بسم الله الرحمن الرحیم

دانشکده ریاضی و علوم کامپیوتر

پایان‌نامه برای دریافت درجه کارشناسی ارشد در رشته علوم تصمیم و مهندسی دانش

محاسبات موازی در بهینه‌سازی تُنُک

استاد راهنما

دکتر محمود امین‌طوسی

استاد مشاور

دکتر مهدی زعفرانیه

پژوهشگر:

بابک خوش‌نویس

شهریور ۱۳۹۷

باسمه تعالی

فرم ارزشیابی و صورتجلسه دفاع از پایان‌نامه کارشناسی ارشد

فرم ۱۱۳-ت

جلسه دفاع از پایان‌نامه آقای /خانم **بابک خوش‌نویس** دانشجوی رشته علوم تصمیم و مهندسی دانش به شماره دانشجویی ۹۵۱۳۱۳۷۰۳۷ با عنوان:

محاسبات موازی در بهینه‌سازی تُنُک

در مورخه ..... در دانشکده ریاضی و علوم کامپیوتر تشکیل و توسط هیات داوران مورد ارزشیابی قرار گرفت و نمره ..... برابر درجه ..... برای آن تعیین گردید .

به‌این ترتیب از این تاریخ آقای/ خانم بابک خوش‌نویس به عنوان کارشناس ارشد در رشته مذکور شناخته می‌شود .

موارد ارزشیابی	موارد	حداکثر نمره	نمره کسب شده
۱- کیفیت نگارش	رعایت اصول نگارش انسجام در تنظیم بخشهای مختلف ،کیفیت تصاویر ، جداول و اشکال ،تنظیم فهرست ها ، منابع و ماخذ	٤	
۲- کیفیت علمی	بررسی تاریخچه و سابقه تجربی و نظری موضوع انسجام منطقی دربخش های مختلف پایان نامه ،ابتکار و نوآوری ،اهمیت و ارزش علمی پایان نامه ،استفاده از منابع معتبر و جدید ،کیفیت تجزیه و تحلیل یافته ها و نتیجه گیری ،روشن بودن روش کار ،هدف ها و فرضیه های تحقیق ،جدید بودن روش تحقیق	۱۰	
۳-کیفیت ارایه در جلسه دفاع	تسلط بر موضوع و بیان واضح وتفهیم آن ،توانایی در پاسخگویی به سوالات مطرح شده در جلسه ، رعایت زمان ارایه , روش ارائه	٤	
٤-ارزشیابی گزارشات	گزارش های دوره ای پیشرفت کار(حداقل ٤مورد)	۱	
٥- خروجی پایان نامه	مقاله مستخرج از پایان نامه: این نمره به صورت زیر اختصاص می یابد ۱) چکیده کنفرانسی هر مورد ۰/۲۵ نمره تا سقف۰/۵ نمره ۲) مقاله کامل در مجموع مقالات همایشهای معتبر یا مقاله در مجلات علمی-ترویجی معتبر پذیرفته شده یا چاپ شده هر مورد ۰/۵ نمره تا سقف ۱نمره ۳)مقاله پذیرفته شده یا چاپ شده در مجلات علمی پژوهشی معتبر ۱ نمره ٤)مقاله ارسال شدهبه مجلات علمی پژوهشی معتبرهر مورد ۲۵/. نمره تا سقف ۰/۵ نمره ٥) دستگاه ساخته شده دارای گواهی ثبت اختراع یا به سفارش سازمان ها تا سقف ۱ نمره ٦) دستگاه ساخته شده کاربردی که به تایید رئیس دانشکده رسیده باشد تا سقف ۰/۵ نمره	۱	
جمع			

درجه معادل کسب شده: (از ۱۹تا ۲۰ عالی☐ از۱۸تا ۱۸/۹۹بسیارخوب ☐ از ۱٦تا ۱۷/۹۹خوب ☐ از ۱٤ تا ۱۵/۹۹قابل قبول☐ کمتر از ۱٤ غیر قابل قبول☐)

مشخصات هیات دوران

ردیف	نام و نام خانوادگی	سمت	مرتبه علمی	محل کار	امضـــــــــــا
۱	دکتر محمود امین‌طوسی	استاد راهنما	استادیار	دانشگاه حکیم سبزواری	
۲	دکتر مهدی زعفرانیه	استاد مشاور	استادیار	دانشگاه حکیم سبزواری	
۳	دکتر مینا مسعودی‌فر	استاد داور	استادیار	دانشگاه حکیم سبزواری	
۴	دکتر رحیمه پورخاندانی	نماینده تحصیلات تکمیلی	استادیار	دانشگاه حکیم سبزواری	

امضا

مدیر گروه

امضا

رئیس دانشکده

سوگند نامه دانش آموختگان دانشگاه حکیم سبزواری

به نام خداوند جان و خرد کزین برتر اندیشه بر نگذرد

اینک که به خواست آفریدگار پاک، کوشش خویش و بهره گیری از دانش استادان و سرمایه‌های مادی و معنوی این مرز و بوم، توشه‌ای از دانش و خرد گردآورده‌ام، در پیشگاه خداوند بزرگ سوگند یاد می‌کنم که در به کارگیری دانش خویش، همواره بر راه راست ودرست گام بردارم. خداوند بزرگ، شما شاهدان، دانشجویان و دیگر حاضران را به عنوان داورانی امین گواه می گیرم که ازهمه دانش و توان خود برای گسترش مرزهای دانش بهره گیرم و از هیچ کوششی برای تبدیل جهان به جایی بهتر برای زیستن، دریغ نورزم. پیمان می‌بندم که همواره کرامت انسانی را در نظر داشته باشم و همنوعان خود را در هر زمان و مکان تا سر حد امکان یاری دهم. سوگند می خورم که در به کارگیری دانش خویش به کاری که با راه و رسم انسانی، آیین پرهیزگاری، شرافت و اصول اخلاقی برخاسته از ادیان بزرگ الهی، به ویژه دین مبین اسلام، مباینت دارد دست نیازم. همچنین در سایه اصول جهان شمول انسانی و اسلامی، پیمان می‌بندم از هیچ کوششی برای آبادانی و سرافرازی میهن و هم میهنانم فروگذاری نکنم و خداوند بزرگ را به یاری طلبم تا همواره در پیشگاه او و در برابر وجدان بیدار خویش و ملت سرافراز، بر این پیمان تا ابد استوار بمانم.

نام و نام خانوادگی: بابک خوش‌نویس

تاریخ و امضا:

تأییدیه‌ی صحت و اصالت نتایج

باسمه تعالی

اینجانب بابک خوش‌نویس به شماره دانشجویی ۹۵۱۳۱۳۷۰۳۷ دانشجوی رشته علوم تصمیم و مهندسی دانش مقطع تحصیلی کارشناسی ارشد تأیید می‌نمایم که کلیه‌ی نتایج این پایان‌نامه حاصل کار اینجانب و بدون هرگونه دخل و تصرف است و موارد نسخه برداری شده از آثار دیگران را با ذکر کامل مشخصات منبع ذکر کرده‌ام. درصورت اثبات خلاف مندرجات فوق، به تشخیص دانشگاه مطابق با ضوابط و مقررات حاکم (قانون حمایت از حقوق مؤلفان و مصنفان و قانون ترجمه و تکثیر کتب و نشریات و آثار صوتی، ضوابط و مقررات آموزشی، پژوهشی و انضباطی ...) با اینجانب رفتار خواهد شد و حق هرگونه اعتراض درخصوص احقاق حقوق مکتسب و تشخیص و تعیین تخلف و مجازات را از خویش سلب می‌نمایم. در ضمن، مسؤولیت هرگونه پاسخگویی به اشخاص اعم از حقیقی و حقوقی و مراجع ذی صلاح (اعم از اداری و قضایی) به عهده ی اینجانب خواهد بود و دانشگاه هیچ گونه مسؤولیتی در این خصوص نخواهد داشت.

نام و نام خانوادگی: بابک خوش‌نویس

تاریخ و امضا:

مجوز بهره برداری از پایان‌نامه

بهره برداری از این پایان‌نامه در چهارچوب مقررات کتابخانه و با توجه به محدودیتی که توسط استاد راهنما به شرح زیر تعیین می‌شود، بلامانع است:

☐ بهره برداری از این پایان‌نامه برای همگان بلامانع است.

☐ بهره برداری از این پایان‌نامه با اخذ مجوز از استاد راهنما، بلامانع است.

☐ بهره برداری از این پایان‌نامه تا تاریخ ..... ممنوع است.

استاد راهنما: دکتر محمود امین‌طوسی

تاریخ و امضا:

تقدیم به:

پدر و مادرم

سپاس خداوندگار حکیم را که با لطف بی کران خود، آدمی را زیور عقل آراست.

در آغاز وظیفه خود می دانم از زحمات بی دریغ استاد راهنمای خود، جناب آقای دکتر محمود امین‌طوسی و دکتر مهدی زعفرانیه صمیمانه تشکر و قدردانی کنم که قطعاً بدون راهنمایی های ارزنده ایشان، این مجموعه به انجام نمی رسید.

از جناب آقای دکتر امین‌طوسی و همچنین جناب آقای دکتر زعفرانیه که زحمت مطالعه و مشاوره این رساله را تقبل فرمودند و در آماده سازی این رساله، به نحو احسن اینجانب را مورد راهنمایی قرار دادند، کمال امتنان را دارم.

همچنین لازم می دانم از گروه پارسی‌لاتک در پاسخگویی به مشکلات کاربران کمال قدردانی را داشته باشم.

در پایان، بوسه می زنم بر دستان خداوندگاران مهر و مهربانی، پدر و مادر عزیزم و بعد از خدا، ستایش می‌کنم وجود مقدس شان را و تشکر می‌کنم از خانواده عزیزم به پاس عاطفه سرشار و گرمای امیدبخش وجودشان، که بهترین پشتیبان من بودند.

بابک خوش‌نویس

شهریور ۱۳۹۷

فهرست مطالب

فهرست جداول

فهرست تصاویر

فرم چکیده ی پایان‌نامه ی دوره ی تحصیلات تکمیلی

مدیریت تحصیلات تکمیلی

دانشگاه حکیم سبزواری

نام خانوادگی دانشجو: خوش‌نویس	نام: بابک	ش. دانشجویی: ۹۵۱۳۱۳۷۰۳۷
استاد راهنما: دکتر محمود امین‌طوسی		
استاد مشاور: دکتر مهدی زعفرانیه		
دانشکده ریاضی و علوم کامپیوتر	رشته: علوم تصمیم و مهندسی دانش	
مقطع: کارشناسی ارشد	تاریخ دفاع: شهریور ۱۳۹۷	تعداد صفحات: ۹۶
عنوان پایان‌نامه: محاسبات موازی در بهینه‌سازی تُنُک		
کلید واژه‌ها: نمایش تُنُک، کدینگ تُنُک، بهینه‌سازی، محاسبات موازی، محاسبات گرافیکی، پردازنده گرافیکی، کودا		

چکیده: امروزه بهینه‌سازی تُنک به عنوان یک مدل جدید و کارآمد در اکثر مسائل و مدل‌سازی‌ها مورد استفاده وسیع قرار می‌گیرد. اغلب حل این مسائل خصوصا در مورد داده‌های حجیم با پیچیدگی محاسباتی بالا و کندی عملکرد همراه است. در چنین شرایطی موازی‌سازی راه‌کاری موثر تلقی می‌گردد.

در این پایان‌نامه روش‌های مختلف موازی‌سازی در جهت بهبود عملکرد مسائلی که در آن‌ها از بهینه‌سازی تُنُک استفاده شده، مورد تحلیل و بررسی قرار گرفته‌است. ابتدا مباحث ریاضی نمایش تُنک شرح داده شده است. سپس الگوریتم جستجوی تطابقی بیان می‌گردد، جستجوی تطابقی متعامد با جزئیات بیش‌تر شرح داده می‌شود. معماری پردازنده‌های مرکزی و گرافیکی با هم مقایسه شده و محاسبات موازی ناهمگون به عنوان روشی نوین در موازی‌سازی توضیح داده می‌شود. مقدماتی از برنامه‌نویسی کودا که یکی از موثرترین راه‌کارها برای موازی‌سازی روی پردازنده گرافیکی می‌باشد، بیان می‌گردد. در پایان سعی شده کلیه روش‌ها، امکانات و ابزار موازی‌سازی در متلب از قبیل برداریزه کردن، حلقه‌های parfor، آرایه‌های جی‌پی‌یو، کرنل و توابع cudamex به صورت عملی پیاده‌سازی و تجزیه تحلیل شود که در زمینه فراتفکیک‌پذیری (برنامه تصاویر وضوح فوق‌العاده) موازی‌سازی روی پردازنده مرکزی به بهبود نزدیک به ۵۰ درصدی سرعت اجرای برنامه انجامیده است. همچنین تلاش برای به‌کارگیری کُد آماده الگوریتم جستجوی تطابقی روی داده‌های تصادفی در فراتفکیک پذیری انجام شده‌است.

پیش‌گفتار

تئوری نمایش تُنک یک مدل جدید و موثر به همراه ابزار مورد نیاز برای استفاده عملی از آن را مطرح می‌نماید. نمایش تنک مبتنی بر دانش گسترده‌ای‌ست که در دهه‌های اخیر گردآوری شده‌است. این مدل برپایه تبدیلات و تئوری‌ها در پردازش سیگنال (همچون تئوری موجک، تجزیه تحلیل چندسطحی و ...)، مفاهیم ریاضی (همچون جبرخطی، نگرش بهینه‌سازی، نگرش تخمین)، و فراتر از این‌ها نمایش تنک ایده‌هایی از یادگیری ماشین در زمینه انطباق یک مدل با یک منبع داده را وام‌دار است. محصول تمامی این زیرساخت‌ها مدلی است که در راستای کاربردهای متعددی همچون فشرده‌سازی، نویززدایی، وضوح فوق‌العاده و ... نشان داده‌شده‌است.

اگر سیگنال ورودی بردار ستونی x باشد و بردار نمایش α باشد، آن‌گاه ارتباط بین این دو یک دستگاه خطی است که در آن ماتریس D دیکشنری می‌باشد که ستون‌های آن اتم‌ها با اندازه سیگنال x هستند. با داشتن D و x ما به دنبال تنک‌ترین جواب در این دستگاه هستیم. در واقع هدف ارائه ساده‌ترین راه برای توصیف x به عنوان یک ترکیب خطی با کم‌ترین اتم‌های ممکن است.

یکی از چالش‌هایی که الگوریتم‌های بهینه‌سازی تُنک با آن مواجه هستند، کاهش عملکرد و سرعت اجرای آن‌ها با افزایش حجم داده‌های ورودی می‌باشد. به عنوان مثال در مسئله بهبود کیفیت تصاویر یا تصاویر با وضوح بالا، در صورت بزرگ‌تر شدن تصویر ورودی، عملکرد برنامه ضعیف و زمان اجرا بالا می‌رود. این مسئله خصوصا در کاربردهای دنیای واقعی و مسائل مربوط به آن غیرقابل قبول است و لذا نیاز به بهبود عملکرد مدل‌سازی تُنک به شدت احساس می‌شود.

موازی‌سازی راه‌کاری بسیار کارآمد است که در اکثر مسائل برای افزایش سرعت و بهبود عملکرد مورد استفاده قرار می‌گیرد. در گذشته موازی‌سازی با استفاده از پردازنده مرکزی بسیار مورد استفاده قرار گرفته و امروزه نیز با افزایش توان این پردازنده‌ها همچنان از این روش استفاده می‌شود. اما با پیشرفت و توسعه پردازنده‌های گرافیکی و فراهم شدن امکان انجام محاسبات عمومی روی این دستگاه‌ها (که قبلا تنها مختص پردازش عملیات پیکسلی و تصویری بودند) و همچنین ماهیت موازی و معماری منحصربفرد آن‌ها، علم محاسبات موازی متحول و دگرگون شد.

هدف اصلی این پایان‌نامه استفاده از روش‌ها و راه‌کارهای موازی‌سازی برای بهبود عملکرد عملیات بهینه‌سازی تنک و کاربرد آن در پردازش تصویر است. در این راستا تعدادی مقالات و پایان‌نامه‌های مرتبط مورد مطالعه و بررسی قرار گرفت و همچنین سعی شد کلیه روش‌ها و راه‌کارهای موازی‌سازی در عمل مورد آزمایش و مقایسه قرار گیرند.

این نوشتار شامل ۴ فصل است:

در فصل ۱، بهینه‌سازی تُنک معرفی و کاربردهای آن شرح داده شده است. همچنین سعی بر این بوده مفاهیم ریاضی به صورت ترتیبی و با بیان ساده شرح داده شود. در بخش پایانی این فصل، به الگوریتم جستجوی تطابقی متعامد به عنوان نمونه پرداخته شده‌است که از انواع الگوریتم‌های جستجو می‌باشد و برای حل مسائل تُنک استفاده می‌شود.

در فصل ۲، مفاهیم اولیه و بنیادین درباره موازی‌سازی و انواع آن توضیح داده شده‌است. معماری پردازنده مرکزی و پردازنده گرافیکی مقایسه شده است، توضیحات بیش‌تری در مورد ویژگی‌ها و قابلیت‌های پردازنده گرافیکی و ساختار حافظه در این پردازنده‌ها ارائه شده‌است چرا که در راستای بهره‌وری حداکثری از این پردازنده نیاز مبرم به شناخت انواع حافظه به کار گرفته شده در آن وجود دارد. همچنین در این فصل مقدمات برنامه‌نویسی کودا که به زبان سی شرح داده‌شده است، بیان می‌گردد.

در فصل ۳، مستندات و ملزومات لازم برای پیاده‌سازی راه‌کارهای موازی‌سازی در نرم‌افزار برنامه‌نویسی متلب بیان شده‌است تا بتوان با استفاده از این روش‌ها، برنامه‌های نوشته شده به زبان متلب را موازی و عملکرد آن‌ها را بهبود بخشید.

در فصل ۴، کارهای انجام شده شرح داده شده‌است . موازی‌سازی برنامه متلب تصاویر وضوح بالا بر روی پردازنده مرکزی انجام شده که منجر به بهبود ۲ برابری عملکرد برنامه شده‌است. همچنین سعی شده برای موازی‌سازی و کاهش زمان اجرای این برنامه از امکانات متلب در موازی‌سازی روی پردازنده گرافیکی استفاده شود و نتایج حاصل شده بررسی گردد. در پایان برنامه کودا جستجوی تطابقی با استفاده از داده‌های تصادفی که با استفاده از کتابخانه cuBlas نوشته شده مورد تحلیل و بررسی قرار گرفته‌است.

فصل ۱

نمایش تنک

۱-۱ مقدمه

در ابتدا کار را با تشریح هدف اصلی مبحث نمایش تنک [۲] آغاز می‌کنیم. لازم است بدانیم این مبحث درباره چیست؟ پاسخ این سوال بسته به شخص پاسخ دهنده متفاوت است چرا که محققین از رشته‌های مختلفی با این مبحث سروکار دارند. از جمله این رشته‌ها می‌توان به ریاضیات، ریاضیات کاربردی، آمار، پردازش سیگنال و تصویر [۳]، نظریه علوم کامپیوتر، یادگیری ماشین، فیزیک، ژئوفیزیک، روانشناسی و ... اشاره کرد. در این فصل بیش‌تر از دیدگاه پردازش سیگنال و تصویر به مبحث نمایش تنک خواهیم پرداخت. نمایش تنک در واقع یک تبدیل [۴] جدید برای سیگنال‌ها و مطالعات آن‌هاست. ایده تبدیل یک سیگنال و تغییر نمایش آن را در نظر بگیرید. یک تبدیل برای دستیابی به بازدهی بیش‌تر، سادگی در پردازش، سرعت و ... اعمال می‌شود. بدون تبدیل‌ها علم پردازش سیگنال و تصویر غیر قابل تصور است. برخی از تبدیلات رایج شامل تبدیل فوریه [۵] [۱]، تبدیل کسینوسی گسسته [۶] [۲]، تبدیل موجک [۷] [۳] و تحلیل مؤلفه‌های اصلی [۸] [۴] می‌گردد. نمایش تنک به نسبت یک تبدیل جدید و موثر است که در این فصل قصد داریم به معرفی آن بپردازیم. گرچه ساختار این تبدیل از ایده‌های گذشته نشأت یافته اما رفتار متفاوتی را با داده بروز می‌دهد. منابع اصلی استفاده شده در این فصل، [۵] و [۶] می‌باشد.

۱-۲ علم نمایش تنک درباره چیست؟

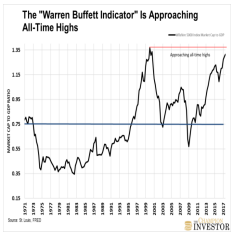
در این بخش به تفسیر گسترده از مبحث نمایش تنک به عنوان یک روش فراگیر برای مدل‌سازی داده می‌پردازیم. با اتخاذ دیدگاه وسیع‌تری نسبت به مبحث پردازش اطلاعات شروع می کنیم. مجموعه داده‌هایی مانند تصاویر ثابت، داده‌های بازار بورس،

---

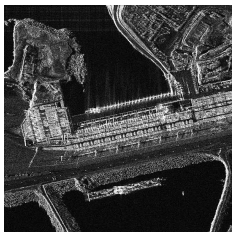
[۲] Sparse Representation [۳] Signal and Image Processing [۴] Transform [۵] Fourier [۶] Discrete Cosine Transform (DCT) [۷] Wawelets [۸] Principle Component Analysis (PCA)

سیگنال‌های صوتی، تصاویر ردیاب‌های ماهواره‌ای، سیگنال‌های نوسان قلب، اطلاعات ترافیک شهری، تصاویر پزشکی و ...را در نظر بگیرید. با وجود تفاوت این مجموعه داده‌ها با یکدیگر، آن‌ها در یک مورد اساسی با هم مشترک هستند. هر کدام از این منابع یک ساختار و نظم داخلی دارند. این ساختار، قواعدی‌ست که این سیگنال‌ها از آن پیروی می‌کنند. در واقع همین ساختارها هستند که امکان پردازش چنین داده‌هایی را فراهم می‌سازند. [۷]

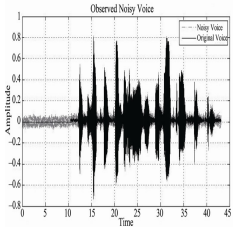
(آ) تصاویر ثابت



(ب) داده‌های بازار بورس

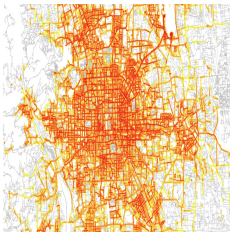


(ج) تصاویر ردیاب ماهواره



(د) سیگنال‌های صوتی

(ه) تصاویر پزشکی



(و) اطلاعات ترافیک

شکل ۱-۱: مجموعه داده‌های مختلف

در یک مثال ساده، برفرض یک حسگر برای سنجش پدیده فیزیکی خاصی در زمان مشخص ساخته شده‌است. هر روز در ساعت خاصی یک اندازه‌گیری انجام می‌دهد. نتایج سنجش در طول یک ماه مورد بررسی قرار می‌گیرند. همانند همه حسگرها، این حسگر نیز کامل نیست و دارای نقص می‌باشد. سازنده حسگر اطلاع داده که دستگاه بوسیله نویز سفید افزایشی با انحراف معیار مشخصی مختل شده‌است. پاکسازی داده‌های جمع‌آوری شده براساس اطلاعات داده‌شده غیر ممکن است. اما اگر یک متخصص در زمینه این پدیده فیزیکی فاش کند که رفتار سیستم در طول زمان از یک فرم قطعی ثابت تبعیت می‌کند و در واقع اندازه‌گیری‌ها به غیر از برخی پرش‌های لحظه‌ای تمایل دارند یکسان بمانند، اکنون با این اطلاعات می‌توان سیگنال را پاکسازی نمود. پاسخ، یک سیگنال پله‌ای در شبیه‌ترین حالت به نمونه‌برداری انجام شده به همراه خطایی که با مقدار نویز حسگر مطابقت دارد خواهد بود. در نتیجه امکان از بین‌بردن نویز، وابسته به مدلی‌ست که برای سیگنال مربوطه ساخته شده‌است. چنین مدل‌هایی تنها برای از بین بردن نویز کارایی ندارند بلکه تقریبا در مورد همه کارهایی که با منابع داده انجام می‌گیرد، به مدل‌سازی نیاز داریم. [۵]

مدل‌های بسیاری با هدف خدمت‌رسانی به سیگنال‌های دلخواه وجود دارند. چنین مدل‌هایی معمولا به صورت توصیف ریاضی قوانینی ارائه می‌شوند که این سیگنال‌ها دنبال می‌کنند. به عنوان مثال برخی از مدل‌هایی که در پردازش تصویر استفاده

می‌شوند [۸] ، عبارتند از تحلیل مؤلفه‌های اصلی، میدان تصادفی مارکف [۱] ، فیلتر وینر [۲] و ... .

