مثال در مورد D=۶ الگوریتم لی و همکاران[۱۱] ۲.۷ برابر سریع‌تر در الگوریتم ما است اما تعداد بلوک‌های اصلی ۵ برابر کوچک‌تر می‌یابد. همین‌طور مطرح می‌سازیم که مشخصات سخت‌افزاری نمونه لی و همکاران [۱۱] نسبت به نمونه‌ی ما قوی‌تر است.

فصل پنجم:
نتیجه گیری و پیشنهادات

۵-۱ نتیجه گیری

پردازنده گرافیکیS راه‌حل کم‌هزینه‌تر دارای اجرا الگوریتم‌های مناسب، همراه با عملکردهای محاسباتی قابل‌مقایسه پارامتر از کامپیوترهای شدیداً پرهزینه نشان می‌دهند. NLM، به خاطر ماهیت ذاتاً موازی بودن را می‌توان در سخت‌افزار پردازنده گرافیکی اجرا کرد. در حقیقت NLM به‌عنوان الگوریتم حذف نویز پیشرفته مطرح است. ثابت‌شده است که آن در مشکلات کاربردی بسیار مؤثر است جایی که تصاویر ذاتاً دارای نویز می‌باشند، مانند تصاویر MR که در این مقاله موردتوجه قرار گرفت. ازلحاظ ساختاری، NLM با توجه به زمان محاسباتی لازم بسیار موردتوجه قرارگرفته است. برای مثال ، اجرای C++مؤثر روی دستگاه‌های intel ۷ مدرن به حدود یک دقیقه زمان لازم دارند تا یک قطعه ۲۵۶*۲۵۶ پیکسلی را مورد پردازش قرار هند. این باعث می‌شود کاربردهای در زمان حقیقی غیرممکن باشد و حتی نمونه‌های آفلاین نیز غیرعملیاتی باشد. برعکس زمان محاسباتی در پیکربندی پردازنده گرافیکیS ۲ به چند صدم ثانیه کاهش پیدا می‌کند.

در ارتباط با متدلوژی چند مقوله‌ای زمان برای پردازنده اصلی و پردازنده گرافیکیS ۲ برای پردازش یک قطعه به ترتیب ۵ دقیقه برای اول و نیم ثانیه برای بعدی است. چون غالباً حذف نویز مجموعه آزمایش هدایت‌شده از آزمایش‌ها برای تنظیم سازی پارامترهای مرتبط به این متدلوژی (یعنی پارامترهای فیلترینگ نیاز دارد). هزینه بالای پردازنده اصلی مانع از اجرا آزمایش‌های کافی دراین‌ارتباط می‌شود.

الگوریتم یک اساس ثابت و محکم برای پیشرفت‌های آینده است که هم پلتفرم سخت‌افزاری و هم متدلوژی را مربوط می‌شود. ازیک‌طرف می‌توان آن را به‌آسانی در مدار Kepler NVIDIA پردازنده گرافیکی هماهنگ‌سازی کرد که ویژگی محاسباتی ۳.۰ و بالاتر را دارد و محدودیت‌هایی در محیط مالتی –جی پی یو[1] وجود دارد که به خاطر مالکیت حافظه دستگاه است. این نقص اصلی الگوریتم NLM حاضر را که با مدیریت کاربرد حافظه مطرح است را برطرف می‌سازد. از طرف دیگر متدلوژی NLM که ذاتاً موازی است را می‌توان در این الگوریتم اجرا کرد.

[1] پردازنده گرافیکی-Multi

منابع و مآخذ

منابع

[۱] Lu Y. Zheng L. Li L. Guo M. ۲۰۱۵. Parallelism vs. speculation: exploiting speculative genetic algorithm on پردازنده گرافیکی. Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, San Francisco, USA, February ۷–۸, pp. ۶۸– ۷۴.

[۲] Ghorpade J. Parande J. et al. ۲۰۱۲. Gpپردازنده گرافیکی processing in کودا architecture. Advanced Computing: An International Journal, vol. ۳, no. ۱, pp. ۱۰۵–۱۲۰.