وقتی مدل جدیدی ارائه می‌شود، همواره تنش اندکی میان میل به ارائه ساده‌ترین فرم (حفظ سادگی الگوریتم) و نیاز به برقراری عدالت در حفظ محتویات منابع داده وجود دارد. یک مساله مهم دیگر در مدل‌سازی توجه به تطابق مدل با منبع داده است. به عنوان مثال الگوریتم جی‌پگ [۳] [۹] در فشرده‌سازی تصاویر را در نظر بگیرید، این الگوریتم برای تصاویر معمولی عملکرد بسیار خوبی دارد، اما در مورد تصاویر هواشناسی قابل استفاده نیست.

یک بررسی دقیق در علم پردازش سیگنال و تصویر، الگوی مشخصی را در اکثر مقالات نمایان می‌سازد. (مثال : [۱۰]، [۱۱] و [۱۲]) . ابتدا باید سیگنال مورد نظر انتخاب شود (به عنوان مثال تصاویر صورت) سپس مدل مربوطه تعریف می‌شود (مثلا تحلیل مؤلفه‌های اصلی) و در مرحله بعد یک عملیات مهندسی تعیین می‌شود (همچون نویززدایی [۴]). در پایان، این موارد در قالب یک مسئله با قاعده ریاضی ادغام می‌شوند و یک الگوریتم برای حل آن ارائه می‌گردد. لذا ادبیات مبحث پردازش سیگنال و تصویر در واقع سیرتکاملی مدل‌هاست که در طول زمان پیشرفت می‌کنند و همچنین توسعه راه‌کارهایی برای استفاده از آن‌ها در مسائل مختلف که منجر به الگوریتم‌های با کارایی روزافزون می‌گردد.

تئوری نمایش تنک یک مدل جدید و موثر به همراه ابزار مورد نیاز برای استفاده عملی از آن را مطرح می‌نماید. نمایش تنک مبتنی بر دانش گسترده‌ای‌ست که در دهه‌های اخیر گردآوری شده‌است. این مدل برپایه تبدیلات و تئوری‌ها در پردازش سیگنال (همچون تئوری موجک، تجزیه تحلیل چندسطحی و ...)، مفاهیم ریاضی (همچون جبرخطی، نگرش بهینه‌سازی، نگرش تخمین)، و فراتر از این‌ها نمایش تنک ایده‌هایی از یادگیری ماشین در زمینه انطباق یک مدل با یک منبع داده را وام‌دار است. محصول تمامی این زیرساخت‌ها مدلی است که در راستای کاربردهای متعددی همچون فشرده‌سازی، نویززدایی، وضوح فوق‌العاده و ... نشان داده‌شده‌است.

۱-۳ نگاه دقیق‌تر به نمایش تنک

مدل‌سازی را با یک مثال شرح می‌دهیم. یک مدل با قطعات تصویر در اندازه ۸ * ۸ را معرفی می‌کنیم. مدل‌سازی براساس یک دیکشنری انجام می‌شود. دیکشنری، متشکل از یک دسته قطعات تصویر با همان اندازه (۸ * ۸) است. این قطعات اتم‌های مدل ما هستند. در این مثال برفرض ۲۵۶ اتم وجود دارد. تئوری نمایش تنک بیان می‌دارد که هر قطعه ورودی را می‌توان با ترکیب خطی تعداد کمی از اتم‌های دیکشنری تعریف کرد. عبارت "تعداد کم" در اینجا کلیدی‌ست. دو ویژگی مهم در این مدل، **تُنکی** [۵] و **زائدی** [۶] است.

پس با یک قطعه ۸ * ۸ شروع کردیم که حاوی ۶۴ پیکسل است و مدل ارائه شده توصیف جدیدی از این قطعه را به صورت ترکیب خطی اتم‌های دیکشنری ارائه می‌نماید. در نتیجه نمایش جدید یک بردار با اندازه ۲۵۶ است که حامل وزن‌های ترکیب خطی می‌باشد. واضح است که این نمایش دارای خاصیت زائدی است چرا که اندازه آن (۲۵۶)، بزرگ‌تر از اندازه سیگنال (قطعه ۶۴ پیکسلی) می‌باشد. از طرف دیگر این نمایش یک بردار تنک است چرا که بیش‌تر درایه‌های آن صفر

[۱] Markov Random Fields [۲] wiener filter [۳] JPEG [۴] Denoising [۵] Sparsity [۶] Redundancy

می‌باشد. در این مثال برفرض تنها ۳ عضو این بردار غیرصفر هستند. بدین ترتیب مدل نمایش تنک ارائه سیگنال اصلی با ۶۴ پیکسل را تنها با ۶ مقدار امکان‌پذیر می‌سازد. ۳ تا برای مقادیر درایه‌های غیرصفر و ۳ تا برای محل این مقادیر در بردار نمایش.

در واقع نمایش تنک یا هر مدل دیگری درباره کاهش ابعاد هستند با هدف ارائه یک جایگزین و تعریف ساده از داده‌های ورودی برای اثبات این ادعا که اطلاعاتی که روی آن کار می‌کنیم، از آن‌چه به نظر می‌رسند ساده‌تر هستند. نمایش تنک را می‌توان همانند علم شیمی در اطلاعات دانست. در این تفسیر دیکشنری مانند جدول تناوبی عناصر است. سیگنال‌هایی که مدل‌سازی می‌شوند را می‌توان به عنوان ملکول‌ها در نظر گرفت که هر یک ترکیبی از تعداد کمی از عناصر بنیادین هستند.

اگر سیگنال ورودی بردار ستونی x باشد و بردار نمایش α باشد، آن‌گاه ارتباط بین این دو یک دستگاه خطی است که در آن ماتریس D دیکشنری می‌باشد که ستون‌های آن اتم‌ها هم اندازه سیگنال x هستند. با داشتن D و x ما به دنبال تنک‌ترین جواب در این دستگاه هستیم. در واقع هدف ارائه ساده‌ترین راه برای توصیف x به عنوان یک ترکیب خطی با کم‌ترین اتم‌های ممکن است.

$$\overset{\longleftarrow\ m\ \longrightarrow}{\underset{n\ \updownarrow}{\begin{bmatrix} & D & \end{bmatrix}}} \begin{bmatrix} \\ \alpha \\ \\ \end{bmatrix}_{1\times m} = \begin{bmatrix} \\ x \\ \\ \end{bmatrix}_{1\times n} \tag{۱-۱}$$

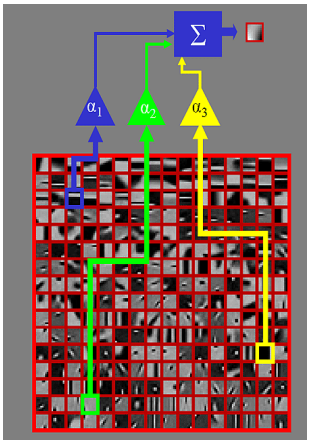
در رابطه ۱-۱ ملاحظه می‌شود که اندازه بردار نمایش تُنک هم اندازه تعداد اتم‌هاست. نمایش تنک ممکن است آسان به نظر برسد اما در واقع با چالش‌های بسیاری همراه است که ممکن است در ابتدا غیرقابل حل به نظر برسند.

۱-۳-۱ اولین چالش، تجزیه اتم

یکی از ابتدایی‌ترین این چالش‌ها **عملیات تجزیه اتم** [۱] است که به روند پیدا کردن تنک‌ترین نمایش ممکن اطلاق می‌شود. در یک مثال اگر ۲۰۰۰ اتم وجود داشته‌باشد و مشخص باشد که تنها ۱۵ تا از آن‌ها برای ساخت سیگنال مورد نظر استفاده شده‌است. قطعا بررسی همه ترکیب‌های ۱۵تایی از ۲۰۰۰ اتم عملیاتی بسیار زمان‌بر و غیرمنطقی است. لذا استفاده از الگوریتم‌های تقریبی [۲] معقول به نظر می‌رسد. نکته بسیار جذاب این است که تحت شرایط خاصی از تعداد عناصر غیرصفر در α، این الگوریتم‌ها یافتن پاسخ بهینه را تضمین می‌کنند.

در تصویر ۱-۲ به صورت نمادین عملیات تجزیه اتم نمایش داده‌شده است. تنها با ترکیب خطی تعداد کمی از اتم‌های دیکشنری می‌توان تقریب قابل قبولی از سیگنال مورد نظر بدست آورد.

[۱] Atom Decomposition [۲] Approximation Algorithms



شکل ۱-۲: تصویر نمادین از عملیات تجزیه اتم

۱-۳-۲ دومین چالش، ساخت دیکشنری

یک محقق از نمایش تنک برای کار با تصاویر طبیعت استفاده می‌کند، محقق دیگری از آن برای سیگنال‌های صوتی و یا دیگری برای پیش‌بینی بازار بورس. طبیعتا هر منبع سیگنال نیاز به دیکشنری متفاوتی دارد و سوال اصلی این‌جاست که چگونه چنین دیکشنری قابل دستیابی‌ست؟ پاسخ مدل نمایش تنک **آموزش** [۱] است. آموزش با استفاده از یک مجموعه داده بزرگ از سیگنال‌ها و یافتن بهترین دیکشنری که منجر به تنک‌ترین توصیف ممکن شود. الگوریتم‌های مختلفی مانند کی-اس‌وی‌دی [۲] در این حوزه وجود دارند. [۱۳] ایده ساخت دیکشنری برای داده‌ها یکی از نقاط قوت برجسته نمایش تنک است. ساخت دیکشنری منجر به امکان به‌کار بردن هر منبعی از داده و در نتیجه خدمت‌رسانی این نمایش به عنوان یک مدل فراگیر را فراهم می‌سازد.

۱-۴ مقدمات ریاضی

در این بخش برخی مفاهیم و مقدمات ریاضی و روند شکل‌گیری و دستیابی به نمایش تُنُک بیان می‌گردد. در آماده‌سازی این بخش به طور کامل از [۵] استفاده شده‌است و اثبات قضایا نیز در آن موجود می‌باشد.

۱-۴-۱ سیستم‌های خطی فرومعین [۳]

اگر در سیستم خطی $Ax = b$، مجهولات بیش‌تری نسبت به معادلات داشته‌باشیم و A با ابعاد m ستون و n سطر باشد، $m > n$ است. با فرض این‌که A رتبه کامل باشد، این سیستم باید بی‌نهایت جواب شدنی داشته‌باشد. قطعا مدل‌سازی ریاضی مسئله‌ای که جواب‌های شدنی زیادی دارد قانع‌کننده نیست. چنین مسئله‌ای غیرعملی و غیرقابل استفاده است. برای

[۱] Learning [۲] K-SVD [۳] Underdetermined linear systems

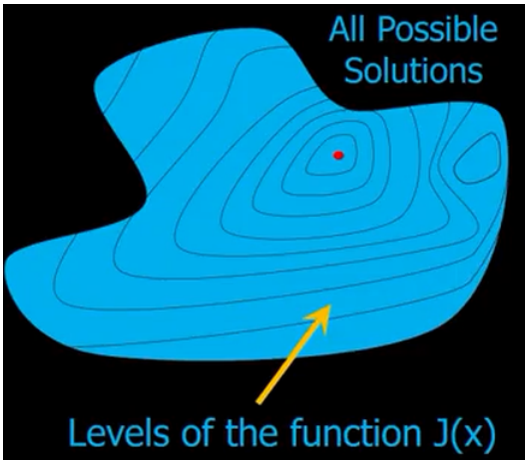
محدودکردن جواب‌های ممکن به یک جواب یکتا چه می‌توان کرد؟

به عنوان مثال، اگر x یک تصویر در ابعاد ۱۰۰در۱۰۰ پیکسل باشد که به یک چهارم اندازه اصلی زیرنمونه‌برداری [۱] شده‌باشد. ارتباط بین تصویر کوچک شده b با تصویر اصلی x (که در عالم واقع ناشناخته و غیرقابل دسترس است) بوسیله یک سیستم معادلات خطی فرومعین ارائه می‌شود. بازسازی x از b عملیات بازسازی تصویر تکی با وضوح بالا [۲] [۱۴] نامیده می‌شود. واضح است که در میان بی‌نهایت تصاویر ممکن x که بتوانند b را توصیف کنند، بهترین آن‌ها مدنظر است.

اکنون می‌توان مفهوم منظم‌سازی [۳] را بیان نمود. ایده این روش این است که براساس میزان کیفیت هر جواب، یک رتبه به هریک از جواب‌های ممکن تخصیص داده‌شود و جواب با بالاترین رتبه به عنوان جواب نهایی انتخاب گردد که آن را به صورت رابطه زیر نشان می‌دهیم:

$$\min_x J(x) \quad s.t. \quad Ax = b \tag{۲-۱}$$

در معادله (۱-۲)، $J(x)$ شرط منظم‌سازی [۴] است که به هر پاسخ شدنی یک رتبه تخصیص می‌دهد. در این‌جا تابع J نمایانگر جریمه است به طوری که مقادیر کم‌تر بهتر تلقی می‌شوند. وقتی عملیات کمینه‌سازی روی x انجام می‌شود، به دنبال کم ارزش‌ترین جوابی که در معادله $Ax = b$ صدق کند هستیم. از دیدگاه هندسی، همه جواب‌های شدنی معادله $Ax = b$ را در نظر گرفته شده و بعد تابع جریمه اعمال می‌شود. هدف شناسایی جوابی‌ست که مقدار کمینه تابع $J(x)$ باشد. چالش اصلی نحوه انتخاب $J(x)$ می‌باشد. واضح است در صورتی که این تابع را به درستی انتخاب کنیم، به هدف



شکل ۱-۳: رهیافت هندسی جستجوی جواب با استفاده از منظم‌سازی

خواهیم رسید. انتخاب $J(x)$ به موارد بسیاری بستگی دارد. برخی از این موارد رفتار مورد انتظار، مجهولات و ابزارهای قابل دسترس می‌باشند. درحالی که مسئله اصلی دارای تکینگی [۵] قدرتمندی است، استفاده از $J(x)$ مسئله را منظم می‌سازد.

---

[۱] Subsampling [۲] Single Image Super-Resolution [۳] Regularization [۴] Regularization Term
[۵] Singularity

رایج‌ترین انتخاب برای $J(x)$، عبارات مبتی بر L_2 به فرم رابطه زیر می‌باشد:

$$J(x) = \frac{1}{2} \|Bx\|_2^2 \qquad \text{برای یک ماتریس انتخاب شده } B \tag{۱-۳}$$

دلایلی که این روش تا این حد رایج است به شرح ذیل می‌باشد:

- عبارات مبتی بر L_2 به سادگی قابل فهم هستند و مشتق‌گیری به راحتی حاصل می‌گردد.
- با این منظم‌سازی، مسئله دارای یک جواب بسته [1] خواهد بود
- تحت شرایط مشخص، یک جواب منحصربفرد برای مسئله منظم‌شده تضمین شده‌است.

مشتق این مسئله با تعریف لاگرانژین [2] آغاز می‌شود ۱-۴.

$$L(x, \lambda) = \frac{1}{2} \|Bx\|_2^2 + \lambda^\dagger (Ax - b) \tag{۱-۴}$$

بردار λ ضریب لاگرانژ برای مجموعه محدودیت‌ها می‌باشد. در مرحله بعد گرادیان لاگرانژین را نسبت به x حذف می‌کنیم:

$$\nabla L = B^\dagger Bx + A^\dagger \lambda = 0 \implies x = -(B^\dagger B)^{-1} A^\dagger \lambda \tag{۱-۵}$$

اگر B رتبه کامل ستونی باشد، یک جواب منحصربفرد بدست می‌آید. اگر چه این جواب برحسب λ بدست آمده‌است. با قراردادن عبارت ۱-۵ در معادله $Ax = b$، امکان یافتن λ فراهم می‌شود :

$$Ax = -A(B^T B)^{-1} A^T \lambda = b \implies \lambda = -(A(B^T B)^{-1} A^T)^{-1} b \tag{۱-۶}$$

و در آخر به جواب بسته مورد نظر می‌رسیم:

$$\implies x = (B^T B)^{-1} A^T (A(B^T B)^{-1} A^T)^{-1} b \tag{۱-۷}$$

سوالی که در این‌جا وجود دارد این است که آیا مزیت‌های L_2 برای استفاده به عنوان منظم‌ساز کفایت می‌کنند؟ آیا همیشه بهترین راهکار است؟ به عنوان مثال در مورد فیلتر وینر که از این روش استفاده می‌کند، نتایج ضعیفی حاصل می‌شود. در نتیجه نیاز به قواعد منظم‌سازی جایگزین با هدف حفظ برخی ویژگی‌های L_2 وجود دارد. لذا به معرفی توابع محدب [3] می‌پردازیم.

---

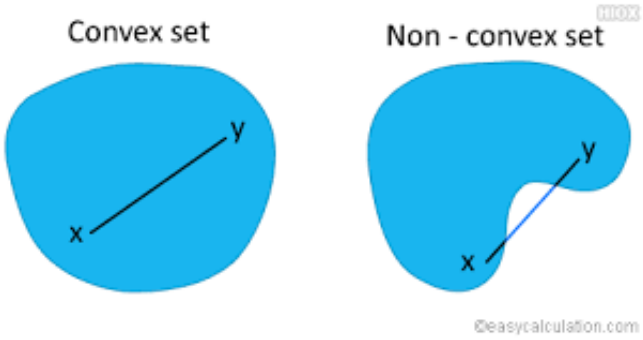
[1] Closed-Form Solution [2] Lagrangian [3] Convex

۱-۴-۲ تحدب

باتوجه به نیاز مبرم به جواب یکتا و منحصربفرد برای مسئله منظم‌سازی، تحدب بر این امر دلالت دارد. بحث را با معرفی مجموعه‌های محدب شروع می‌کنیم.

تعریف ۱-۴-۱ (تحدب). مجموعه Ω محدب است، اگر $\forall x_1, x_2 \in \Omega$ و $\forall t \in [0,1]$ ترکیب محدب $x = tx_1 + (1-t)x_2$ نیز در مجموعه Ω باشد.

به بیان ساده‌تر در مجموعه محدب برای هر جفت نقطه در داخل مجموعه، خط متصل کننده آن‌ها نیز در مجموعه قرار دارد. مجموعه تمام جواب‌های یک سیستم خطی $Ax = b$ یک مجموعه محدب را تشکیل می‌دهد. برای ملاحظه این امر،



شکل ۱-۴: مقایسه یک مجموعه محدب و مجموعه غیرمحدب

Ω را به عنوان مجموعه xهایی که در این معادله صدق می‌کنند قرار می‌دهیم و هر دو نقطه دلخواه در آن را در نظر می‌گیریم.

$$\Omega = \{x | Ax = b\}$$

$$\left.\begin{array}{l} Ax_1 = b \\ Ax_2 = b \end{array}\right\} A(tx_1 + (1-t)x_2) = tAx_1 + (1-t)Ax_2 = tb + (1-t)b = b$$

در تصویر ۱-۴، مجموعه سمت چپ محدب و مجموعه سمت راست نامحدب است.

تعریف ۱-۴-۲ (تابع محدب). تابع $f(x) : \Omega \to \mathbb{R}$ محدب است، اگر $\forall x_1, x_2 \in \Omega$ و $\forall t \in [0,1]$ ترکیب محدب $x = tx_1 + (1-t)x_2$ در رابطه $f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$ صدق کند.

اگر تابع هموار و مشتق‌پذیر باشد، می‌توان تعریف درک‌پذیرتری برای تحدب یک تابع جایگزین نمود.

قضیه ۱-۴-۳. تابع $f(x) : \Omega \to \mathbb{R}$ محدب است اگر $\forall x_1, x_2 \in \Omega$ داشته‌باشیم:

$$f(x_2) \geq f(x_1) + \nabla f(x_1)^\dagger (x_2 - x_1)$$

به بیان دیگر، با استفاده از گرادیان، اگر ابرصفحه مماسی در هر نقطه x تابع را از زیر دربرگیرد، $f(x)$ محدب خواهد بود.

قضیه ۱-۴-۴. تابع $f(x) : \Omega \to \mathbb{R}$ محدب است اگر $\forall x \in \Omega$ داشته‌باشیم:

$$\nabla^۲ f(x) \geq ۰$$

تعریف ۱-۴-۵ (محدب موکد). یک تابع محدب موکد است در صورتی که شرط تساوی از نامساوی حذف شود.

$$\nabla^۲ f(x) > ۰$$

اکنون با توجه به قضایا ۱-۴-۳ و ۱-۴-۴ ، چون مشتق دوم تابع ۱-۳ برای همه Bها نیمه معین مثبت است، لذا این تابع قطعا محدب می‌باشد.

$$\nabla J(x) = B^T Bx$$
$$\nabla^۲ J(x) = B^T B$$

اگر B رتبه کامل ستونی باشد، $B^T B$ معین مثبت خواهد بود و در نتیجه این تابع محدب موکد خواهد شد. با مینیمم کردن تابع محدب روی یک دامنه محدب، هیچ نقطه مینیمم محلی نخواهیم داشت و همه جواب‌های بهینه در یک مجموعه محدب متمرکز خواهند بود. اگر تابع محدب موکد باشد، پاسخ بهینه منحصربفرد خواهد بود.

۱-۴-۳ نُرم L_p

در صورتی که تابع منظم‌سازی $J(x)$ محدب موکد باشد، آن‌گاه بهترین جواب در مجموعه $Ax = b$ دقیقا همانند مسئله $L_۲$ منحصربفرد خواهد بود. بنابراین می‌توان به دنبال توابع منظم‌سازی محدب و یا محدب موکد بود. یک نوع از توابع که دارای این ویژگی هستند، خانواده نرم L_p (برای $p \geq ۱$) نام دارند. $L_۲$ نیز زیرمجموعه همین خانواده است که از مربع آن به عنوان تابع هزینه $J(x)$ برای منظم‌سازی در رابطه ۱-۳ استفاده شده‌است.

تعریف ۱-۴-۶ (نُرم p). نُرم L_p [۱] خانواده خاص و پرکاربردی از نُرم‌ها است که بصورت زیر تعریف می‌شود

$$L_p : \|x\|_p = \left(\sum_k |x_k^p| \right)^{\frac{1}{p}} \quad for \;\; p \in [1, \infty) \tag{۸-۱}$$

تمامی خانواده نُرم L_p منجر به توابع محدب می‌شوند. یک مورد خاص نرم L_1 است که قدر مطلق مجموع مقادیر ورودی بردار را شامل می‌شود.

$$L_1 : \|x\|_1 = \sum_k |x_k| \tag{۹-۱}$$

مورد خاص دیگر مربوط به نُرم بی‌نهایت است که بزرگ‌ترین مقدار ورودی را انتخاب می‌کند.

$$L_\infty : \|x\|_\infty = \max_k |x_k| \tag{۱۰-۱}$$

نرم L_1 محدب است اما محدب موکد نیست. در واقع تمامی خانواده نُرم L_p تنها درصورتی که به توان p برسند محدب موکد خواهند بود. برای $p < 1$ عبارت ۱-۸ محدب نیست. در واقع برای مقادیر $p < 1$ عبارت ۱-۸ نرم محسوب نمی‌شود. نرم در یک فضای برداری تابعی است که روی بردارها عملیات انجام می‌دهد و مقادیر اسکالری تولید می‌کند. این مقادیر به عنوان طول بردار تفسیر می‌گردند. برای این‌که یک تابع به عنوان یک نُرم رسمی شناخته شود، باید از سه قاعده کلی پیروی کند:

- مقدار آن نمی‌تواند منفی باشد و تنها برای بردار صفر مقدارش صفر است.

$$\forall x \|x\| \geq 0 \quad and \quad \|x\| = 0 \rightarrow x = 0$$

- تابع باید از خاصیت همگنی [۲] تبعیت کند. به این معنا که مقدار نُرم برای بردار مقیاس‌یافته برابر با ضریب مقیاس در مقدار نُرم بردار است.

$$\forall x \,\&\, \forall c, \quad \|cx\| = |c| \,.\, \|x\|$$

- آخرین خاصیتی که نرم‌ها باید رعایت کنند، نامساوی مثلثی [۳] است. یعنی نرم مجموع بردارها از مجموع نُرم هر یک از بردارها کم‌تر است.

$$\forall x, y, \quad \|x + y\| \leq \|x\| + \|y\|$$

با توجه به خاصیت همگنی و نامساوی مثلث و روابط زیر نتیجه می‌گیریم که هر تابع نرم معتبر الزاما محدب است و چون

[۱] L_p Norm [۲] Homogeneity [۳] Triangle-Inequality

برای $p < ۱$ تابع محدب نیست، در نتیجه نرم معتبر هم نیست.

$$\|tx + (۱-t)y\| \leq \|tx\| + \|(۱-t)y\| \quad \text{(Triangle)}$$
$$\leq t\|x\| + (۱-t)\|y\| \quad \text{(Homog.)}$$

۱-۴-۴ مینیمم‌سازی $L_۱$

مسئله $P_۱$ به صورت رابطه ۱-۱۱ تعریف می‌شود. این مسئله به دنبال کوتاه‌ترین جواب $L_۱$ برای یک سیستم خطی $Ax = b$ است.

$$(P_۱) \quad \min_x \|x\|_۱ \ \ s.t. \ \ Ax = b \tag{۱۱-۱}$$

محبوبیت مسئله $P_۱$ به این دلیل است که این مسئله به جواب‌های تنک منجر می‌شود. اولین ویژگی مورد بحث در مسئله $P_۱$ در مورد رفتار جواب‌های این مسئله است.

قضیه ۱-۴-۷. مجموعه جواب‌های بهینه $P_۱$ (که با S مشخص می‌شود)، بسته و محدب است.

چرا جواب بهینه یکتا نیست و یک مجموعه است؟ به این دلیل که $L_۱$ محدب موکد نیست. البته قضیه ۱-۴-۷ بیان می‌دارد که این مجموعه جواب متمرکز [۱] است. برای اثبات تحدب این قضیه در مجموعه جواب‌های بهینه S، هر جفت جواب تصادفی $x_۱$ و $x_۲$ در این مجموعه که هر دو به طول مینیمم J_{min} دست یافته‌اند را در نظر می‌گیریم. هر دو آن‌ها در معادله $Ax = b$ صدق می‌کنند. اگر x^* یک ترکیب محدب از این دو جواب باشد، به سادگی می‌توان نشان داد که x^* نیز در معادله $Ax = b$ صدق می‌کند و لذا یک جواب شدنی است. همچنین با درنظر گرفتن طول $L_۱$ برای x^*، می‌توان با استفاده از نامساوی مثلث و همگنی نشان داد که این طول با حد بالای J_{min} احاطه شده است. اما J_{min} کمینه‌ترین طول ممکن است. بنابراین نُرم $L_۱$ برای x^* نیز همان J_{min} است. این نشان می‌دهد که x^* نیز متعلق به S است و لذا تحدب این مجموعه اثبات می‌گردد. بسته‌بودن مجموعه جواب‌های $P_۱$ نیز اثبات شده‌است.

اکنون به کاربرد پیچیده‌تری از منظم‌سازی $L_۱$ و گرایش آن به تولید جواب‌های تنک می‌پردازیم. به دنبال جواب‌هایی برای سیستم‌های خطی هستیم که دارای n معادله با m مجهول ($m > n$) هستند.

قضیه ۱-۴-۸. مجموعه جواب‌های بهینه (S)، مسئله $P_۱$ باید حاوی یک جواب با حداقل n عضو غیرصفر باشد.

جواب‌های شدنی زیادی وجود دارد اما می‌توان در میان آن‌ها تنک‌ترین جواب ممکن با $n < m$ عضو غیر صفر را یافت. البته این میزان تنکی قابل بهبود است. برای حل مسئله $P_۱$ می‌توان آن را به یک مسئله برنامه‌ریزی خطی تبدیل نمود و بعد با استفاده از روش‌های مختلف جواب بهینه را بدست آورد.

[۱] Concentrated

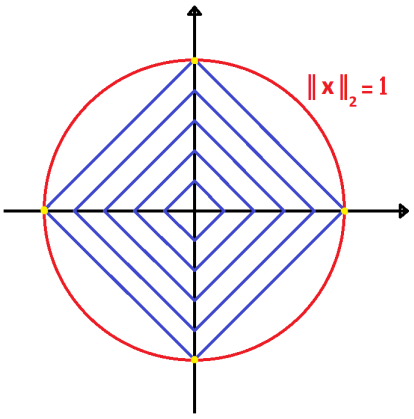
۱-۴-۵ ارتقاء راه‌حل‌های تنک

تا اینجا یک سیستم خطی فرومعین با مجموعه بی‌نهایت جواب شدنی در نظر گرفته‌شد. با اضافه کردن یک تابع منظم‌سازی این مجموعه محدود شد. در مورد این که چه نوع تابع منظم‌سازی انتخاب شود، بحث شد. برای منظم‌سازی تابع $L_۲$ و تولید جواب منحصربفرد با استفاده از آن بررسی شد. همچنین در مقیاس وسیع‌تر نشان داده‌شد که منظم‌سازی محدب موکد نیز منجر به جواب یکتا می‌شود. سپس جایگزین $L_۱$ معرفی شد که نسبت به تولید جواب‌های تنک گرایش دارد. همچنین ذکر شد که مسئله $P_۱$ یک مسئله برنامه‌ریزی خطی‌ست که روش‌های متعددی برای حل آن وجود دارد. اما انتظار برای رسیدن به جواب‌های تنک‌تر وجود دارد.

سوال اینجاست که چرا $L_۱$ به جواب‌های تنک منجر می‌شود؟ برای پاسخ به این سوال، طبق رابطه ۱-۱۶ همه جواب‌های شدنی روی کره واحد $L_۲$ را درنظر می‌گیریم و کوتاه‌ترین بردار در $L_۱$ در این مجموعه را جستجو می‌کنیم.

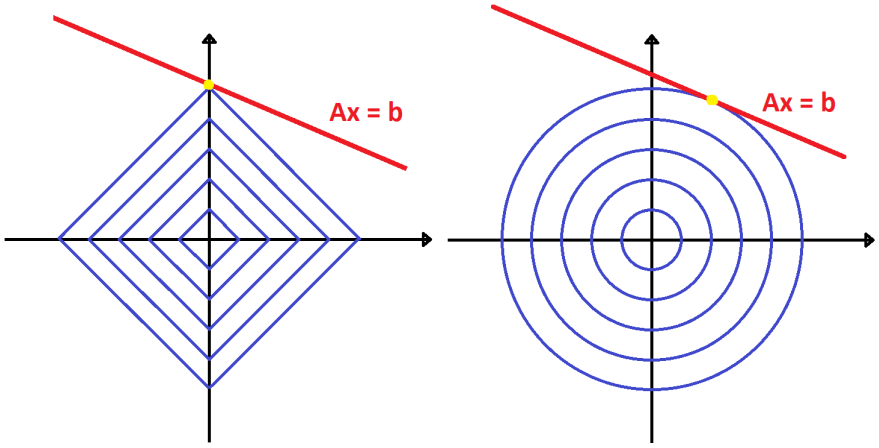
$$\min \|x\|_۱ \quad s.t. \quad \|x\|_۲ = ۱ \tag{۱-۱۲}$$

این مسئله با اتخاذ یک دیدگاه هندسی قابل حل است و برای ساده‌سازی، به صورت دوبعدی نشان داده شده‌است. همان‌طور که در شکل زیر دیده می‌شود، کره واحد $L_۲$ با دایره نشان داده‌شده است که در میان تمام نقاط روی این دایره، به دنبال نقاطی با کم‌ترین مقدار $L_۱$ هستیم. بنابراین آن‌قدر در تابع $L_۱$ می‌دمیم تا محدوده دایره را لمس کند. همان‌طور که می‌بینید، جواب‌ها روی محورهای مختصات پیدا شده‌اند (نقاط گوشه‌ای در تداخل دایره و بزرگ‌ترین مربع). در نتیجه کوتاه‌ترین بردارها در $L_۱$ تنها با یک عضو غیرصفر، گرایش به تُنک بودن دارند. اکنون برای روشن‌تر شدن علت تنکی $L_۱$، دو مسئله $P_۱$ و $P_۲$



شکل ۱-۵: جستجوی جواب بهینه $L_۱$ با محدودیت کره واحد $L_۲$ (دوبعدی)

را درنظر می‌گیریم و مجددا این مسائل را با شمایل دوبعدی به همراه یک معادله خطی به عنوان شرط آن نمایش می‌دهیم. جواب‌های شدنی روی خطوط هستند. با دمیدن توپ‌های $L_۱$ و $L_۲$، به محل جواب بهینه دست می‌یابیم. نتیجه واضح است. درحالی‌که $L_۲$ جواب‌های متراکم ارائه می‌دهد، جواب‌های ارائه شده توسط $L_۱$ تُنک هستند. حال که نرم $L_۱$ نسبت به نرم $L_۲$ نتایج بهتری دارد، چرا برای منظم‌سازی از تابع نرم با $p < ۱$ استفاده نکنیم؟ در شکل ۱-۷ شمایل هندسی برای



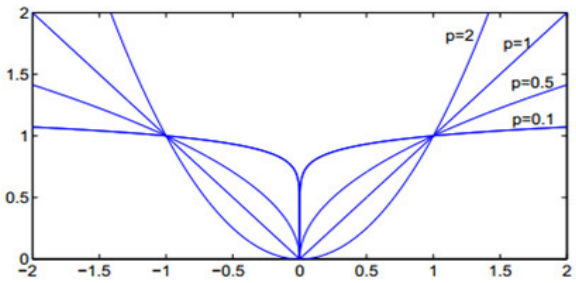
شکل ۱-۶: مقایسه L_1 و L_2 با محدودیت خطی در رسیدن به جواب

$p < 1$ در مقایسه با P_1 نشان داده شده‌است. می‌دانیم که این انتخاب یک نُرم معتبر نیست. و مهم‌تر از آن، می‌دانیم که محدب نیست. اما اگر هدف تولید جواب‌های تنک باشد، ممکن است این روش درستی باشد.

۱-۴-۶ استفاده از نرم L_0 برای منظم‌سازی

با توجه به اینکه هدف یافتن تنک‌ترین جواب ممکن برای سیستم خطی $Ax = b$ است، می‌توان در این راستا p را صفر در نظر گرفت. برای درک بهتر این انتخاب، نُرم عمومی L_p را به توان p فرض می‌کنیم (رابطه ۱-۱۸). وقتی $p = 2$ باشد، تاثیر هر مقدار ورودی منحنی درجه دوم خواهد بود. نمودار $p = 1$ قدرمطلق خواهد بود. برای $p < 1$، نمودار منحنی مقعر خواهد بود. وقتی p به سمت صفر میل می‌کند، حرکت منحنی به سمت تابع نشان‌گر [1] نزدیک و نزدیک‌تر می‌شود. به این‌صورت که در همه جا مقدار این تابع ۱ است و تنها در $x = 0$ مقدار صفر دارد.

$$\|x\|_p^p = \sum_k |x_k|^p \qquad (۱-۱۳)$$



شکل ۱-۷: مقایسه نرم‌ها

[1] Indicator Function

تعریف ۱-۴-۹ (نُرم L_0). یک بردار $x \in \mathbb{R}^m$، نُرم L_0 [1] آن به صورت رابطه زیر (۱-۱۹) تعریف می‌شود:

$$\|x\|_0 \triangleq \lim_{p \to 0} \|x\|_p^p = \sum_{k=1}^{m} I(x_k) \qquad (۱۴-۱)$$

در واقع نُرم L_0 مجموع اعمال توابع نشان‌گر $I(x)$ روی تمامی ورودی‌های بردار x است.به بیان ساده‌تر، نُرم L_0 تعداد عناصر غیرصفر در x را برمی‌گرداند. برخلاف این واقعیت که ما این تابع را نُرم L_0 می‌نامیم، واضح است که این تابع یک نُرم معتبر نیست. چرا که این تابع به شدت غیرمحدب است. در بررسی سه قاعده کلی نُرم‌ها برای L_0 نیز شرط غیرمنفی بودن رعایت شده اما شرط همگنی نقض شده چرا که شمارش تعداد عناصر غیرصفر با تغییر مقیاس بردار تحت تاثیر قرار نمی‌گیرد. با کمال تعجب نامساوی مثلث رعایت شده است.

۱-۴-۷ مسئله P_0

این مسئله بیان‌گر استفاده از نُرم L_0 برای منظم‌سازی است. با وجود شباهت بسیار مسئله P_0 با مسئله P_2، نیاز به اطلاعات بیش‌تری درباره این مسئله وجود دارد.

تعریف ۱-۴-۱۰ (مسئله P_0). مسئله P_0 به دنبال تنک‌ترین جواب‌های معادلات با سیستم خطی نامعین، $Ax = b$، می‌باشد:

$$(P_0) \quad \min_x \|x\|_0 \ \ s.t. \ \ Ax = b \qquad (۱۵-۱)$$

در مسئله P_2 جواب منحصربفرد است و به راحتی بدست می‌آید. آیا در مورد مسئله P_0 نیز یک جواب یکتا وجود دارد؟ تحت چه شرایطی؟ درصورت وجود آیا این جواب بهینه است؟ آیا چنین جوابی در زمان معقول قابل دستیابی است؟

باید تاکید داشت که P_0 یک مسئله بسیار سخت است. به عنوان مثال اگر فرض کنیم دیکشنری A با $m = ۲۰۰۰$ ستون و $n = ۵۰۰$ سطر داشته‌باشیم. اگر b با ۲۰ ستون از A قابل بازسازی باشد (می‌دانیم که جواب ۲۰ تنک وجود دارد)، تنها راه اصولی که رسیدن به این جواب را تضمین می‌کند، بررسی تمام جواب‌های ممکن میان $\binom{۲۰۰۰}{۲۰} = ۳/۹\,e+۴۷$ حالت مختلف و حل سیستم خطی برای هر یک از این حالات می‌باشد. اگر فرضا هریک از این محاسبات یک نانو ثانیه زمان ببرد، با عملیات محاسباتی مواجه خواهیم بود که میلیاردها سال زمان می‌برد.

در یک آزمایش اگر A را به نوعی به صورت تصادفی تولید کنیم، b را نیز تصادفی ترسیم کنیم و بعد این دو را به مسئله P_0 بفرستیم تا تنک‌ترین x را بدست آورد، تنک‌ترین جواب ممکن برابر n خواهد بود. زیرا احتمال این‌که b توسط ستون‌های کم‌تری از A ساخته‌شود، صفر است. اما طبق قضیه ۱-۴-۸ دیدیم که با استفاده از نرم L_1 هم رسیدن به جواب n-تنک ممکن بود. پس استفاده از P_0 چه مزیتی دارد؟

[1] L_0 Norm

یک روش بهتر برای حل مسئله وجود دارد. ماتریس A همانند قبل به صورت تصادفی ساخته می‌شود. اما b به طرز متفاوتی ترکیب می‌شود. برای ساخت b ابتدا یک بردار بسیار تنک x_0 تولید می‌شود به طوری‌که طول این بردار m و دارای s عنصر غیرصفر است که s بسیار کوچکتر از n می‌باشد، ($s \ll n$). سپس b به صورت $b = Ax_0$ تولید می‌شود و حالا ماتریس A و b به مسئله P_0 می‌شوند. حالا جستجو برای جواب تنک معنادار می‌شود. در واقع در مبحث بهینه‌سازی تنک قرض بر این است که از این سناریو استفاده می‌شود. هدف بررسی همه بردارهای b نیست، بلکه تنها آن‌هایی که واقعا دارای یک نمایش تنک هستند مدنظر است.

۱-۴-۸ چشم‌انداز پردازش سیگنال

در این بخش انگیزه استفاده از مسئله P_0 و نرم L_0 را بوسیله ارتباط آن‌ها با نیازهای واقعی در پردازش سیگنال بیان می‌شود. ممکن است تا اینجا تحمیل مسئله P_0 تنها معرفی یک معمای ریاضی به‌نظر برسد. نکته بسیار جذاب این است که با درک بهتر مسئله P_0، ابعاد بیش‌تری نمایان می‌گردد. در واقع این مسئله و انواع آن نقش اساسی در پردازش سیگنال و تصویر ایفا می‌نمایند. یک مدل‌سازی جهانی که در اداره کاربردهای واقعی به خدمت می‌آیند.

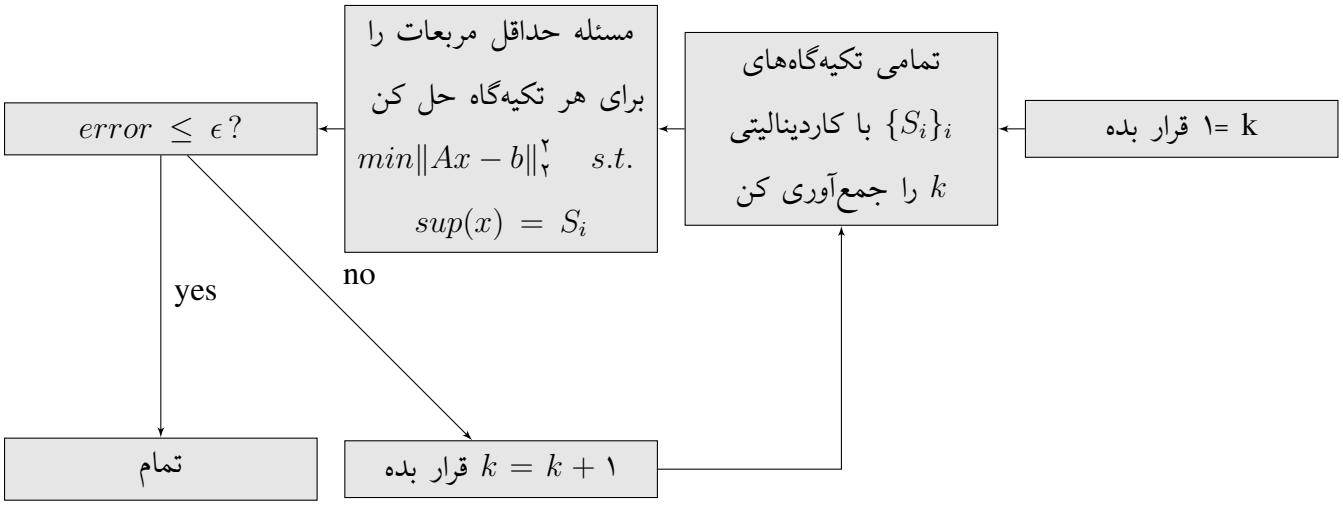
در مدل‌سازی سیگنال، یک ساختار ریاضی برای توصیف منابع سیگنال اعمال می‌گردد. فرض بر این است که سیگنال b بوسیله ضرب یک ماتریس ثابت A در بردار تنک x بدست می‌آید. در این عملیات ضرب، فرض بر این است که سیگنال b ترکیب خطی تعداد بسیار کمی از ستون‌های A است. به اصطلاح فنی b از ترکیب خطی تعداد بسیار کمی از اتم‌های دیکشنری A تشکیل می‌شود. این مدل یک فرضیه ساده‌کننده روی سیگنال مورد نظر اعمال می‌کند. با وجود این‌که این سیگنال‌ها بردارهایی با بعد n می‌باشند، بسیار ساده‌تر هستند و ساکن اجتماع تعداد زیادی زیرمجموعه کم‌بعد. جایی که هر زیرمجموعه به انتخاب متفاوتی از s ستون از A اشاره دارد ($s \ll n$). می‌دانیم که سیگنال‌هایی که روی آن‌ها عملیات انجام می‌دهیم، دارای ساختارهایی هستند. چراکه امکان فشرده‌سازی سیگنال‌ها وجود دارد. بدین معنا که محتوای واقعی اطلاعات از بعد فضایی واقعی آن‌ها بسیار کوچک‌تر است. ایده تنکی ضریب تبدیل در پردازش سیگنال و تصویر محوریت دارد و در بسیاری از امور استفاده می‌شود. ضرب A با یک بردار نمایش تنک نمایان‌گر تبدیل معکوس در نگرش سنّتی‌ست.