[۳] Pospichal P. Jaros J. Schwarz J. ۲۰۱۰. Parallel genetic algorithm on the کودا architecture. International Conference on Applications of Evolutionary Computation, Berlin, Heidelberg, pp. ۴۴۲–۴۵۱. [۴] B. Kirk D. W. Hwu W. ۲۰۰۹. In praise of programming massively parallel processors: a hands-on approach. First Edition. Burlington : Morgan Kaufmann.

[۵] Nowotniak R. Kucharski J. ۲۰۱۱.پردازنده گرافیکی-based massively parallel implementation of metaheuristic algorithms. Automation, AGH University of Science and Technology, vol. ۱۵, no. ۳, pp. ۵۹۵–۶۱۱.

[۶] Kołomycki M. ۲۰۱۳. Use nvidia کودا technology to create genetic algorithms with extensive population. ۲۴th European Students' Conference, Berlin, Germany, September ۴–۷, pp. ۱–۷.

[۷] Merrill D. G. Grimshaw A. S. ۲۰۱۰. Revisiting sorting for gpپردازنده گرافیکی stream architectures. ۱۹th International Conference on Parallel Architectures and Compilation Techniques, Vienna, Austria, September ۱۱–۱۵, pp. ۵۴۵–۵۴۶.

[۸] Tzeng S. Patney A. Owens J. D. ۲۰۱۰. Task management for irregular-parallel workloads on the پردازنده گرافیکی. the Conference on High Performance Graphics, Saarbrucken, Germany, June ۲۵–۲۷, pp. ۲۹–۳۷.

[۹] Wong M.-L. Wong T.-T. Fok K.-L. ۲۰۰۵. Parallel evolutionary algorithms on graphics processing unit. IEEE Congress on Evolutionary Computation, Edinburgh, United Kingdom, September ۲–۵, vol. ۳, pp. ۲۲۸۶–۲۲۹۳.

[۱۰] Shah R. Narayanan P. J. Kothapalli K. ۲۰۱۰. پردازنده گرافیکی-accelerated genetic algorithms. the ۳rd Workshop on Parallel Architectures for Bio-inspired Algorithms in conjunction with Parallel Architectures for Compilation Techniques,Vienna, Austria,

September ۱۱–۱۵, pp. ۱–۸.

[۱۱] Navarro C. A. Hitschfeld-Kahler N. Mateu L. ۲۰۱۴. A survey on parallel computing and its applications in data-parallel problems using پردازنده گرافیکی architectures. Communications in Computational Physics, vol. ۱۵, no. ۲, pp. ۲۸۵–۳۲۹.

[۱۲] Zhang S. He Z. ۲۰۰۹. Implementation of parallel genetic algorithm based on کودا. ۴th International Symposium on Intelligence Computation and Applications, Huangshi, China, October ۲۳–۲۵, pp. ۲۴–۳۰. [۱۳] Karegowda A. G. Manjunath A. S. Jayaram M. A. ۲۰۱۱. Application of genetic algorithm optimized neural network connection weights for medical diagnosis. International Journal on Soft Computing, vol. ۲, no. ۲, pp. ۱۵–۲۳.

[۱۴] Arora R. Tulshyan R. Deb K. ۲۰۱۰. Parallelization of binary and real-coded genetic algorithms on پردازنده گرافیکی using کودا. IEEE World Congress on Computational Intelligence, Barcelona, Spain, July ۱۸–۲۳, pp. ۴۴۲–۴۵۱.

[۱۵] Knysh D. S. Kureichik V. M. ۲۰۱۰. Parallel genetic algorithms: a survey and problem state of the art. Computer and Systems Sciences International, vol. ۴۹, no. ۴, pp. ۵۷۹– ۵۸۹.

[۱۶] Sumati V. ۲۰۱۳. Parallel compact genetic algorithm on کودا-c platform. International ۵۲ Journal of Computer Applications, vol. ۸۴, no. ۵, pp. ۱۳–۱۶.

[۱۷] Al-Marakeby A. ۲۰۱۳. Fpga on fpga: implementation of fine-grained parallel genetic algorithm on field programmable gate array. International Journal of Computer Applications, vol. ۸۰, no. ۶, pp. ۲۹–۳۲. [۱۸] Georgiev D. Atanassov E. Alexandrov V. ۲۰۱۴. A framework for parallel genetic algorithms for distributed memory architectures. Fifth Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, New Orleans, Louisiana, USA, ۱۷ November, vol.۱, pp. ۴۷–۵۳.