همان‌طور که گفته شد، تعداد بیش‌تر ستون‌ها نسبت به سطرها، خاصیت زائدی نامیده می‌شود. با استفاده از خاصیت زائدی در ماتریس، مدل غنی‌سازی می‌گردد چرا که زیرفضاهای بیش‌تری برای توصیف سیگنال وجود خواهد داشت. همچنین با داشتن دیکشنری عریض‌تر، جواب تنک‌تری هم خواهیم داشت. در نتیجه جستجو برای تنک‌ترین جواب یک سیستم خطی بسیار بیش‌تر از یک گریم ریاضیاتی است.

۱-۵ الگوریتم‌های جستجو (پیگرد)

در این بخش الگوریتم‌های مختلف برای حل مسئله P_0 را معرفی و مورد تحلیل قرار می‌دهیم. این الگوریتم‌ها معمولا با نام الگوریتم‌های جستجو [۱] شناخته می‌شوند. با تعریف هدف و توصیف مسیر پیش‌رو شروع می‌کنیم. مسئله P_0 در ۱-۱۵ تعریف شد. این مسئله‌ای‌ست که به عنوان هدف برای یافتن تنک‌ترین جواب سیستم خطی $Ax = b$ مشخص شد. سعی بر این است که این مسئله به نوعی حل شود که در زمان معقول قابل انجام باشد.

یک روش جامع وجود دارد که در عمل قادر به حل P_0 می‌باشد. تعداد k عنصر غیرصفر در راه‌حل را مشخص می‌کنیم و در میان مقادیر ممکن k جستجو می‌کنیم. با $k = 1$ شروع می‌کنیم. باید تمامی گزینه‌های ممکن تکیه‌گاه‌ها [۲] با کاردینالیتی [۳] k را بررسی کنیم و برای هریک امکان تولید جواب را بیازماییم. برای مقدار عمومی k، تعداد $\binom{m}{k}$ چنین آزمون‌هایی برای اجرا وجود دارد. یک روش آزمایش بر روی تکیه‌گاه انتخاب شده، استفاده از مسئله حداقل مربعات [۴] است. اگر خطا حداقل مربعات صفر یا نزدیک به صفر باشد، یک جواب پیدا شده‌است. اگر هیچ یک از این آزمون‌ها با موفقیت انجام نشود، k را یک واحد افزایش می‌دهیم و پروسه را تکرار می‌کنیم. اگر هر یک از این مسائل حداقل مربعات ۱ نانوثانیه زمان ببرد، این



شکل ۱-۸: فلوچارت حل مسئله P_0 با استفاده از حداقل مربعات

جستجو میلیون‌ها سال طول می‌کشد و با یک مسئله ترکیباتی دشوار با پیچیدگی ان‌پی-سخت [۵] مواجه هستیم. واضح است نمی‌توان از این روش جامع استفاد کرد. جایگزین مناسب الگوریتم‌های تخمینی [۶] هستند. بدین معنا که حاضر به سازش هستیم و به جواب نسبتا بهینه برای یک پروسه کاربردی قناعت می‌کنیم.

یک رویکرد خانواده الگوریتم‌های حریصانه [۷] می‌باشد. ایده این الگوریتم‌ها بسیار ساده است و بر این اساس بنا شده است که مفهوم حقیقی حل مسئله P_0 یک طبیعت حریصانه برای شناسایی تکیه‌گاه مناسب جواب دارد. در واقع روش جامعی که معرفی شد، در میان کل احتمالات جستجو می‌کند و همه گزینه‌ها را بررسی می‌نماید. اگر کلیه جواب‌های ممکن ماهیت

[۱] Pursuit [۲] Support [۳] Cardinality [۴] Least Square [۵] NP-hard [۶] Approximation [۷] Greedy

درختی داشته‌باشد، در روش‌های تخمینی درخت گزینه‌ها را هرس می‌کنیم و در نتیجه بسیاری از احتمالات حذف می‌شوند. با این امید که احتمال کم‌تری وجود دارد که موارد حذف شده یک جواب درست باشند.

P_0 چالش مهمی را نمایان می‌سازد، چراکه نرم L_0 به شدت غیرهموار است. لذا رویکرد واضح این است که به نوعی هموارسازی صورت گیرد و بعد الگوریتم‌های بهینه‌سازی رایج اعمال شوند. به این صورت خانواده الگوریتم‌های آزادسازی [۱] مطرح می‌شوند.

بنابراین با هدف تخمین جواب P_0، با دو رویکرد بسیار متفاوت مواجه می‌شویم. روش‌های حریصانه که بر طبیعت گسسته مسئله و ساخت تکیه‌گاه تاکید دارند و همچنین روش‌های آزادسازی که P_0 را هموار می‌سازند تا برای بکارگیری روش‌های بهینه‌سازی مطبوع گردد. در ادامه رویکرد حریصانه مورد بحث قرار می‌گیرد.

۱-۵-۱ الگوریتم جستجوی تطابقی

اخیرا روش‌های جستجوی تطابقی [۲] برای اهداف نویز زدایی سیگنال در نظر گرفته‌شده‌است. جستجوی تطابقی می‌تواند با یک تخمین ارزیابی حریصانه حداقل مربعات به عنوان یک ترکیب خطی اجزا (اتم‌های) یک دیکشنری، یک سیگنال تمیز را از یک نمونه نویزدار بازسازی نماید. این رویکرد بازسازی از محاسبات حجیم راه‌حل حداقل‌مربعات برای مسائل بزرگ جلوگیری می‌نماید، همچنین عملکرد بهبودیافته در زمینه نویز زدایی خصوصا در زمانی که سیگنال سالم تنک باشد بروز می‌دهد. [۱۶]

جستجوی تطابقی اصلی به‌مانند سایر الگوریتم‌های نوع جستجوی تطابقی همچون جستجوی تطابقی متعامد [۳] مسئله بازسازی را با روش انتخاب‌های قطعی اتم‌ها از دیکشنری در هر گام تکرار حل می‌کند. تحقیقات اخیر نسخه‌های تصادفی جستجوی تطابقی و جستجوی تطابقی متعامد که انتخاب تصادفی اتم‌ها را به‌کار می‌گیرند و پتانسیل بهبود بیش‌تر عملیات نویز زدایی را دارا هستند. این روش ابتدا توسط ژانگ و ملت [۴] [۱۵] مطرح شد . در این روش در هر مرحله تنها ضریب یکی از اتم‌ها مشخص می‌شود. در هر مرحله یک اتم دیکشنری که بیشترین شباهت(بزرگ‌ترین ضرب داخلی) را به داده آزمون دارد، بعنوان عضو فعال در ترکیب خطی در نظر گرفته شده، ضریب مربوط به آن محاسبه می شود. در مرحله بعد، باقی‌مانده سیگنال آزمون و اتم نخست با بقیه اتم‌ها مقایسه شده و دوباره مشابه‌ترین اتم انتخاب می‌شود. یعنی تفاضل حاصل ضرب این تقریب ۱تُنُک در دیکشنری از داده آزمون را به عنوان باقی مانده در نظر گرفته ، مراحل فوق تکرار می شوند. در هر مرحله جمع تقریب های ۱-تُنُک به دست آمده با تقریب های قبلی به عنوان تقریب جدید در نظر گرفته می شود و این روند تاجایی ادامه می یابد که یا تعداد مراحل مشخصی طی شود و یا خطا از مقدار معینی کمتر شود.

فرض کنید $b_{n\times 1} = A_{n\times m}x_{m\times 1}$ و x یک بردار k تُنُک است. در این صورت b را می‌توان بصورت ترکیب خطی k تا از ستون‌های A تلقی کرد و عمل بازسازی، معادل با یافتن این k ستون و ضرایب آن‌ها در ترکیب خطی است. در این روش ابتدا ستون‌های مورد استفاده از A در ترکیب خطی آشکار می‌شوند [۵] و سپس به کمک حل یک مسئله حداقل مربعات،

---

[۱] Relaxation [۲] Matching Pursuit (MP) [۳] Orthogonal Matching Pursuit (OMP) [۴] Malat , Zhang [۵] Support recovery

ضرایب این ستون‌ها که همان مقادیر ناصفر بردار x هستند محاسبه می‌شوند. در هر مرحله از روش جستجوی تطابقی(MP) یکی از ستون‌های A به عنوان عضوی فعال در ترکیب خطی آشکار می‌شود، در این مرحله ضرب داخلی b با تمام ستون‌های A محاسبه می‌شود و ستونی که بیشترین اندازه را حاصل کند (که بیشترین شباهت را به b دارد) به عنوان عضو فعال شناسایی می‌شود و مقدار ضرب داخلی که به اندازه این ستون تقسیم شده را به عنوان ضریب آن لحاظ می‌کنیم.

اکنون تقریبی ۱-تُنُک از بردار اصلی بدست آورده‌ایم که در این مرحله انتخاب بهینه به شمار می‌رود $(\hat{x}^{(1)}_{m\times 1})$. برای ادامه، با فرض اینکه محاسبات تا این لحظه صحیح است، بردار $r_{n\times 1} = b_{n\times 1} - A_{n\times m}\hat{x}^{(1)}_{m\times 1}$ را به عنوان باقی‌مانده [1] در نظر می‌گیریم و تمامی مراحل قبلی را طی می‌کنیم تا تخمین ۱-تُنُک جدیدی بدست آید. اکنون حاصل جمع این بردار ۱-تُنُک جدید و $(\hat{x}^{(1)}_{m\times 1})$ را به عنوان تقریب جدید برای $(\hat{x}^{(2)}_{m\times 1})$ تلقی می‌کنیم و این مراحل را مجددا تا رسیدن به شرط پایانی (نُرم باقی‌مانده کوچک یا تعداد مرحله معلوم و یا ترکیبی از هر دو) ادامه می‌دهیم. الگوریتم MP به دلیل اینکه در هر مرحله نیاز به یک جستجوی ساده دارد، بسیار سریع است، اما به دلیل حریصانه بودن، تضمینی وجود ندارد که پاسخ نهایی مشابه با پاسخ تُنُک باشد. بمنظور بهبود این الگوریتم با حفظ سرعت بالا، الگوریتم‌های دیگری پیشنهاد شده است که در ادامه به شرح و توضیح آن‌ها پرداخته شده است.

۱-۵-۲ جستجوی تطابقی متعامد

ایده این الگوریتم به این صورت است که برای سیستم خطی $Ax = b$، به دنبال یک راه‌حل تنک هستیم. با عملکرد ترتیبی و تدریجی تکیه‌گاه S را خواهیم یافت. با جستجو بهترین جواب ممکن با تنها یک تکیه‌گاه تک اتمی شروع می‌کنیم. به این صورت که در میان همه احتمالات می‌گردیم و ستونی در A را می‌یابیم که بهترین تطابق را با $Ax = b$ داشته‌باشد. با پیدا شدن چنین اتمی، این ستون را نگه می‌داریم و برای اضافه کردن اتم دوم به آن جستجو را ادامه می‌دهیم. پس دوباره تمامی احتمالات $A - ۱$ را جستجو می‌کنیم تا دومین اتمی که بیش‌ترین مطابقت را داراست بیابیم. به این صورت تکیه‌گاه هربار با یک عنصر غیرصفر رشد می‌کند. الگوریتم زمانی متوقف می‌شود که $Ax - b$ به اندازه کافی به صفر نزدیک شود. یعنی Ax و b به نزدیک‌ترین حد ممکن برسند. البته شرط‌های توقف دیگری هم وجود دارد.

از دیدگاه درختی که ، با یک تکیه‌گاه خالی شروع می‌کنیم. m انتخاب برای اولین اتم داریم که بهترین جواب ممکن با کاردینالیتی یک را خواهیم داشت $(\|x\|_{\circ} = ۱)$. به جای بررسی تمامی انتخاب‌ها با کاردینالیتی ۲، تنها گزینه‌هایی را بررسی می‌کنیم که مبتنی بر جواب پیدا شده هستند. این پروسه تا زمانی که تخمین قابل‌قبولی بدست آید، ادامه می‌یابد. به این صورت به جای اینکه تعداد زیادی آزمون با پیچیدگی ترکیبی انجام شود، آزمون‌های با مرتبه m برای تکمیل الگوریتم اعمال می‌شود. مسیر مشخص شده در شکل ۱-۹ نشان دهنده رشد تکیه‌گاه با یک عضو غیرصفر در هر مرحله است. الگوریتم‌های حریصانه متعددی برپایه این منطق قابل ارائه هستند. همانند سایر روش‌های حریصانه، جستجوی تطابقی متعامد نیز رشته‌ای از جواب‌ها را با تکیه‌گاهی که تدریجی رشد می‌کند، تولید می‌نماید. این جواب‌ها را با x_k نشان می‌دهیم. این روش ارائه شده از معادله $Ax = b$ تبعیت نمی‌کند. باید بردار خطا $b - Ax_k$ را به صورت r_k نشان دهیم که به عنوان بردار باقیمانده

[1] Residual

الگوریتم ۱-۱ الگوریتم جستجوی تطابقی برای تخمین پاسخ مسئله

$(P_0) : \min_x \|x\|_0 \quad subject\,to \quad Ax = b$

Require: We are given the matrix A, the vector b, and the error threshold ε_0

// Initialize

$k = 0$,

The initial solution $x_0 = 0$,

The initial residual $r_0 = b - Ax_0 = b$

The initial solution support $S_0 = Support\{x_0\} = \varnothing$

// Main Iteration

Increment k by 1 and perform the following steps:

Compute the errors $\varepsilon(j) = \min_{z_j} \|a_j z_j - r_{k-1}\|_2^2$ for all j using the optimal choice $z_j^* = a_j^\dagger r_{k-1} / \|a_j\|_2^2$

//Update Support

Find a minimizer,j_0 of $\varepsilon(j) : \forall\, 1 \le j \le m,\ \varepsilon(j_0) \le \varepsilon(j)$, and update $S_k = S_{k-1} \cup \{j_0\}$.

//Update Provisional Solution

Set $x_k = x_{k-1}$, and update the entry $x_k(j_0) = x_k(j_0) + z_j^*$

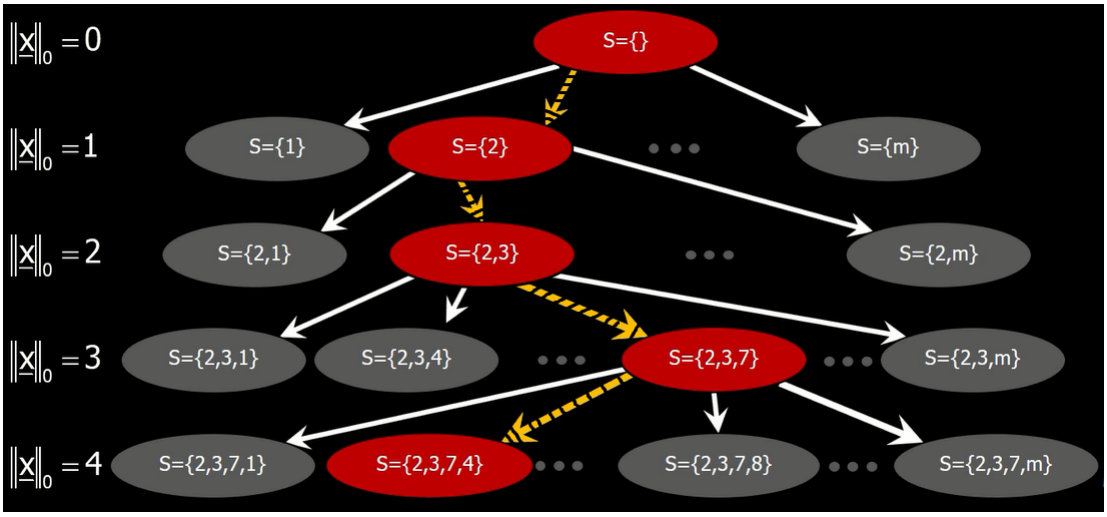
//Update Residual

Compute $r_k = b - Ax_k = r_{k-1} - z_{j_0}^* a_{j_0}$

//Stopping Rule

If $\|r_k\|_2 < \varepsilon_0$, stop. Otherwise, apply another iteration.

return The proposed solution is x_k obtained after k iterations



شکل ۱-۹: تحلیل درختی الگوریتم OMP

الگوریتم ۱-۲ الگوریتم جستجوی تطابقی متعامد برای تخمین پاسخ مسئله
$(P_0) : \min_x \|x\|_0 \ \ subject\,to \ \ Ax = b$.

Require: We are given the matrix A, the vector b, and the error threshold ε_0

// Initialize

$k = 0$, and set

The initial solution $x_0 = 0$,

The initial residual $r_0 = b - Ax_0 = b$

The initial solution support $S_0 = Support\{x_0\} = \varnothing$

//Main Iteration

Increment k by 1 and perform the following steps:

Compute the errors $\varepsilon(j) = \min_{z_j} \|a_j z_j - r_{k-1}\|_2^2$ for all j using the optimal choice $z_j^* = a_j^\top r_{k-1} / \|a_j\|_2^2$

//Update Support

Find a minimizer,j_0 of$\varepsilon(j) : \forall\, j \notin S_{k-1},\ \varepsilon(j_0) \le \varepsilon(j),$ and update $S_k = S_{k-1} \cup \{j_0\}$.

//Update Provisional Solution

Compute x_k, the minimizer of $\|Ax - b\|_2^2$ subject to $Support\{x\} = S_k$

//Update Residual:

Compute $r_k = b - Ax_k$

//Stopping Rule

If $\|r_k\|_2 < \varepsilon_0$, stop. Otherwise, apply another iteration.

return The proposed solution is x_k obtained after k iterations

شناخته می‌شود.

$$r_k = b - Ax_k \qquad (۱۶-۱)$$

نکته اصلی در جستجوی تطابقی متعامد، استفاده از باقیمانده در هر مرحله جهت انتخاب اتم بعدی است. این کار به گونه‌ای انجام می‌شود که اتم انتخاب شده به مقدار کاهش بیشینه در انرژی باقیمانده منجر شود. چون مقدار x_0 صفر است، باقیمانده با بردار b شروع می‌شود. x با اضافه شدن یک عضو غیر صفر بروزرسانی می‌شود (x_1)، انرژی r_1 کم‌تر می‌شود. به همین ترتیب با اضافه کردن یک غیرصفر به جواب در هر تکرار ادامه می‌دهیم تا زمانی که باقیمانده به صفر برسد یا به اندازه کافی کوچک شود.

۱-۵-۳ جزئیات الگوریتم جستجوی تطابقی متعامد

ابتدا مقداردهی اولیه را به صورت زیر انجام می‌دهیم:

$$k = ۰, \quad x_۰ = ۰, \quad r_۰ = b - Ax_۰ = b \quad and \quad S_۰ = \{\}$$

k را یک واحد افزایش داده، سپس مراحل زیر را انجام می‌دهیم:

۱. با داشتن باقیمانده $r_{k-۱}$، برای یافتن بهترین ستون در A جستجو می‌کنیم به طوری که با ضرب شدن در یک اسکالر z، کم‌ترین اختلاف $L_۲$ از باقیمانده را داشته‌باشد:

$$Compute \quad \varepsilon(j) = \min_z \|z.a_j - r_{k-۱}\|_۲^۲ \quad for\ ۱ \le j \le m$$

۲. با فرض انجام تمامی m آزمون و محاسبه همه مقادیر خطا $\varepsilon(j)$، بهترین اتم برای انتخاب، اتم با کم‌ترین مقدار خطاست با این فرض که این اتم $i_۰$ باشد:

$$Choose \quad j_۰ \ s.t. \forall ۱ \le j \le m, \ \varepsilon(j_۰) \le \varepsilon(j)$$

۳. حالا اندیس $i_۰$ به تکیه‌گاه s_k اضافه می‌شود:

$$Update\ S_k\ :\ S_{k-۱} \cup \{j_۰\}$$

۴. ضرایب حقیقی x در محل انتخاب شده تکیه‌گاه که کوچک‌ترین خطای $L_۲$ بین Ax_k و b است را بروزرسانی می‌کنیم. این عملیات در واقع یک پروسه حداقل مربعات ساده است که نتیجه آن یک x_k بروز شده است.

$$LS: \quad x_k = \min_x \|Ax - b\|_۲^۲ \quad s.t.\ sup\{x\} = S_k$$

۵. آخرین مرحله بروزرسانی باقیمانده r_k است:

$$r_k = b - Ax_k$$

اگر باقیمانده بدست آمده به اندازه کافی کوچک باشد، الگوریتم متوقف می‌شود. در غیر اینصورت یک واحد به k اضافه می‌کنیم و مراحل بالا را مجدد تکرار می‌کنیم. اکنون برای فهم بهتر، هر دو گام اساسی این الگوریتم را با جزئیات بیش‌تری

شرح می‌دهیم.

۱-۵-۳-۱ انتخاب اتم بعدی

فرض بر این است که ستون‌های A، L_2-نرمالیزه هستند. با نگاه دقیق‌تر به محاسبه $\varepsilon(j)$، بهینه‌سازی با توجه به اسکالر z که در اتم a_j ضرب شده انجام می‌شود.

$$\forall\, 1 \le j \le m \quad \varepsilon(j) = \min_z \|z.a_j - r_{k-1}\|_2^2 \tag{۱-۱۷}$$

مقدار بهینه z توسط یک مشتق‌گیری ساده از عبارت L_2 در ۱-۱۷ بدست می‌آید.

$$a_j^\top (z.a_j - r_{k-1}) = 0 \implies z_{opt} = \frac{a_j^\top r_{k-1}}{a_j^\top a_j} = a_j^\top r_{k-1} \tag{۱-۱۸}$$

در ۱-۱۸ ملاحظه می‌شود که مخرج کسر بدلیل نرمالیزه بودن اتم‌ها حذف شده‌است.

با وارد کردن عبارت بدست آمده برای z_{opt} در ۱-۱۸ به جای z در ۱-۱۷ و اعمال چند مرحله ساده‌سازی جبری، $\varepsilon(j)$ برابر با مربع نرم باقیمانده منهای مربع ضرب داخلی a_j و r_{k-1} خواهد بود.

$$\varepsilon(j) = \left\| a_j^\top r_{k-1} \,.\, a_j - r_{k-1} \right\|_2^2 = \|r_{k-1}\|_2^2 - (a_j^\top r_{k-1})^2 \tag{۱-۱۹}$$

در نتیجه به جای کمینه کردن $\varepsilon(j)$، می‌توان عبارت $|a_j^\top \,.\, r_{k-1}|$ را بیشینه کرد. این بدین معناست که انتخاب اتم بعدی در الگوریتم جستجوی تطابقی متعامد می‌تواند به این صورت باشد که $A^\top$ در باقیمانده r_{k-1} ضرب شود. مقدار بیشینه قدرمطلق بردار جواب این ضرب با m عضو به اتم انتخاب شده اشاره دارد.

۱-۵-۳-۲ حداقل مربعات

با تمرکز بر مرحله بروزرسانی S_k که محاسبه حداقل مربعات روی بخشی از تمام بردار x می‌باشد، با داشتن تکیه‌گاه S_k تنها ستون‌های انتخاب شده در A و ورودی‌های متناظر این ستون‌ها در بردار x استخراج می‌شوند. ماتریس جدید را A_S می‌نامیم. در نتیجه بار محاسباتی حداقل مربعات به k مجهول ساده‌سازی می‌شود و جواب با ضرب شبه معکوس A_S در b مشخص می‌شود.

$$x_k = \min_x \|Ax - b\|_2^2 \;\; s.t. \;\; sup\{x\} = S_k$$

$$\min_x \|A_S x - b\|_2^2 \implies x_k = (A_S^\top A_S)^{-1} A_S^\top b = A_S^\top b$$

این سوال بوجود می‌آید که چرا نام این الگوریتم "جستجوی تطابقی متعامد" است؟ عبارت "تطابقی" مربوط به همبستگی [۱] است که میان باقیمانده و اتم‌ها در A برای پیدا کردن اتم بعدی اعمال می‌شود. اما چرا عبارت "متعامد" ؟ در مسئله حداقل مربعات، درباره امکان یافتن جواب بهینه بوسیله مشتق خطای L_2 نسبت به x بحث شد:

$$\min_x \|A_S\, x - b\|_2^2 \implies A_S^\top (A_S\, x_k - b) = 0 \qquad (۲۰-۱)$$

عبارت $(A_S\, x_k - b)$ در رابطه ۱-۲۰ در واقع همان باقیمانده بروز شده است (r_k). بنابراین بعد از بروزرسانی جواب S_k، ضرب داخلی باقیمانده جدید با اتم‌های تکیه‌گاه متعامد است. فایده این متعامد بودن در این است که وقتی یک اتم انتخاب می‌شود، هیچ‌گاه دوباره انتخاب نخواهد شد. چون ضرب داخلی آن با باقیمانده صفر خواهد بود.

در مورد حداقل مربعات که S_k را بروزرسانی می‌کند، یک میانبر عددی موثر وجود دارد. راه‌حل منظم شامل معکوس ماتریس گرم [۲] به اندازه $k \times k$ است که برای ماتریس A_{S_k} محاسبه می‌شود:

$$\min_x \|A_{S_k} x - b\|_2^2 = (A_{S_k}^\top A_{S_k})^{-1} A_{S_k}^\top b \qquad (۲۱-۱)$$

اگر چه یک مرحله قبل معکوس گرم ماتریس مشابهی با اندازه $(k-1) \times (k-1)$ برای $A_{S_{k-1}}$ یا یک اتم کم‌تر محاسبه شده‌است. لذا می‌توان از نتیجه قبلی برای جواب جدید استفاده کرد. یک ستون جدید برای ساخت A_{S_k} اضافه شده‌است. به همراه آن یک مقدار اسکالر ناشناخته نیز به بردار x اضافه شده‌است. یک روش بازگشتی برای حل دنباله افزایشی از مسائل حداقل مربعات وجود دارد که بیان می‌دارد نیازی به محاسبه معکوس هیچ ماتریسی در جستجوی تطابقی متعامد وجود ندارد.

در یک جمع‌بندی، الگوریتم جستجوی تطابقی متعامد شامل دو گام اصلی محاسباتی‌ست. اولین گام جستجو برای اتم بعدی برای اضافه شدن است که یک عملیات ضرب دیکشنری در باقیمانده $A^\top r_{k-1}$ که نیاز به محاسبات $O(mn)$ دارد. دومین گام محاسبه حداقل مربعات است که x_k را با محاسبه $A_s^\top A_s$ بروزرسانی می‌کند که بار محاسباتی $O(k^2 m)$ را دارد که البته روش‌های میانبر وجود دارد. در نتیجه میزان کلی محاسبات مورد نیاز برای جستجوی تطابقی متعامد دارای پیچیدگی $O(mnk)$ است. k در این‌جا کاردینالیتی جواب نهایی است.

۱-۶ معرفی اجمالی انواع جستجو تطابقی

نویز زدایی مدت‌هاست یک مسئله مهم در زمینه پردازش سیگنال می‌باشد. درون یک محیط داده‌شده، نمونه‌های یک سیگنال می‌تواند بوسیله نویز مورد اختلال واقع شود که بدون دسترسی به سیگنال اصلی و سالم، نویز زدایی موثر برای استخراج اطلاعات صحیح از نمونه‌ها حیاتی‌ست. [۶] برای مسائل بزرگ‌مقیاس وقتی روش‌های تخمینی مستقیم مانند حداقل مربعات

[۱] Correlation [۲] Gram matrix

استفاده می‌شود، نویز زدایی نیاز محاسباتی بالایی می‌خواهد. نیاز به روش‌های محاسباتی دقیق، خصوصا در کاربردهای کم‌قدرت سیار و محاسبات سطح بالا [۱] در تحقیقات، به استفاده از رویکردهای حریصانه همچون جستجوی تطابقی منجر شده‌است. جستجوی تطابقی به‌طور حریصانه راه‌حل حداقل مربعات را با بازسازی مکرر یک سیگنال از اجزاء (اتم‌های) منتخب یک دیکشنری به صورت انتخاب یک اتم در هر تکرار تخمین می‌زند. با پرهیز از معکوس‌سازی‌های ماتریسی مورد نیاز در روش حداقل مربعات، جستجوی تطابقی نیاز به محاسبات کم‌تری دارد و همچنین سادگی ریاضیاتی این روش آن را برجسته می‌کند.

محبوبیت ابتدایی جستجوی تطابقی موجب دست‌یابی به دیگر الگوریتم‌های مبتنی بر امپی مانند جستجوی تطابقی متعامد گردید. در زمینه نویز زدایی، الگوریتم‌های مبتنی بر امپی نه تنها با ساده‌سازی ملزومات محاسباتی، بلکه با بهبود عملکرد نویز زدایی، خصوصا در مورد سیگنال‌های تنک نتایج محیجی بروز می‌دهند. برخی الگوریتم‌های مبتنی بر امپی همچون CoSaMP و جستجوی تطابقی متعامد تنظیم‌شده [۲] به مسائل بازسازی سیگنال و سنجش فشرده [۳] که مرتبط با نویز زدایی هستند اعمال شده‌اند. این الگوریتم‌ها توسعه‌یافته الگوریتم‌های ابتدایی امپی و جستجوی تطابقی متعامد بوسیله معرفی تغییراتی در جهت حل مسائل مورد نظر می‌باشند. برای مثال، StOMP الگوریتم جستجوی تطابقی متعامد را به جای یک اتم برای انتخاب چندین اتم در هر تکرار در جهت تسریع بازسازی سیگنال اصلاح می‌کند.

تحقیقات بیش‌تر در زمینه نویز زدایی مبتنی بر جستجوی تطابقی انتخاب تصادفی اتم‌ها از دیکشنری را به جای انتخاب قطعی در الگوریتم‌های اولیه ارائه می‌دهد. این رویکردهای جستجوی تطابقی تصادفی و جستجوی تطابقی متعامد تصادفی که به عنوان RMP و ROMP شناخته می‌شوند، پتانسیل بهبود بیش‌تر عملکرد نویز زدایی را دارا هستند. مثال‌هایی از الگوریتم‌های مبتنی بر جستجوی تطابقی با انتخاب تصادفی اتم‌ها شامل جستجوی تطابقی متعامد تصادفی [۴] ، جستجوی تطابقی احتمالاتی [۵] و جستجوی تطابقی متعامد احتمالاتی [۶] می‌باشد. معمولا این الگوریتم‌های جدیدتر ایده‌های ارائه شده قبلی را ترکیب می‌کنند مثل RROMP ، که انتخاب تصادفی اتم‌ها را در RandOMP با انتخاب چندین اتم در هر تکرار در StOMP ترکیب می‌کند.

علاوه بر این، در زمینه نویز زدایی کارآمد، الگوریتم‌های مبتنی بر جستجوی تطابقی پتانسیل قابل توجهی نشان می‌دهند. محاسبات موازی به تازگی اهمیت بیش‌تری در به‌کارگیری فناوری موازی موجود در جهت بهبود عملکرد یافته است. یک ابزار موازی واحد پردازنده گرافیکی [۷] است، که اخیرا به یک پردازنده موازی همه‌منظوره [۸] مبدل شده‌است. جی‌پی‌یوهای همه‌منظوره و در کل روش پردازش گرافیکی می‌تواند برای نویز زدایی موازی به‌کار گرفته‌شود. در فصل‌های آینده درباره مفاهیم پایه و انواع روش‌های موازی‌سازی بحث خواهد شد و در ادامه سعی خواهیم کرد این روش‌ها را به صورت عملی در متلب و Visual Studio پیاده‌سازی نماییم.

---

[۱] High Performance Computations (HPC) [۲] Regularized OMP [۳] Compressed Sensing [۴] RandOMP [۵] Probabilistic MP (PMP) [۶] PROMP [۷] Graphical Processing Unit (GPU) [۸] General Purpose GPU

فصل ۲

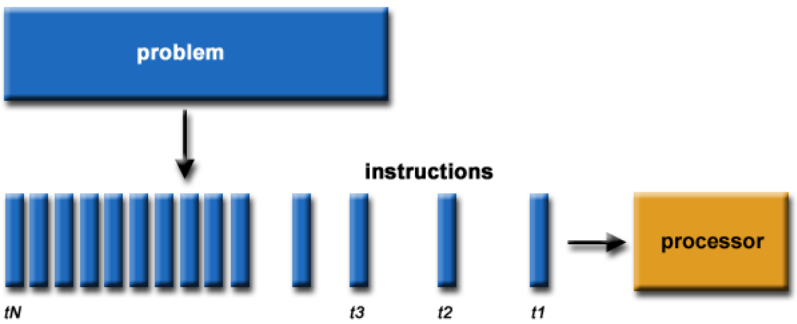
موازی‌سازی

۲-۱ مقدمه

در این فصل مفاهیم پایه و اساسی محاسبات موازی معرفی می‌گردد. درباره معماری پردازنده مرکزی و پردازنده گرافیکی بحث می‌شود و این دو معماری با هم مقایسه می‌شوند. محاسبات ناهمگون [۲] به عنوان یک روش نوین در برنامه‌نویسی معرفی می‌گردد. معماری و ساختار پردازنده‌های گرافیکی دارای قابلیت کودا (CUDA) [۳] شرح داده می‌شود و در پایان مقدماتی درباره برنامه‌نویسی کودا و مفاهیم آن عنوان شده‌است. منابع اصلی استفاده شده در این فصل، [۱۷] و [۱۸] می‌باشد.

۲-۲ معرفی محاسبات موازی

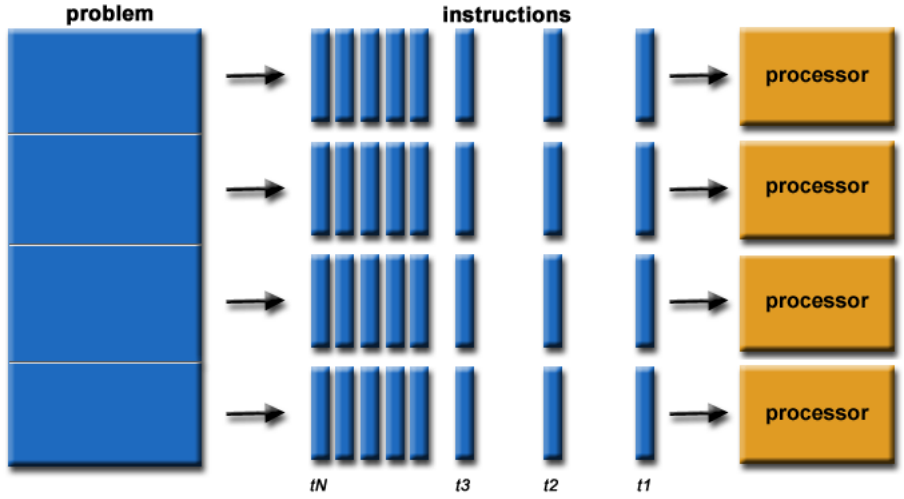
یک برنامه که برای محاسبات سری نوشته‌شده، بر روی یک پردازنده واحد اجرا می‌شود. از این رو، مسئله به مجموعه‌ای از دستورالعمل‌های گسسته تقسیم می‌شود که به صورت ترتیبی یکی پس از دیگری اجرا می‌شوند. از طرف دیگر، برنامه‌ای



شکل ۲-۱: محاسبات سری

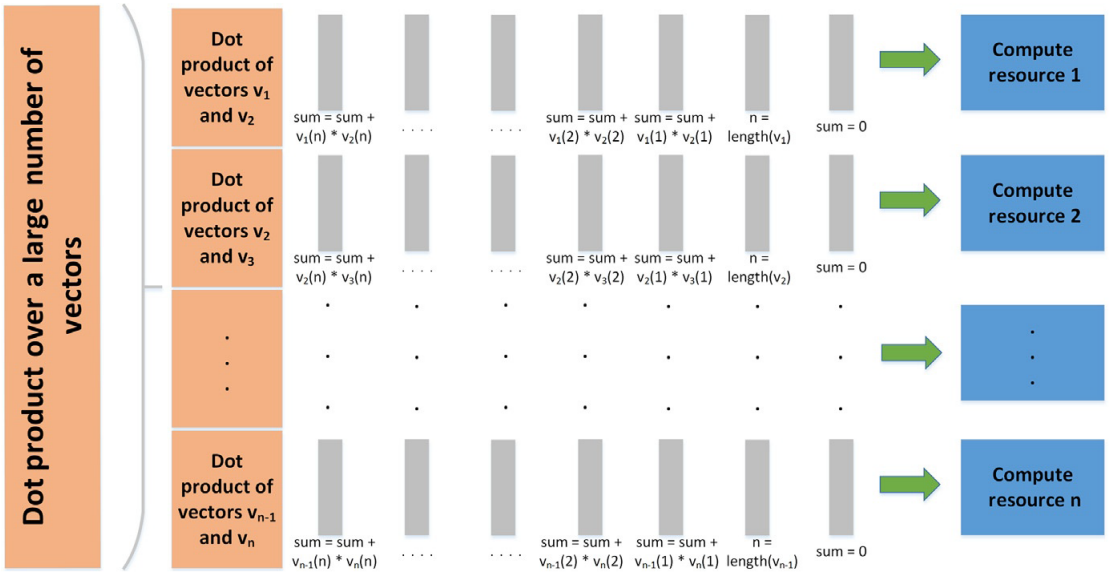
[۲] Heterogeneous Parallel Computing [۳] Compute Unified Device Architecture

که برای محاسبات موازی نوشته شده‌است، روی چندین منبع محاسباتی اجرا می‌گردد. از این رو، مسئله به قطعات گسسته تقسیم می‌شود که هر یک امکان اجرا به صورت هم‌زمان را دارند و قطعات به یک سری دستورالعمل‌های بیش‌تر شکسته می‌شوند. دستورات در هر قطعه به صورت هم‌زمان روی منابع محاسباتی متفاوت اجرا می‌شوند. قطعا دستورات درون یک قطعه به صورت ترتیبی یکی پس از دیگری روی یک منبع محاسباتی خاص اجرا می‌شوند. به عنوان مثال اگر بخواهیم ضرب



شکل ۲-۲: محاسبات موازی

نقطه‌ای تعداد زیادی بردار را محاسبه کنیم، می‌توان ضرب نقطه‌ای هر دو بردار را در هر یک از منابع محاسبات انجام داد. که در شکل ۲-۳ ملاحظه می‌گردد.



شکل ۲-۳: اجرای موازی ضرب بردارها

منابع محاسباتی معمولا ممکن است هسته‌های یک رایانه با یک پردازنده باشند یا چند پردازنده در یک رایانه و یا پردازنده‌های رایانه‌های متفاوت متصل در شبکه باشند.

محاسبات موازی به طور کلی برای دلایل ذیل استفاده می‌شوند:

- **کاهش زمان اجرای یک برنامه :** استفاده از منابع محاسباتی بیش‌تر برای اجرای یک برنامه احتمالا زمان اجرای این برنامه را کاهش خواهد داد. اغلب این کاهش زمان اجرا موجب صرفه‌جویی در هزینه می‌گردد. نیاز به کاهش زمان اجرا برای برنامه های کاربردی در زمان واقعی [۱] که برنامه باید در بازه زمانی مشخصی اجرا شود، بسیار حائز اهمیت است.
- **اجرای وظایف بیش‌تر به صورت هم‌زمان :** یک منبع محاسباتی هر بار تنها می‌تواند یک وظیفه خاص را انجام دهد. استفاده از چندین منبع محاسباتی تعداد وظایفی که می‌توانند به صورت هم‌زمان اجرا شوند را افزایش می‌دهد.
- **حل مسائل بزرگ‌تر:** استفاده از منابع محاسباتی بیش‌تر امکان حل مسائل بزرگ‌تر که روی یک کامپیوتر با حافظه محدود قابل اجرا نیستند را فراهم می‌سازد.

گرچه، برخی مسائل وجود دارند که یا هرگز امکان موازی‌سازی ندارند یا موازی‌سازی آن‌ها بسیار پیچیده است. در ادامه این واقعیت به صورت دو مثال شرح داده می‌شود.

مثال ۲-۲-۱ (مسئله قابل موازی‌سازی). شمارش تعداد رخداد یک کلمه خاص در مقدار زیادی متن. این مسئله می‌تواند به صورت موازی حل شود. هر منبع محاسباتی تعداد رخدادها در بخشی از متن را می‌یابند و نتیجه هر یک جهت تعیین مقدار نهایی رخدادها در کل متن به اشتراک گذاشته می‌شود. مسائل این چنینی که هیچ وابستگی یا وابستگی اندکی میان وظایف وجود دارد، وظایف شدیدا موازی [۲] یا وظایف کاملا موازی [۳] نامیده می‌شوند.

مثال ۲-۲-۲ (مسئله غیرقابل موازی‌سازی). شمارش n عدد ابتدایی اعداد فیبوناچی (۰, ۱, ۱, ۲, ۳, ۵, ۸, ۱۳, . . .) با استفاده از رابطه $f(n) = f(n-۱) + f(n-۲)$، که $f(۰) = ۰$ و $f(۱) = ۱$ می‌باشد. این مسئله نمی‌تواند موازی‌سازی شود چرا که محاسبه اعداد فیبوناچی شامل محاسبات وابسته است. محاسبه $f(n)$ به محاسبه $f(n-۱)$ و $f(n-۲)$ وابسته است. این سه مقدار نمی‌توانند به صورت مستقل محاسبه شوند.

نکاتی وجود دارد که قبل از اقدام به موازی‌سازی یک برنامه سری باید رعایت شوند:

- باید مشخص شود که آیا مسئله امکان موازی‌سازی دارد یا خیر. باید بخش‌هایی از برنامه که امکان اجرای مستقل بدون ارتباط با سایر وظایف موازی را داشته‌باشند شناسایی شوند. اگر مشخص گردد که هیچ وظیفه مستقل وجود ندارد یا هزینه ارتباط میان وظایف موازی بسیار بالاست، باید به دنبال الگوریتم دیگری برای حل مسئله بود. برای مثال همان طور که در مثال ۲-۲-۲ گفته شد، محاسبه n عدد ابتدایی اعداد فیبوناچی نمی‌تواند با استفاده از رابطه $f(n) = f(n-۱)+f(n-۲)$ موازی شود. اگر چه، می‌توان از فرمول بینت [۴] برای محاسبه nامین عدد فیبوناچی به صورت $f(n) = \frac{(۱+\sqrt{۵})^n-(۱-\sqrt{۵})^n}{۲^n\sqrt{۵}}$ استفاده کرد. با استفاده از این فرمول، می‌توان به راحتی محاسبات n عدد ابتدایی اعداد فیبوناچی را به صورت موازی پیاده‌سازی نمود.

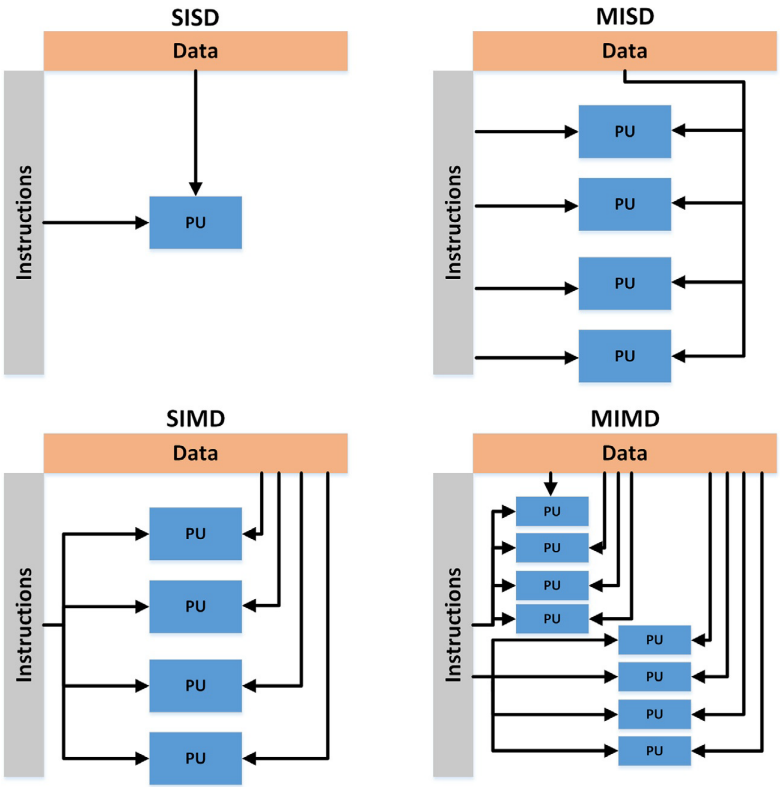
---

[۱] Real-time Applications [۲] mbarrassingly Parallel Tasks [۳] Perfectly Parallel Tasks [۴] Binet's Formula

- یافتن قسمت‌های کلیدی در برنامه با استفاده از پروفایلر [۱] و ابزارهای تجزیه و تحلیلی برای شناسایی بخش‌هایی از برنامه که بیش‌ترین وظایف زمانبر را برعهده دارند. ابزار پروفایلر متلب در فصل بعدی شرح داده خواهد شد.
- شناسایی تگناهایی که موجب ایجاد وقفه در وظایف قابل موازی‌سازی می‌گردد.
- استفاده از کتابخانه‌های موازی بهینه‌سازی شده شخص ثالث در صورت امکان

۳-۲ دسته‌بندی رایانه‌های موازی

بر اساس طبقه‌بندی فلین [۲] ، معماری چندهسته‌ای رایانه می‌تواند به دو بُعد دسته‌بندی گردد: ۱) جریان دستورات (I) [۳] و ۲) جریان داده (D) [۴] . هر یک از این ابعاد یک وضعیت از دو وضعیت ممکن را دارا هستند: ۱) منفرد (S) [۵] و ۲) چندگانه (M) [۶] . بنابراین چهار نوع رایانه وجود دارد:



شکل ۲-۴: دسته‌بندی رایانه‌های موازی

- اس‌آی‌اس‌دی (دستور منفرد داده منفرد) [۷] : یک رایانه ترتیبی که تنها واحد پردازش آن، یک دستور روی یک جریان داده منفرد در هر چرخه ساعت اجرا می‌کند. این معماری بسیار قدیمی است و قدیمی‌ترین کامپیوترها با یک

---

[۱] Profiler [۲] Flynn's Taxonomy [۳] Instruction Stream [۴] Data Stream [۵] Single
[۶] Multiple [۷] Single Instruction Single Data (SISD)

پردازنده/هسته از آن استفاده می‌کردند.