[۱۹] Wu E. Liu Y. ۲۰۰۸. Emerging technology about gpپردازنده گرافیکی. IEEE Asia Pacific Conference on Circuits and Systems. Grant of University of Macau, China, pp. ۶۱۸–۶۲۲.

[۲۰] Harding S. Banzhaf W. ۲۰۰۷. Fast genetic programming and artificial developemental systems on پردازنده گرافیکیs. ۲۱st International Symposium on High Performance Computing Systems and Applications, Saskatoon, SK, Canada, May ۱۳–

۱۶, pp. ۹۰–۱۰۱.

[۲۱] Cekmez U. Ozsiginan M. Sahingoz O. K. ۲۰۱۳ . Adapting the ga approach to solve traveling salesman problems on کودا architecture . ۱۴th IEEE International Symposium on Computational Intelligence and Informatics, Budapest, Hungary, November ۱۹–۲۱, pp. ۴۲۳–۴۲۸.

[۲۲] Ventura S. ۲۰۱۲. Genetic programming – new approaches and successful applications. Augusto D. A. Bernardino H. S. Barbosa H. J. C.. Parallel genetic programming on graphics processing units. Croatia : InTech, pp. ۹۵–۱۱۴.

[۲۲] S. T. Andersen, Efficient chip multi-processor programming, Programming a multi-core processor, Faculty of science, (۲۰۱۱), ۲۱-۳۲.

[۲۳] J. Singler, P. Sanders, and F. Putze, MCSTL: The multi-core standard template library, Proceedings of the ۱۳th International Euro-Par Conference, Springer verlag (۲۰۰۷), ۶۸۲-۶۹۴.

[۲۴] T. H. Cormen, C. E. Leiserson, R. L. Riverst, and C. Stein, Introduction to Algorithms, ۳th Edition, The MIT Press (۲۰۰۹) Chapter ۱۸.

[۲۵] P. J. Varman, S. D. Scheuer, B. R. Iyer, and G. R. Ricard, Merging multiple lists on hierarchical-memory multiprocessors, Journal of Parallel and Distributed Computing ۱۲, ۲ (۱۹۹۱), ۱۷۱-۱۷۷.

[۲۶] P. Tsigas and Y. Zhang, A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise ۱۰۰۰۰, Proceedings of Euromicro Conference on Parallel, Distributed, and Network-Based Processing, IEEE Computer Society (۲۰۰۳), ۳۷۲-۳۹۱.

Abstract

Nowadays, NLM filtering is considered as an advanced tool as an advanced algorithm for removing noise in MRI images. Which belongs to a category of noise cancellation methods for which the noise cancellation intensity in a pixel is selected as the measured average of the intensities in the correct pixels. The logic of calculating NLM size depends on two different categories: First, sizes are defined among pixels as similarity in an area of pixels rather than single pixels. This method selects the local textures of the image. Despite its success with its very natural NLM algorithm, it is computationally highly regarded, unsuitable for up-to-date applications, and even difficult to process offline and adjust its parameters without imposing severe constraints on pixel regions. To overcome these problems, several corrections have been suggested in this dissertation. Over the past decade, GPUs have come to be regarded as a hardware platform that complements the central processing devices in modern computers. This is due to the low cost of GPU cases that exist even on most PCs, as well as increasing technological advances that have improved computing performance more than twice as fast as the main processor or suitable algorithms for large-scale parallelization. On the other hand, the extreme parallel nature of many noise canceling algorithms is compatible with the GPU hardware features, making them a complete tool for speeding up the algorithm. Several experiments are performed on MR images to show the performance of the algorithm in several areas. Comparisons and computations of NLM algorithms using GPUs are also provided.

Keywords: Non-Local Noise, Image Noise Elimination, Magnetic Imaging, Graphic Processing Unit.

Besat Insitute of Higher Education

Faculty of Sciences

Department of Computer Engineering

Parallelism removes non-local noise from MRI images using a graphics card processor

Prepared by:

Mehri Salari

Supervisor:

Dr. Fahimeh Yazdanpanah

A Thesis Submitted as a Partial Fulfillment of the Requirements for the Degree of Master of Science in Software Engineering (M. Sc.)

September ۲۰۱۹