- اس‌آی‌ام‌دی (دستور منفرد چندین داده) [۱] : یک نوع رایانه موازی که واحدهای پردازشگر آن می‌توانند یک دستور منفرد را روی جریان‌های داده مختلف در هر چرخه زمان اجرا کنند. این معماری توسط آرایه‌های پردازشگر، خط لوله‌های برداری و بسیاری رایانه‌های امروزی مورد استفاده قرار می‌گیرد. همچنین واحد پردازنده گرافیکی (جی‌پی‌یو) نیز از این نوع معماری بهره می‌برد
- ام‌آی‌اس‌دی (چندین دستور داده منفرد) [۲] : نوعی رایانه موازی که واحدهای پردازش‌گر آن می‌توانند دستورات مختلفی روی جریان داده منفرد در هر چرخه ساعت اجرا کنند. این معماری بیش‌تر نظری است و هیچ مثال شناخته شده‌ای از رایانه‌های موازی که از آن استفاده کنند، وجود ندارد.
- ام‌آی‌ام‌دی (چندین دستور چندین داده) [۳] : نوعی از رایانه موازی که واحدهای پردازشگر آن می‌توانند دستورات مختلفی روی جریان داده‌های مختلف در هر چرخه زمانی اجرا کنند. این معماری توسط اکثر ابررایانه‌ها و رایانه‌های چندهسته‌ای استفاده می‌شود.

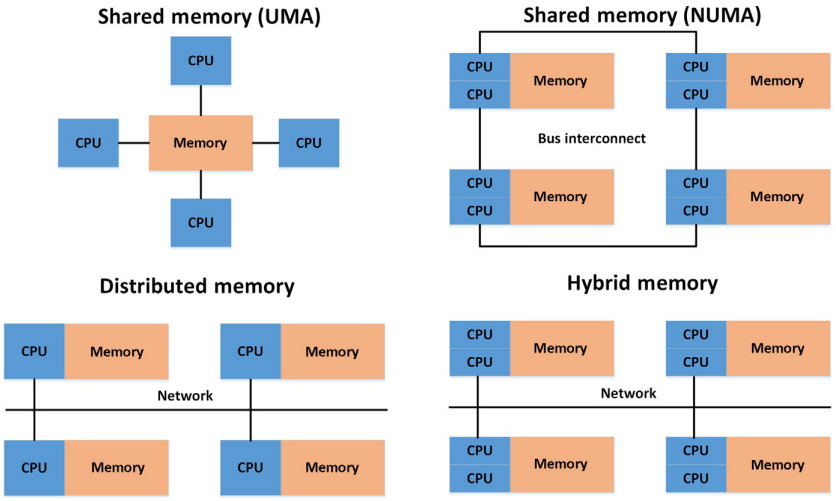
۴-۲ انواع معماری حافظه رایانه‌های موازی

سه نوع معماری برای حافظه رایانه‌های موازی وجود دارد:

- **حافظه اشتراکی** [۴] : پردازنده های چندگانه می توانند به طور مستقل دسترسی پیدا کنند اما حافظه را در رایانه های موازی مشترک به اشتراک می گذارند. اگر یک پردازنده تغییری در محلی از حافظه بدهد، این تغییر برای همه پردازنده های دیگر قابل مشاهده است.
- **حافظه توزیع‌شده** [۵] : همه پردازنده‌ها حافظه محلی خودشان را دارند و می‌توانند مستقلا به آن دسترسی داشته‌باشند. همه رایانه‌ها از طریق یک شبکه به هم متصل‌اند و می‌توانند از سایر رایانه‌ها درخواست داده داشته‌باشند (یک رایانه به حافظه رایانه دیگر دسترسی ندارد اما امکان انتقال داده از یک رایانه به دیگری با برنامه‌نویسی وجود دارد). نوع شبکه‌ای که برای اتصال کامپیوترها استفاده می‌شود، در سرعت انتقال داده تاثیرگذار است.
- **حافظه ترکیبی** [۶] : اکثر ابر رایانه‌های امروزی از نوعی حافظه ترکیبی استفاده می‌کنند که دو نوع حافظه اشتراکی و توزیع‌شده را ترکیب می‌نمایند. تمامی پردازنده‌های یک ماشین می‌توانند حافظه را میان پردازنده‌های همان ماشین به اشتراک گذارده و همچنین از سایر رایانه‌ها درخواست داده نمایند (یک رایانه به حافظه رایانه دیگر دسترسی ندارد اما امکان انتقال داده از یک رایانه به دیگری با برنامه‌نویسی وجود دارد).

---

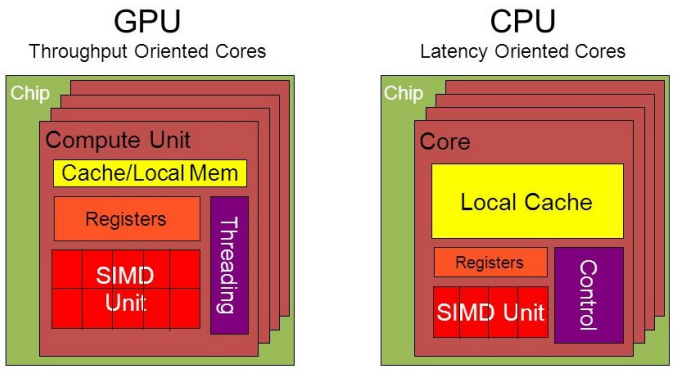
[۱] Single Instruction Multiple Data (SIMD) [۲] Multiple Instruction Single Data (MISD) [۳] Multiple Instruction Multiple Data (MIMD) [۴] Shared Memory [۵] Distributed Memory [۶] Hybrid Memory



شکل ۲-۵: معماری حافظه رایانه‌های موازی

۲-۵ محاسبات موازی ناهمگون

هدف از این بخش بیان تفاوت‌های اساسی میان ابزارهای تاخیری [۱] (هسته‌های پردازنده مرکزی) و ابزارهای عملیاتی [۲] (هسته‌های پردازنده گرافیکی) می‌باشد. همچنین در راستای درک بهتر علت استفاده از هر دو این ابزارها در برنامه‌های موفق توضیحاتی ارائه می‌شود.



شکل ۲-۶: تفاوت پردازنده گرافیک و پردازنده مرکزی

پردازنده‌های مرکزی و پردازنده‌های گرافیکی ساختار و معماری بسیار متفاوتی دارند. پردازنده‌های مرکزی به عنوان هسته‌های تاخیرگرا طراحی شده‌اند درحالی که پردازنده‌های گرافیکی به عنوان هسته‌های جریان‌گرا طراحی شده‌اند. پردازنده‌های مرکزی تمایل به داشتن حافظه موقت [۳] بزرگ‌تری نسبت به پردازنده‌های گرافیکی دارند. پردازنده‌های گرافیکی تعداد ثبات بیش‌تری برای پشتیبانی از تعداد بیش‌تری رشته [۴] نسبت به پردازنده‌های مرکزی دارند. پردازنده‌های گرافیکی واحدهای اس‌آی‌امدی [۵] بیش‌تری نسبت به پردازنده‌های مرکزی دارند. پردازنده‌های مرکزی تمایل به داشتن منطق کنترل پیچیده‌ای

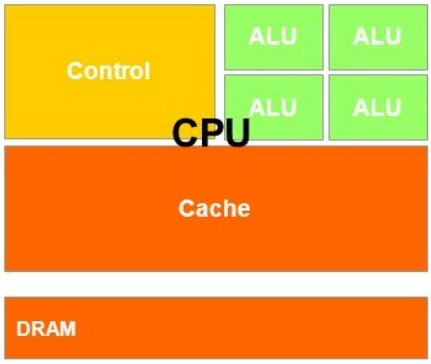
---

[۱] Latency Devices [۲] Throughput Devices [۳] Cache [۴] Thread [۵] Single Instruction Multiple Data (SIMD) Unit

[1] دارند، درحالی‌که پردازنده‌های گرافیکی منطق کنترل ساده‌ای دارند اما در عوض دارای رشته‌های بیش‌تری برای مدیریت هستند. در ادامه جزئیات بیش‌تری از معماری هریک بررسی می‌شود.

۲-۵-۱ معماری پردازنده مرکزی

در طراحی پردازنده مرکزی، سه ویژگی برجسته وجود دارد. یکی از آن‌ها واحدهای منطق ریاضی [2] است که به عنوان بخشی از منطق محاسبات عددی ریاضی را در تعداد چرخه ساعت [3] کمی انجام می‌دهند. در هسته‌های سی‌پی‌یو معمول امروزی برای یک عملیات جمع یا ضرب ۶۴ بیتی اعشاری با دقت دو برابر، بین ۱ تا ۳ چرخه ساعت انجام می‌شود. این چرخه‌ها در فرکانس‌های بسیار بالایی در حدود ۱.۵ تا بیش از ۳ گیگاهرتز صورت می‌گیرند. بنابراین در مورد تاخیر زمانی خالص عملیات ریاضی، پردازنده‌های مرکزی تاخیر بسیار ناچیزی در تولید مقادیر اعشاری دارند. ویژگی برجسته بعدی این است



شکل ۲-۷: طراحی پردازنده مرکزی

که سی‌پی‌یوها تمایل به داشتن حافظه محلی بزرگی دارند. این حافظه‌های بزرگ برای تبدیل عملیات دسترسی به حافظه اصلی با تاخیر بالا به دسترسی به حافظه محلی با تاخیر کم تعبیه می‌شوند. این در واقع تدبیری است برای نگه داشتن داده در حافظه محلی تا جای ممکن، تا این‌که هرگاه هریک از واحدهای اجرایی سی‌پی‌یو نیاز به دسترسی به داده داشته‌باشند، احتمال وجود داده در حافظه محلی سی‌پی‌یو از یک دسترسی قبلی بسیار بالا باشد. در نتیجه زمان دسترسی به داده به میزان قابل توجهی کاهش می‌یابد.

سومین ویژگی منحصربفرد سی‌پی‌یو کنترل پیچیده است. منطق کنترل معمولا خود را به دو صورت آشکار می‌سازد. یکی پشتیبانی پیش‌بینی شاخه [4] برای کاهش تاخیر شاخه و دیگری انتقال داده [5] برای کاهش تاخیر انتقال داده.

تعریف ۲-۵-۱ (دستورات شاخه). این دستورات در سی‌پی‌یو برای ساختارهای زبان سطح بالا مانند عبارات if و else همچون حلقه‌ها تولید می‌شوند. تصمیم‌گیری در انتخاب یکی از مسیرها در چنین ساختارهایی یا تصمیم‌گیری برای این‌که آیا یک حلقه باید دوباره تکرار شود یا خروج انجام شود، با دستورات کنترل شاخه صورت می‌گیرد.

---

[1] Sophisticated Control Logic [2] Arithmetic Logic Unit (ALU) [3] Clock Cycles [4] Branch Prediction Support [5] Data Forwarding

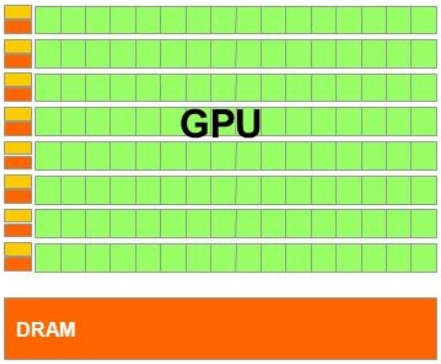
تصمیم‌گیری‌های کنترلی ممکن است زمان‌بر باشد بنابراین وجود یک قابلیت پیش‌بینی در اکثر ریزپردازنده‌های امروزی برای پیش‌بینی مسیر دستورات شاخه، امکان دریافت و اجرای سریع همه این دستورات که در مسیر پیش‌بینی شده قرار دارند را بوجود می‌آورد. اگر چه باید امکان وجود حالت‌هایی که پیش‌بینی می‌شود شکست بخورد، فراهم شود. اگر پیش‌بینی اشتباه باشد، باید حالت‌های محاسباتی کافی برای جبران پیش‌بینی غلط وجود داشته‌باشد و بنابراین باید همه این منابع در منطق کنترل وجود داشته‌باشد تا بتوان پیش‌بینی شاخه را پشتیبانی کرد و پیش‌بینی به همراه بازیابی از پیش‌بینی اشتباه را انجام داد.

همچنین بخش زیادی از منطق پیچیده به انتقال داده اختصاص داده می‌شود. این بخش مربوط به شرایطی است که نتایج خروجی یک دستور توسط دستورات متعاقب مورد نیاز باشد. منطق انتقال داده محل چنین دستوراتی در خط لوله [۱] را مشخص می‌کند و مسیریابی نتایج مربوطه را به آن دستورات در سریع‌ترین زمان ممکن انجام می‌دهد. این عملیات شامل تعداد زیادی مدارات مقایسه و مسیریابی خواهد بود که نیازمند منابع اجرایی زیادی می‌باشد.

در نتیجه همه این سازوکارها در راستای کاهش تاخیر به کار گرفته می‌شوند. اولی برای کاهش تاخیر محاسبات ریاضی، دومی برای کاهش تاخیر دسترسی به حافظه و سومی برای کاهش تصمیم‌گیری شاخه به همراه کاهش تاخیر تولید مقادیر داده و اجرای همه دستوراتی که این مقادیر را به عنوان ورودی دریافت می‌کنند.

۲-۵-۲ معماری پردازنده گرافیکی

جی‌پی‌یوها به عنوان ابزارهای عملیاتی (اجرایی) طراحی شده‌اند. آن‌ها حافظه‌های محلی بسیار کوچکی دارند. این حافظه‌ها جهت نگهداری داده درون خود برای دسترسی‌های آینده طراحی نشده‌اند. بلکه این حافظه‌های محلی به عنوان واحدهای طبقاتی برای تعداد زیادی رشته طراحی شده‌اند. وقتی تعداد زیادی رشته که به صورت هم‌زمان درحال اجرا هستند، نیاز به داده‌های یکسان داشته‌باشند، حافظه محلی این درخواست‌ها را به یک درخواست ادغام می‌کند تا آن درخواست به حافظه تصادفی پویا [۲] رجوع نموده و وقتی داده برگردانده می‌شود، کنترل کننده حافظه محلی به عنوان یک منطق حمل و نقل در جهت توزیع داده به همه واحدهای اجرایی یا همان رشته‌ها که نیاز به آن داده دارند، اقدام نماید. بنابراین تاخیر برای رجوع



شکل ۲-۸: طراحی پردازنده گرافیکی

به حافظه تصادفی پویا وجود خواهد داشت. اگر چه این منطق اجازه می‌دهد که چندین دسترسی در تعداد دسترسی کم‌تری

[۱] Pipeline [۲] Dynamic Random-Access Memory (DRAM)

ادغام شود تا امکان اعمال محدودیت ترافیک ورودی به حافظه پویا وجود داشته‌باشد.

دومین ویژگی مهم جی‌پی‌یوها داشتن واحد کنترل ساده است که در تصویر به صورت مربع‌های کوچک نارنجی رنگ نشان داده‌شده. در کنترل جی‌پی‌یو معمولا پیش‌بینی شاخه وجود ندارد و کنترل انتقال داده نیز به مقدار بسیار اندک وجود دارد. ویژگی سوم، واحدهای منطق ریاضی با انرژی کارآمد [۱] است. به جای ساخت تعداد کمی واحد منطق ریاضی قدرتمند با تاخیر بسیار کم، جی‌پی‌یوها با تعداد زیادی واحد منطق ریاضی با تاخیر بالا اما با مصرف انرژی کارآمد ساخته می‌شوند. این ای‌ال‌یوها معمولا به شدت دارای سیستم خط لوله‌ای هستند تا بتوانند در هر چرخه ساعت یک عمل را دریافت کنند و بعد زمان زیادی برای تولید جواب هر عمل صرف می‌شود. اما در خروجی ALU در هر چرخه ساعت یک جواب از خط لوله خارج می‌شود. بنابراین با داشتن تعداد زیادی ALU و چنین خط لوله طولانی برای هر یک از آن‌ها، نیاز به داشتن تعدادی زیادی رشته است تا در هر مرحله درون این ای‌ال‌یوها یک عملیات ریاضی را انجام دهند. در نتیجه امکان بهره‌وری حداکثری از سخت افزار فراهم می‌گردد. به همین دلیل این دستگاه‌ها به گونه‌ای طراحی شده‌اند که اگر تعداد رشته‌های بسیار زیادی داشته باشیم، هر یک به صورت همزمان بخشی از عملیات را انجام می‌دهند و به این صورت امکان بهره‌وری این تعداد زیاد ALU برای تولید جواب در یک توانایی بسیار بالا بوجود می‌آید. با وجود این‌که هر رشته زمان زیادی برای اجرای هر عملیات نسبت به همتایان خود در سی‌پی‌یو صرف می‌کنند.

اکنون پس از بررسی فلسفه طراحی هسته‌های سی‌پی‌یو و جی‌پی‌یو، لزوم استفاده برنامه‌های امروزی از هر دوی این ابزارها قابل درک است. از سی‌پی‌یو برای اجرای قسمت‌های ترتیبی برنامه جایی که تاخیر به واقع دارای اهمیت است، استفاده می‌شود. چرا که در بخش ترتیبی یک برنامه عملیات بسیار اندکی برای اجرای موازی وجود دارد. بنابراین لازم است این عملیات اندک به سرعت انجام شود تا بتوان در کوتاه‌ترین زمان ممکن از آن‌ها عبور کرد. از طرف دیگر از جی‌پی‌یوها برای پردازش موازی استفاده می‌شود جایی که بازدهی برنده است. در قسمت‌های موازی برنامه تعداد بسیار زیادی عملیات دارای قابلیت اجرای موازی وجود دارد. لذا از این عملیات برای به‌کارگیری تعداد بی‌شمار ای‌ال‌یوها در جی‌پی‌یو استفاده می‌شود تا بتوان به سرعت از بخش‌های موازی عبور کرد. هرگاه از جی‌پی‌یو برای اجرای این بخش‌های موازی استفاده می‌شود، می‌توان به عملکرد ۱۰ برابری یا بیش‌تر نسبت به سی‌پی‌یوها دست یافت. از طرف دیگر واضح است که سی‌پی‌یوها نیز می‌توانند عملکردی چندین برابر جی‌پی‌یوها در عملیات ترتیبی داشته‌باشند.

معمولا به محاسباتی که هم از جی‌پی‌یو و هم سی‌پی‌یو برای اجرا استفاده می‌کنند، محاسبات موازی ناهمگون و به اختصار محاسبات جی‌پی‌یو [۲] گفته می‌شود. این عرصه به سرعت در حال رشد است. مقالات متعددی در این عرصه در سال‌های اخیر در مباحث کاربردی مختلف به همراه برخی ابزارها و بسترها منتشر شده‌است. این امکانات به یاری توسعه دهندگان برنامه‌ها در جهت استفاده از قابلیت‌های محاسبات موازی ناهمگون آمده است. عرصه‌هایی که با موفقیت محاسبات موازی ناهمگون را به کار بسته‌اند رو به افزایش‌اند. برخی از این عرصه‌ها شامل تحلیل مالی، شبیه‌سازی علمی، شبیه‌سازی مهندسی، تجزیه و تحلیل اطلاعات فشرده، تصویربرداری پزشکی، پردازش صوت دیجیتال، پردازش ویدیو دیجیتال، بینایی ماشینی، اطلاع رسانی زیست پزشکی، اتوماسیون طراحی الکترونیکی، مدل‌سازی آماری، روش‌های عددی،

[۱] Energy Efficient ALUs [۲] GPU Computing

ردیابی اشعه ری و فیزیک تعاملی می‌باشد.

۲-۶ برنامه‌نویسی جی‌پی‌یو

یک واحد پردازنده گرافیکی (جی‌پی‌یو) امکان اجرای گرافیک با کیفیت بالا را بر روی رایانه فراهم می‌سازد. همانند سی‌پی‌یو، جی‌پی‌یو نیز یک پردازنده چندهسته‌ای تک تراشه است. اگرچه، صدها هسته جی‌پی‌یو به نحوی هماهنگ شده است که یک واحد سخت‌افزاری یکتا را تشکیل می‌دهد. یک جی‌پی‌یو می‌تواند هزاران رشته سخت‌افزاری هم زمان داشته باشد. جی‌پی‌یو برای داده‌های موازی و بخش‌های فشرده محاسباتی یک الگوریتم به‌کار گرفته می‌شود. الگوریتم‌های داده-موازی برای چنین دستگاه‌هایی بسیار مناسب هستند چرا که سخت‌افزار می‌تواند از نوع دستور یکتا چندین رشته (اس‌آی‌ام‌تی) طبقه‌بندی شود. جی‌پی‌یوها از لحاظ جی‌فلاپس (گیگا عملیات اعشاری بر ثانیه) [۱] از سی‌پی‌یو عملکرد بهتری دارند. برای مثال، توان محاسباتی یک پردازنده هسته i۷ با ۳.۴۶ گیگاهرتز، به حداکثر ۵۵.۳۶ جی‌فلاپس می‌رسد، در حالی که توان محاسباتی حداکثری یک کارت گرافیک ان‌ویدیا تسلا K۴۰ [۲] به ۵ ترافلاپ می‌رسد.

جی‌پی‌یوها عملکرد محاسباتی و پهنای باند حافظه مناسبی ارائه می‌دهند. همین موضوع باعث شده جی‌پی‌یو یک معماری جذاب برای اجرای الگوریتم‌های محاسباتی فشرده تلقی گردد. در حال حاضر دو مدل کدنویسی عمده برای توسعه برنامه روی جی‌پی‌یو در دسترس است، کودا (معماری دستگاه محاسبه یکپارچه) [۳] و اوپن‌سی‌ال (زبان محاسبات باز) [۴]. شرکت ان‌ویدیا کودا را در اواخر سال ۲۰۰۶ معرفی کرد و این معماری تنها به همراه جی‌پی‌یوهای این شرکت قابل استفاده است. در حالی که شرکت اپل اوپن‌سی‌ال را در سال ۲۰۰۸ معرفی کرد و برای جی‌پی‌یوهای مختلف و حتی سی‌پی‌یوها در دسترس است. زبان‌های برنامه‌نویسی عمومی مانند کودا و اوپن‌سی‌ال، بر پایه مدل پردازش جریان [۵] بنا شده‌اند. این مدل به دلیل ماهیت سطح پایین آن، برنامه نویسی جی‌پی‌یو را سخت می کند.

کار اصلی جی‌پی‌یو محاسبه توابع ۳ بعدی است چرا که این نوع محاسبات در صورت اجرا روی سی‌پی‌یو بسیار زمان‌بر هستند. اگرچه امروزه جی‌پی‌یوها با انجام محاسبات عمومی تکامل یافته‌اند و امکان رسیدگی به سایر وظایف را دارند. اگر وظایف پیچیده یک برنامه را روی جی‌پی‌یو اجرا کنیم، امکان آزادسازی سی‌پی‌یو بوجود می‌آید و لذا می‌توان از سی‌پی‌یو برای سایر وظایف استفاده کرد. برای مثال سی‌پی‌یوها برای اجرای وظایف مختلف سیستم‌عامل‌ها همچون برنامه‌ریزی کارها [۶] و مدیریت حافظه [۷] بسیار مناسب هستند. همچنین سی‌پی‌یوها در مورد برنامه‌های وظیفه-موازی [۸] بسیار کارآمد هستند درحالی که جی‌پی‌یوها در برنامه‌های داده-موازی [۹] کارایی بیشتری نشان می‌دهند. در مجموع، طراحی سی‌پی‌یوها برای اجرای ترتیبی کد بهینه‌سازی شده در حالی‌که جی‌پی‌یوها برای رسیدگی به محاسبات عددی فشرده طراحی شده‌اند. بنابراین، بیش‌تر برنامه‌ها باید از هردوی سی‌پی‌یو و جی‌پی‌یو استفاده نمایند. بخش ترتیبی برنامه توسط سی‌پی‌یو اجرا شود و

[۱] Giga Floating point Multiple Threads (GFLOPS) [۲] NVIDIA Tesla K40 [۳] CUDA (Compute Unified Device Architecture) [۴] OpenCL (Open Computing Language) [۵] Stream Processing Model [۶] Job Scheduling [۷] Memory Management [۸] Task-parallel Programs [۹] Data-parallel Programs

بخش‌های زمان‌بر می‌توانند بوسیله جی‌پی‌یو تسریع گردند.

۲-۶-۱ انواع حافظه در جی‌پی‌یو

هر جی‌پی‌یو کودا دارای انواع حافظه زیر است:

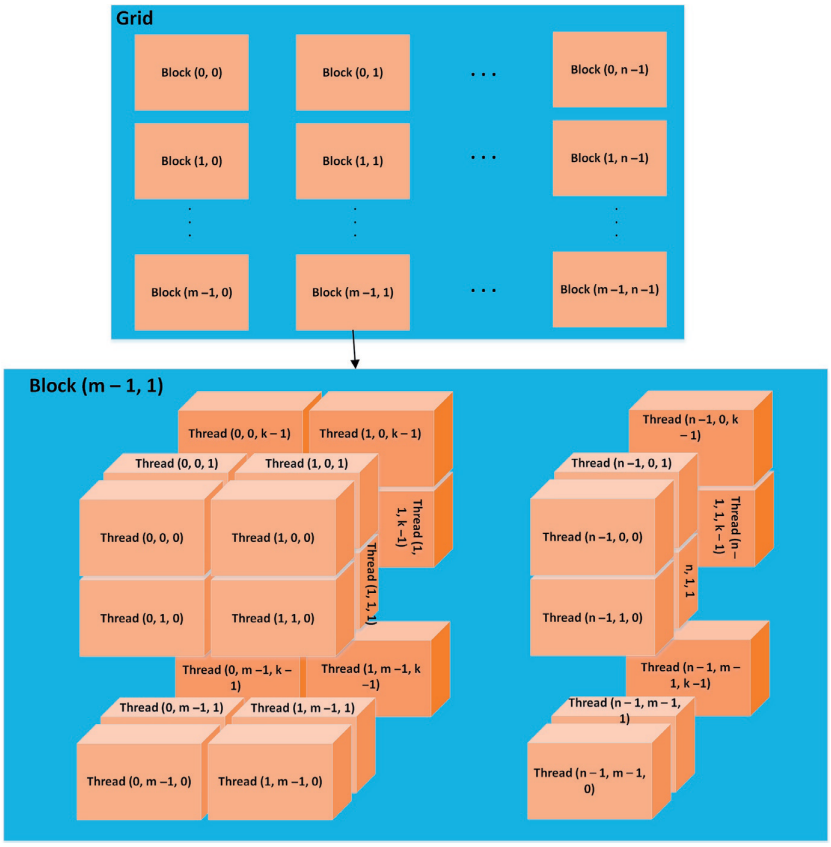
- **حافظه ثبّات** [۱] : سریع‌ترین حافظه است. در اکثر مواقع، دسترسی به ثبّات در هر دستورالعمل صفر چرخه ساعت زمان مصرف می‌کند. یک عدد حافظه ثبّات برای هر رشته جهت ذخیره و بازیابی سریع داده‌های کوچک، مانندشمارنده‌ها، تخصیص داده شده‌است. این داده‌ها در ثبّات تنها در مدت عمر آن رشته باقی می‌مانند. داده‌های ذخیره شده در حافظه ثبّات تنها برای رشته مربوط به خودش قابل دسترس است.
- **حافظه محلی** [۲]: این نوع حافظه عموما برای داده‌هایی که نمی‌توانند در داخل رجیسترها جای گیرند استفاده می‌شوند. این حافظه‌ها نوع فیزیکی حافظه نیستند بلکه انتزاعی از حافظه سراسری هستند. از این رو، حافظه محلی کند و کش نشده است.
- **حافظه اشتراکی** [۳]: این نوع از حافظه برای تمامی رشته‌های موجود در یک بلاک قابل دسترسی است و تنها در مدت عمر آن بلاک باقی می‌ماند. حافظه اشتراکی به رشته‌های یک بلاک اجازه می‌دهد تا در میان خودشان داده به اشتراک بگذارند و در ارتباط باشند. عملکرد حافظه اشتراکی قابل مقایسه با حافظه ثبّات می‌باشد. اگرچه، دسترسی متوالی به داده توسط تعداد زیادی رشته ممکن است منجر به تگناهایی بشود.
- **حافظه ثابت** [۴]: این نوع حافظه متغیرهای کرنل ثابت را ذخیره می‌کند که در طول اجرای یک کرنل تغییر نمی‌کنند. سرعت کندی دارند و تنها امکان خواندن از آن‌ها وجود دارد.
- **حافظه بافتی** [۵]: این نوع هم یک حافظه کش شده روی تراشه است. کش‌های بافتی مخصوص کاربردهای گرافیکی هستند که الگوهای دسترسی حافظه در آن‌ها به مقدار زیادی موقعیت فضایی بروز می‌دهد.
- **حافظه سراسری** [۶]: این نوع از حافظه برای همه رشته‌های درون برنامه و همچنین میزبان، در دسترس است و تنها در مدت عمر تخصیص میزبان باقی می‌ماند. حافظه کند و کش نشده است.

۲-۶-۲ ساختار کودا

هر برنامه‌ای که روی جی‌پی‌یو اجرا می‌شود، به تعداد زیادی رشته تقسیم می‌شود. هر رشته به طور مستقل همان برنامه را روی داده‌های مختلف اجرا می‌کند. یک رشته بلاک گروهی از رشته‌هاست که از طریق حافظه اشتراکی همکاری می‌کنند و

[۱] Register Memory [۲] Local Memory [۳] Shared Memory [۴] Constant Memory [۵] Texture Memory [۶] Global Memory

به منظور هماهنگ کردن دسترسی به حافظه اجرا شدن خود را همگام سازی می‌نمایند. یک توری [1] شامل گروهی از رشته بلاک‌ها است و یک کرنل روی تعدادی توری اجرا می‌شود. یک کرنل برنامه حاصله پس از کامپایل است. ساختار سلسله مراتبی معماری کودا در شکل ۲-۹ شامل واحدهای پایه ذیل است:



شکل ۲-۹: ساختار کودا

تعریف ۲-۶-۱ (توری). یک توری گروهی از رشته‌هاست که یک کرنل یکتا را اجرا می‌کند. یک توری به عنوان یک آرایه دوبعدی از بلاک‌ها سازماندهی شده‌است. هر فراخوانی از سی‌پی‌یو از طریق یک توری انجام می‌شود. رشته‌های موجود در یک توری در یک سلسله مراتب دو سطحی با استفاده از مختصات منحصر بفرد به نام‌های $blockId$ و $threadId$ سازماندهی شده‌اند. همه رشته‌های درون یک توری مقدار شماره بلاک $blockId$ یکسانی دارند. بلاک‌های یک توری دارای دو جزء هستند. یکی مختصات x که به صورت $blockId.x$ نشان داده می‌شود و دیگری مختصات y که به صورت $blockId.y$ نشان داده می‌شود.

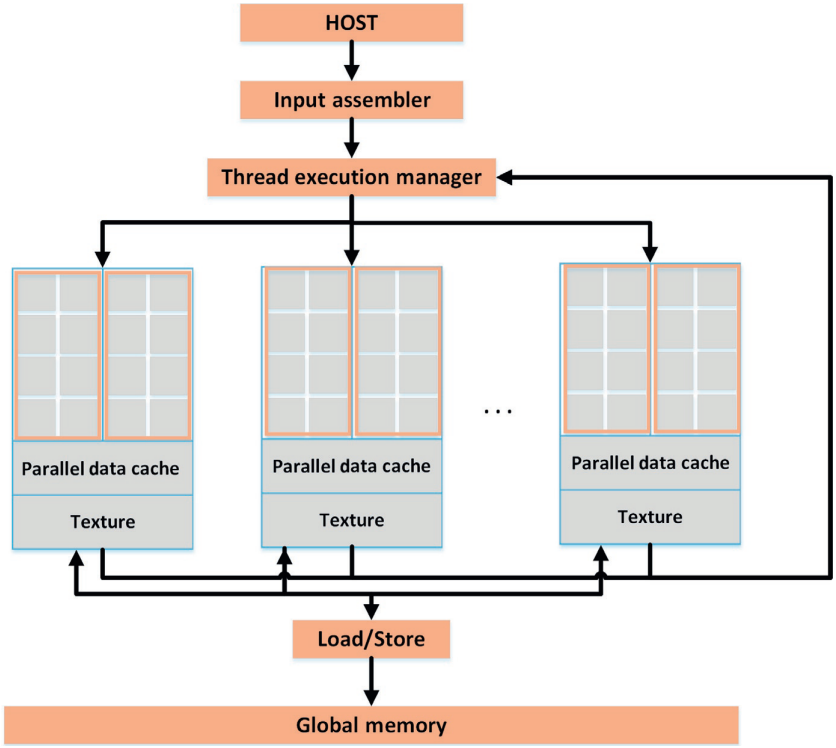
تعریف ۲-۶-۲ (بلاک). یک بلاک یک واحد منطقی شامل تعدادی رشته هماهنگ و مقداری مشخصی حافظه اشتراکی می‌باشد. یک بلاک به عنوان یک آرایه سه‌بعدی از رشته‌ها سازماندهی می‌گردد. رشته‌های درون یک بلاک دارای سه جزء هستند. اولی مختصات x، $threadId.x$ دومی مختصات y، $threadId.y$ و آخری مختصات z، $threadId.z$.

[1] Grid

تعریف ۲-۶-۳ (رشته). یک رشته بخشی از یک بلاک است. رشته‌ها روی هسته‌های مستقل زیرپردازنده‌ها اجرا می‌شوند اما فقط به یک هسته محدود نیستند. هر رشته دارای مقدار مشخصی حافظه ثبّات است.

۲-۷ معماری کودا

ان‌ویدیا کودا معماری است که محاسبات داده‌موازی را روی جی‌پی‌یو مقدور می‌سازد. کودا سی/سی++ یک توسعه از زبان برنامه‌سازی سی و سی++ برای محاسبات همه‌منظوره است. یک برنامه کودا شامل دو بخش است، یک بخش روی سی‌پی‌یو اجرا می‌شود و بخش دیگر روی جی‌پی‌یو. سی‌پی‌یو باید به عنوان دستگاه میزبان مشاهده شود، در حالی‌که جی‌پی‌یو به عنوان پردازنده همکار یا هم‌پردازنده [۱] تلقی می‌گردد. بخشی از برنامه که قابلیت موازی‌سازی دارد، به عنوان کِرنل [۲] روی جی‌پی‌یو اجرا می‌شود. یک کِرنل محاسبات نوع برنامه یکتا چندین داده (اس‌پی‌امدی) است که با استفاده از تعداد زیادی رشته موازی اجرا می‌شود. سی‌پی‌یو اجرای هر قسمت برنامه را شروع می‌کند و یک تابع کِرنل را فراخوانی می‌کند، سپس اجرا به جی‌پی‌یو منتقل می‌شود. ارتباط میان حافظه سی‌پی‌یو و حافظه جی‌پی‌یو بوسیله یک ارتباط سریع نقطه به نقطه PCI $16x$ [۳] صورت می‌گیرد. تصویر ۲-۱۰ معماری یک جی‌پی‌یو دارای قابلیت کودا را نشان می‌دهد. جی‌پی‌یو به آرایه‌ای از چندپردازنده‌های



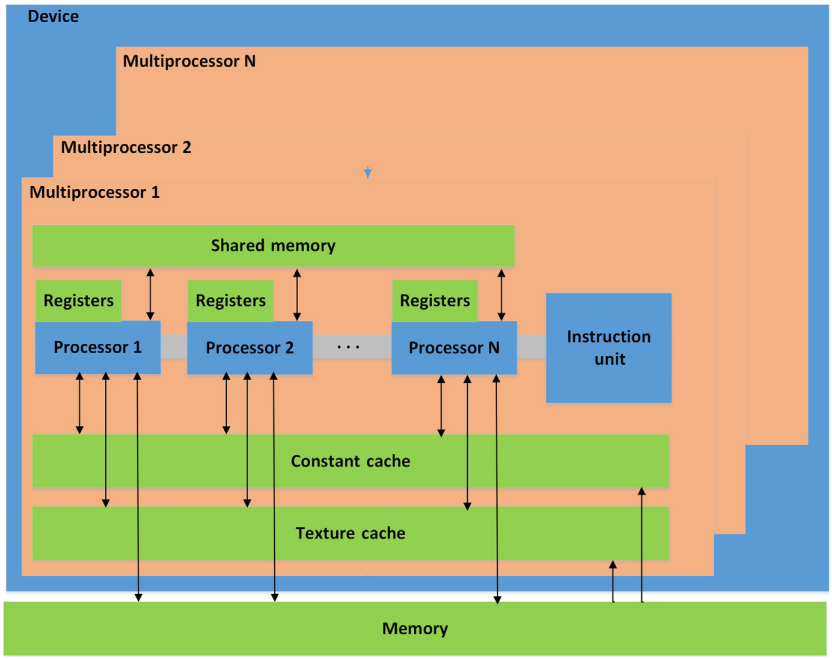
شکل ۲-۱۰: معماری جی‌پی‌یو با قابلیت کودا

جریانی (اس‌ام) [۴] به شدت رشته‌ای سازماندهی شده‌است. تعدای اس‌ام یک بلوک ساختاری [۵] را تشکیل می‌دهند و هر اس‌ام

[۱] Co-processor [۲] Kernels [۳] PCIe 16x [۴] Streaming Multiprocessor (SM) [۵] Building Block

دارای تعدادی پردازنده جریانی (اس‌پی) است که منطق کنترل و حافظه کش دستورات را به طور مشترک در اختیار دارند. همچنین جی‌پی‌یوها یک حافظه گرافیکی نرخ داده دوتایی [۱] دارند. جی‌پی‌یوهای امروزی حافظه چند گیگابایتی دارند. این حافظه خارج تراشه [۲] دارای پهنای باند بالایی است و در مقایسه با حافظه سیستمی تاخیر بیش‌تری دارد.

کارت گرافیک Geforce GTX 1050 که در این پایان‌نامه از آن استفاده شده‌است، شامل ۴ گیگابایت حافظه و ۱۱۲ گیگابایت بر ثانیه پهنای باند حافظه است. هر اس‌ام دارای مقدار کمی حافظه اشتراکی در تراشه [۳] و سخت‌افزار تخصصی برای ذخیره داده های فرمت شده به عنوان بافت [۴] و یا به عنوان ثابت [۵] است (شکل ۲-۱۱).



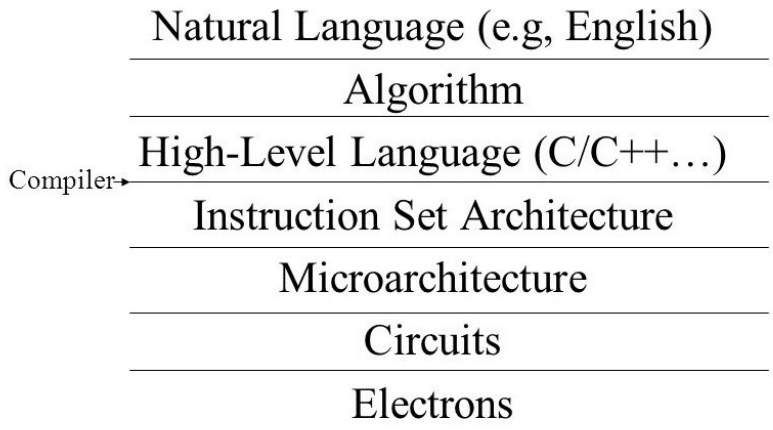
شکل ۲-۱۱: معماری حافظه جی‌پی‌یو کودا

۲-۷-۱ معماری مجموعه دستورالعمل

آی‌اس‌ای یا معماری مجموعه دستورالعمل [۶] یک قرداد میان سخت‌افزار و نرم‌افزار و یک مجموعه دستورالعمل برای سخت‌افزار است که اجرا کند. هر گاه برنامه‌ای به زبان کودا نوشته می‌شود، در نهایت به سطح پایین‌تر معماری مجموعه دستورالعمل کامپایل می‌شود. در این سطح یک برنامه مجموعه‌ای از دستورات است که در حافظه ذخیره شده و امکان خوانده، تفسیر و اجرا شدن توسط سخت‌افزار را دارد. دستورات برنامه روی داده‌های موجود در حافظه و یا داده‌های فراهم شده توسط ابزارهای ورودی/خروجی عملیات انجام می‌دهد.

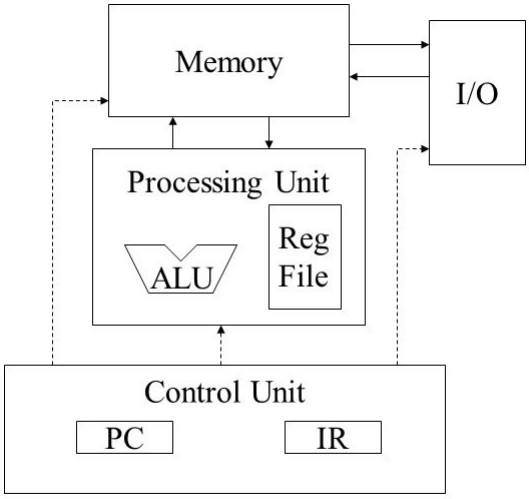
---

[۱] Graphics Double Data Rate (GDDR) [۲] Off-chip Memory [۳] On-chip [۴] Texture Cache
[۵] Constant Cache [۶] Instruction Set Architecture (ISA)



©Yale Patt and Sanjay Patel, *From bits and bytes to gates and beyond*

شکل ۲-۱۲: معماری مجموعه دستورالعمل



شکل ۲-۱۳: پردازنده وان-نومن

۲-۷-۲ پردازنده وان-نومن

در تصویر (۲-۱۳) نحوه سازماندهی سخت‌افزار برای اجرای برنامه‌های در سطح معماری مجموعه دستورالعمل نشان داده شده‌است. این نمودار در واقع بر اساس مدل پردازنده وان-نومن رسم شده‌است که توسط جان وان-نومن در دهه ۱۹۴۰ ارائه شده. امروزه به صورت مجازی تمامی هسته‌های پردازنده بر پایه گونه‌های مختلف این مدل طراحی شده‌اند. در قسمت پایین واحد کنترل نمایش داده شده که شامل شمارنده [۱] برنامه و ثبات دستورات [۲] است. شمارنده برنامه الزاما موقعیت در حافظه را مشخص می‌کند. جایی که برنامه می‌تواند دستور بعدی که باید اجرا شود را بیابد. یک ارتباط از واحد کنترل به حافظه دیده می‌شود که مربوط به رساندن مقدار شمارنده به حافظه و درخواست برگرداندن بیت‌های دستور است. زمانی

[۱] Program Counter(PC) [۲] Instruction Register (IR)

که بیت‌های دستور از حافظه برمی‌گردند، در ثبّات دستورات (آی‌آر) قرار می‌گیرند. در اینجاست که سخت افزار بیت‌های دستور را بررسی و تمامی فعالیت‌هایی که باید برای اجرای دستور رخ دهد را تعیین می‌کند. این فعالیت‌ها بوسیله سیگنال‌های کنترل که با فِلش از واحد کنترل به واحد پردازش نشان داده‌شده، هماهنگ‌سازی می‌گردند. این سیگنال‌های کنترل لزوما فعالیت‌هایی که ALU، فایل ثبّات و سایر اجزاء واحد پردازش باید در هر چرخه ساعت برای اجرای دستور دریافت کنند را تعریف می‌نمایند. در طول اجرا، برخی دستورات نیاز به دسترسی به داده از حافظه خواهند داشت که با خطوط نشان‌گر از واحد پردازش به حافظه نشان داده شده‌است. در نهایت برخی از داده‌ها به حافظه یا به ورودی/خروجی بازگردانده می‌شوند.

اکنون می‌توان درباره جزئیات بیش‌تری از یک رشته کودا بحث نمود. یک رشته کودا در واقع یک پردازنده وان-نومن انتزاعی یا مجازی شده است. می‌توان هر رشته کودا را یکی از این پردازنده‌ها در نظر گرفت. هر یک از این پردازنده‌ها می‌توانند یک برنامه را اجرا کنند. تابع کِرنل که درمورد آن توضیح داده شد، همان برنامه است. سخت‌افزار تعداد زیادی از این پردازنده‌های وان-نومن را تولید می‌کند. هر یک از پردازنده‌ها تابع کرنل را اجرا خواهند کرد. مجازی‌سازی به گونه‌ای شده که اگر به سخت‌افزار بنگریم، تعداد پردازنده‌های واقعی بسیار کم‌تر از رشته‌هایی است که یک برنامه کودا تولید خواهد کرد. بنابراین بسیاری از این رشته‌ها باید روی پردازنده‌های واقعی به نوبت اجرا شوند.

۲-۸ ویژگی‌های بنیادین کودا سی

پدیده موازی‌سازی داده به این صورت است که هر قسمت از داده را می‌توان مستقل از دیگری پردازش نمود. به عنوان مثال در جمع دو بردار، هر درایه متناظر به طور مستقل با هم جمع می‌شوند. بنابراین اگر بردارهای بزرگ و سخت‌افزار به اندازه کافی بزرگ داشته‌بشیم، امکان انجام تمامی این عملیات جمع به صورت موازی وجود دارد. اساسا به همین صورت می‌توان به عملکرد بالا در یک برنامه کودا دست یافت. به همین دلیل از این مثال ساده برای نشان دادن مفهوم پایه کودا استفاده می‌کنیم.

مدل اجرایی موازی کودا و فامیل نزدیکش اوپن‌سی‌ال، هر دو بر پایه یک نوع آرایش میزبان به همراه دستگاه (میزبان + دستگاه) عمل می‌کنند. بنابراین مفهوم پایه این است که وقتی شروع به اجرای یک برنامه می‌کنیم، این برنامه روی میزبان (هسته سی‌پی‌یو) اجرا می‌شود و وقتی به قسمت موازی برنامه می‌رسیم، فرصت استفاده از یک دستگاه بازده‌گرا (عملیاتی، در اینجا همان جی‌پی‌یو) بوجود می‌آید. برای این کار از توابع مخصوصی به نام کِرنل استفاده می‌کنیم که در بخش قبل هم معرفی شدند. این توابع کرنل شباهت زیادی به توابع زبان برنامه‌نویسی سی دارند. همانند این توابع پارامترهای ورودی دارند. اگرچه علاوه بر پارامترهای تابع، پارامترهای پیکربندی هم باید برای توابع کرنل تنظیم شود.

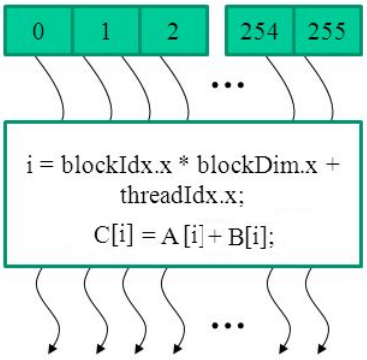
برنامه ۲-۱: جمع دو بردار به زبان سی

```
KernelA<<<nBlk, nTid>>>(args);
```

در فراخوانی (۲-۲) ، $nBlk$ نشان دهنده تعداد رشته بلاک‌ها و متغیر بعدی $nTid$ تعداد رشته‌های هر بلاک است. عبارت "دستگاه" [۱] در کودا به معنای دستگاه اجراکننده موازی است که در اکثر مواقع مقصود جی‌پی‌یوهای عمل‌گرا می‌باشد.

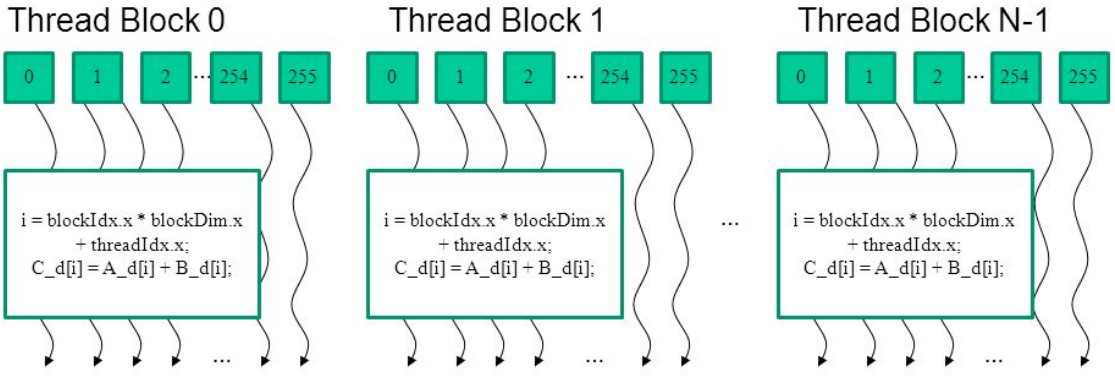
[۱] Device

۲-۸-۱ آرایه‌هایی از رشته‌های موازی



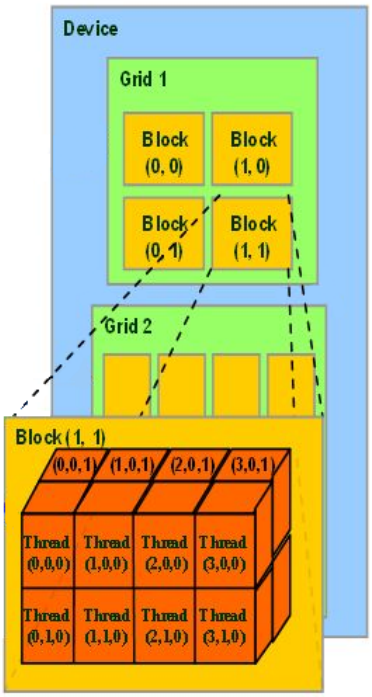
شکل ۲-۱۴: توری تک بعدی با یک بلاک یک بعدی

یک کرنل کودا توسط توری اجرا می‌شود که آرایه‌ای از رشته‌هاست. در تصویر یک رشته بلاک تک بعدی نشان داده شده‌است. فرض بر این است که توری تنها یک رشته بلاک دارد. همه رشته‌ها یک کد یکسان را اجرا می‌کنند. هر رشته یک مقدار اندیس متفاوتی دارد که برای محاسبه آدرس‌های حافظه و تصمیم‌گیری‌های کنترلی استفاده می‌شوند. در این مثال ۲۵۶ رشته در بلاک وجود دارد که هریک از آن‌ها اندیس رشته منحصربفردی از ۰ تا ۲۵۵ دارند. بخشی از کد در کرنل داخل مستطیل نوشته شده که این کد iامین متغیر را بر اساس اندیس رشته محاسبه می‌کند. متغیر i برای هر رشته خصوصی است. هر پردازنده وان-نومن که متناظر با رشته مربوطه است، یک متغیر i منحصربفرد خواهد داشت. رشته صفرم i مربوط به خودش را خواهد داشت، رشته یکم نیز همینطور و به همین ترتیب تا آخر هر رشته اندیس‌دهی می‌شود. اندیس رشته به صورت $threadIdx.x$ نشان داده می‌شود. در نتیجه زمانی که رابطه $C[i] = A[i] + B[i]$ را اجرا می‌کنیم، مقدار i برای هر رشته متفاوت خواهد بود. اکنون یک توری با چندین رشته بلاک را شرح می‌دهیم. تصویر (۲-۱۵) یک توری از رشته‌ها



شکل ۲-۱۵: توری تک بعدی با N بلاک یک بعدی

که به N رشته بلاک سازماندهی شده‌است را نشان می‌دهد. همچنان هر رشته بلاک شامل ۲۵۶ رشته است. در این‌جا هر



شکل ۲-۱۶: توری ۲ بعدی با بلاک‌های سه بعدی

رشته علاوه بر اندیس رشته دارای اندیس بلاک نیز هست که به صورت $blockIdx.x$ نشان داده می‌شود. $blockIdx.x$ و $threadIdx.x$ متغیرهای از پیش تعریف شده کودا هستند که می‌توان در نوشتن یک کرنل از آن‌ها استفاده کرد. این متغیرها برای هر رشته توسط سخت‌افزار مقداردهی اولیه می‌شوند. بنابراین نیازی به مقداردهی اولیه آن‌ها وجود ندارد. در چنین چینشی مقدار اندیس i برای اولین رشته در دومین بلاک با $threadIdx.x = ۰$ و $blockIdx.x = ۱$، ۲۵۶ خواهد بود. در نتیجه، مقادیر i برای اولین رشته بلاک از ۰ تا ۲۵۵ خواهد بود و همه مقادیر بلاک دوم یا بلاک شماره ۱، بین ۲۵۶ و ۵۱۲ هستند و به همین ترتیب همه رشته‌ها یک پوشش یکنواخت از اجزاء آرایه تشکیل می‌دهند. این نوع پوشش در واقع یک پوشش خطی از یک آرایه تک بعدی است. با استفاده از این فرمول می‌توان مطمئن بود که هر جزء A، B و C توسط یکی از رشته‌ها پوشش داده شده‌است. در واقعیت ممکن است هر رشته بیش از یک جزء را پوشش دهد. رشته‌های درون یک رشته بلاک می‌توانند بوسیله حافظه اشتراکی، عملیات اتمی و هماهنگ سازی مانع [۱] حافظه اشتراکی به رشته‌ها امکان تبادل داده می‌دهد، عملیات اتمی به رشته‌ها اجازه می‌دهد بروزرسانی‌هایشان را به متغیرهای یکسان هماهنگ کنند و هماهنگ‌سازی مانع باعث می‌شود رشته‌ها بتوانند سایر رشته‌ها را وادار به انتظار نمایند. همه این فعالیت‌ها امکان هماهنگی میان رشته‌های یک بلاک بوجود می‌آورند. اگر چه رشته‌های یک بلاک، امکان تعامل با رشته‌های بلاک دیگر ندارند.

اندیس رشته و اندیس بلاک فقط تک بعدی نیستند. در کودا، هر اندیس بلاک و هر اندیس رشته می‌تواند یک بعدی، دو بعدی یا سه بعدی باشد. به همین دلیل مثلا در انتهای نام متغیر اندیس بلاک، یک نقطه و بعد یکی از حروف x، y یا z به صورت $blockIdx.x$ قرار دارد. بسیاری از برنامه‌های کاربردی، روی داده‌های دوبعدی مانند پردازش تصویر، داده‌های سه‌بعدی مانند حل معادلات دیفرانسیل با حجم جزئی کار می‌کنند. بنابراین سیستم سه‌بعدی اندیس‌دهی بلاک و رشته در

[۱] Barrier Syncronization

کودا کار با داده‌های چندبعدی و خواندن برنامه‌ها را راحت‌تر می‌کند. در شکل (۲-۱۶) یک ساختار دوبعدی بلاک نشان داده شده که درون هر بلاک یک ساختار سه‌بعدی رشته‌ها وجود دارد. بنابراین در درون یک توری، هر بلاک دارای دو اندیس x و y است. هر بعد در کودا می‌تواند از ۲ تا ۱۶ رشد کند. در شکل(۲-۱۶) بلاک $(۱, ۱)$ بسط داده شده‌است. در این بلاک ۱۶ رشته وجود دارد که بعد x از اندیس ۰ تا ۳ و بعد y و z، از اندیس ۰ تا ۱ متغیر هستند. بنابراین می‌توان یک ساختار توری دو بعدی با بلاک‌های سه‌بعدی یا بر اساس نیازهای برنامه و کاربرد آن ابعاد متفاوتی در نظر گرفت.

۲-۹ برنامه جمع دو بردار به زبان کودا

در ادامه در قالب مثال ساده جمع دو بردار، نحوه کد نویسی به زبان کودا را شرح می‌دهیم.

۲-۹-۱ تخصیص حافظه

در برنامه معمولی برای جمع دو بردار به زبان سی، تخصیص حافظه، خواندن بردارهای A و B و تعداد درایه‌های آن‌ها (N) از ورودی و سپس عملیات جمع هر یک از درایه‌ها به صورت ترتیبی با استفاده از یک حلقه انجام می‌شود.

برنامه ۲-۲: جمع دو بردار به زبان سی

```
تابع جمع دو بردار                                                    ۱
void vecAdd (float* h_A, float* h_B, float* h_C, int n)              ۲
{                                                                    ۳
int i;                                                               ۴
for (i=0, i<n; i++) h_C[i] = h_A[i] + h_B[i];                        ۵
}                                                                    ۶
تابع اصلی                                                            ۷
int main()                                                           ۸
{                                                                    ۹
تخصیص حافظه برای متغیرهای h_A، h_B و h_C                            ۱۰
عملیات ورودی/خروجی برای خواندن h_A و h_B                            ۱۱
...                                                                  ۱۲
vecAdd (h_A, h_B, h_C, N);                                           ۱۳
}                                                                    ۱۴
```

اکنون نحوه تبدیل نظام‌مند این برنامه را به کد کودا شرح می‌دهیم. به جای اجرای محاسبات، تابع VecAdd یک تابع کرنل را فراخوانی می‌کند که این تابع روی جی‌پی‌یو اجرا می‌گردد. این تابع قبل از فراخوانی کرنِل، باید عملیات برون‌سپاری از قبیل تخصیص حافظه برای متغیرها در جی‌پی‌یو و ارسال این متغیرها به آن را انجام دهد. سرانجام پس از اتمام محاسبات در جی‌پی‌یو، خروجی جواب باید از دستگاه به میزبان بازگردانده شود.

برنامه ۲-۳: شبه کد جمع دو بردار با استفاده از کودا

```
#include <cuda.h>                                                    ۱
```

الگوریتم ۲-۱ الگوریتم جمع دو بردار با استفاده از کودا و فراخوانی کرنل

ورودی: دو برداری که قرار است جمع شوند و همچنین تعداد اعضای آن‌ها

خروجی: بردار حاصل جمع دو بردار ورودی

تعریف متغیرهای لازم برای ارسال به جی‌پی‌یو

تخصیص حافظه روی جی‌پی‌یو برای متغیرهای ورودی و خروجی و ارسال آن‌ها به جی‌پی‌یو

فراخوانی و اجرای تابع کرنل

ارسال بردار جواب از جی‌پی‌یو به میزبان

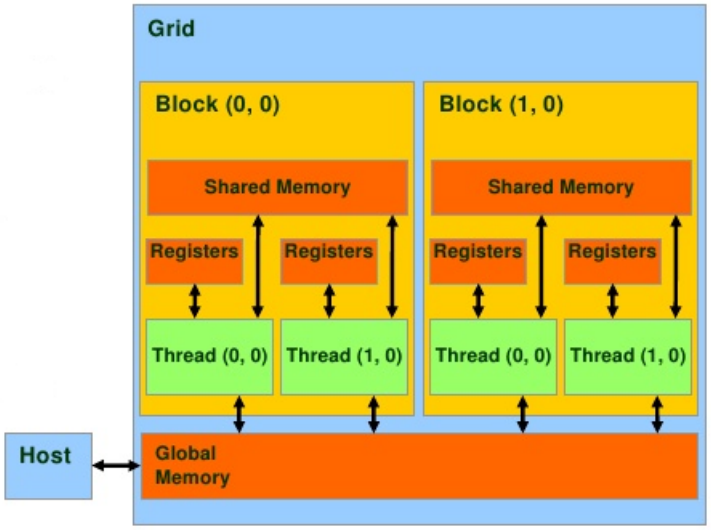
```
void vecAdd (float* h_A, float* h_B, float* h_C, int n)
{
int size = n* sizeof(float);
float* d_A, d_B, d_C;

// Allocate device memory for A,B and C
// copy A and B to device memory

// Kernel launch code - the device performs the actual vector
    addition

// Copy C from the device memory back to host
}
```

برای درک کامل آن‌چه در توابع رابط برنامه‌نویسی کاربردی [۱] رخ می‌دهد، باید فهم ادراکی از حافظه‌های کودا داشت. در تصویر (۳-۴)، همه انواع حافظه کودا در جی‌پی‌یو نشان داده نشده‌است. تنها دو قسمت مهم با هدف فهم ای‌پی‌آی نمایش داده شده است. در یک کرنل برخی از دسترسی‌ها به حافظه اشتراکی خواهد بود. نکته مهم در اینجا این است که کد میزبان می‌تواند در حافظه سراسری جی‌پی‌یو حافظه تخصیص دهد و هم‌چنین درخواست دریافت از این حافظه را داشته‌باشد.



شکل ۲-۱۷: ارتباط حافظه جی‌پی‌یو و میزبان

[۱] Application Programming Interface (API)

۲-۹-۱-۱ توابع مدیریت حافظه کودا

اصلی‌ترین توابع رابط برنامه‌نویسی کاربردی کودا، cudaMalloc() و cudaFree() هستند. تابع cudaMalloc() در حافظه سراسری جی‌پی‌یو برای اشیاء یا متغیرها حافظه تخصیص می‌دهد. این تابع ۲ مولفه دریافت می‌کند. یکی آدرس اشاره‌گر اختصاص داده شده به شیء و دیگری اندازه حافظه اختصاص داده شده به واحد بایت است. تفاوت این تابع با تابع Malloc() در زبان سی این است که تابع Malloc() یک مقدار اشاره‌گر را به شیء تخصیصی برمی‌گرداند اما در این‌جا در واقع آدرس یک اشاره‌گر برگردانده می‌شود و بعد یک تابع تخصیص این مقدار را به اشاره‌گر بسط می‌دهد. بنابراین این یک فعالیت فراخوانی با ارجاع [۱] است. دلیل این تفاوت این است که همه توابع رابط برنامه‌نویسی کاربردی کودا کدهای خطا برمی‌گردانند لذا تنها راهی که تابع cudaMalloc() اشاره‌گر مربوطه را به شیء تخصیص داده‌شده برگرداند، اجرای این قرارداد فراخوانی با ارجاع است.

دومین تابع، cudaFree() است. این تابع شیء را از حافظه سراسری آزاد می‌سازد تا فضای حافظه امکان بازیافت داشته‌باشد. تنها یک پارامتر دارد که اشاره‌گر به شیء آزاد شده‌است. مولفه در cudaFree() ، اشاره‌گر به شیء آزاد شده‌است در حالی که اولین مولفه در cudaMalloc() ، آدرس اشاره‌گر به شیء تخصیص داده‌شده است.

۲-۹-۱-۲ تابع انتقال داده

cudaMemcpy() تابع انتقال داده است که نیاز به ۴ مولفه دارد. اولین پارامتر، اشاره‌گر به مقصد است. دومین مولفه اشاره‌گر به منبع داده، سومی تعداد بایت‌های که باید کپی شود و چهارمی نوع یا جهت انتقال است که معمولا از متغیرهای ثابت از پیش تعریف شده برای آن استفاده می‌شود. انتقال به جی‌پی‌یو توسط این تابع ناهمگام [۲] است. بدین معنا که می‌توان یک انتقال با فراخوانی cudaMemcpy() درخواست کرد اما این تابع در همان لحظه و قبل از آن‌که عملیات انتقال کامل شود، بازمی‌گردد تا بتوان بلافاصله یک عملیات cudaMemcpy() دیگر انجام داد.

اکنون می‌توان برنامه (۲-۳) را تکمیل کرد و کد سمت میزبان را به صورت کامل نمایش داد:

برنامه ۲-۴: کد سمت میزبان جمع دو بردار با استفاده از کودا

```
#include <cuda.h>                                                    ۱
void vecAdd (float* h_A, float* h_B, float* h_C, int n)              ۲
{                                                                    ۳
int size = n* sizeof(float);                                         ۴
float* d_A, d_B, d_C;                                                ۵
                                                                     ۶
cudaMalloc((void **) &d_A, size);                                    ۷
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);                  ۸
cudaMalloc((void **) &d_B, size);                                    ۹
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);                  ۱۰
cudaMalloc((void **) &d_C, size);                                    ۱۱
                                                                     ۱۲
```

```
                                                                     ۱۵
```

```
// Kernel invocation code -to be shown later                    ۱۳
                                                                ۱۴
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);             ۱۶
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);                    ۱۷
}                                                               ۱۸
```

برای رسیدن به عملکرد بهتر نسبت به اجرای برنامه روی میزبان، تحمل هزینه تبادل داده قبل و بعد از هر کرنل غیرقابل قبول است. لذا در برنامه‌های واقعی تمایل داریم که مقدار زیادی داده در حافظه جی‌پی‌یو مستقر باشد و ما دائم کرنل‌ها را روی حافظه جی‌پی‌یو فراخوانی نماییم و نیازی به بازگرداندن جواب‌ها به میزبان وجود نداشته‌باشد چرا که این جواب‌ها برای استفاده در آینده روی حافظه جی‌پی‌یو باقی می‌مانند.

۲-۹-۱-۳ بررسی خطا در کودا

در برنامه (۲-۴) تابع cudaMalloc() فقط فراخوانی شده. اما در عمل امکان بررسی خطا در توابع کودا وجود دارد و بهتر است انجام شود. یک متغیر از نوع cudaError_t تعریف می‌شود که یک نوع از پیش تعریف شده در رابط برنامه‌نویسی کاربردی کودا می‌باشد و در فایل cuda.h ذخیره شده‌است. متغیر تعریف شده را به تابع cudaMalloc() نسبت می‌دهیم. با اجرای تابع، خطا بررسی می‌شود. اگر تابع با موفقیت اجرا نشود، باید بررسی لازم صورت گیرد و علت مشکل پیدا شود. در اکثر مواقع، علت بروز خطا در تخصیص حافظه عدم وجود حافظه کافی برای پاسخ به درخواست تخصیص است. برای دریافت پیام خطا از دستور cudaGetErrorString استفاده می‌شود. این تابع پیغام خطا را به یک رشته قابل خواندن توسط انسان تبدیل می‌کند و بعد درست مانند توابع استاندارد سی از دستور __File__ و __Line__ برای چاپ محل بروز خطا استفاده می‌شود. همچنین با استفاده از تابع exit از برنامه خارج می‌شود. در واقع با این کار از ادامه برنامه با وجود بروز خطا جلوگیری می‌شود و امکان خطایابی و رفع مشکل وجود خواهد داشت.

برنامه ۲-۵: بررسی بروز خطا در کودا

```
cudaError_t err = cudaMallox((void**) &d_A, size);              ۱
                                                                ۲
if (err != cudaSuccess) {                                       ۳
   printf("%s in %s at line %d\n" , cudaGetErrorString(err),    ۴
      __FILE__, __LINE__);
   exit(EXIT_FAILIURE);                                         ۵
}                                                               ۶
```

۲-۹-۲ برنامه‌نویسی موازی اس‌پی‌ام‌دی مبتنی بر کرنل

در این بخش مفاهیم ساده تابع کرنل کودا از قبیل نحوه تعریف یک کرنل، چگونگی استفاده از متغیرهای داخلی [۱] و همچنین نگاشت اندیس رشته به اندیس داده شرح داده می‌شود. یک مثال ساده از نحوه تعریف یک کرنل کودا در برنامه (۲-۵)ملاحظه

[۱] Built-in

می‌شود. این تابع همان کرنل جمع دو بردار است که در کد میزبان فراخوانی می‌شود. قسمت‌های متعددی در این کرنل وجود دارد. اولین بخش، کلمه کلیدی است. هر تابع کرنل کودا باید به دنبال عبارت __global__ تعریف شود. این عبارت به کامپایلر علامت می‌دهد که تابع تعریف شده یک تابع کرنل است. باقی سر انداز [1] تعریف تابع همانند تعریف یک تابع سی است.

برنامه ۲-۶: کرنل جمع دو بردار

```
__global__ void vecAddKernel(float* A, float* B, float* C, int n)
{
int i = threadIdx.x + blockDim.x*blockIdx.x;
if(I<n) c[i] = A[i] + B[i];
}
```

در درون کرنل، امکان استفاده از تعداد زیادی متغیرهای داخلی وجود دارد. متغیرهایی که در این مثال استفاده شده‌اند، threadIdx.x، blockDim.x و blockIdx.x هستند. این متغیرهای داخلی بوسیله سخت‌افزار مقداردهی اولیه می‌شوند تا هر رشته بتواند به این متغیرها دسترسی داشته‌باشد. بسته به اندیس رشته و اندیس بلاک، هر رشته ترکیب متفاوتی از مقادیر اندیس رشته و اندیس بلاک را خواهد داشت. به همین دلیل امکان تعیین مقدار متفاوتی از i برای رشته‌های مختلف وجود دارد. در نهایت می‌توان از i به عنوان یک اندیس داده برای دسترسی به داده ورودی استفاده کرد.

در برنامه (۲-۷) کد راه‌اندازی (فراخوانی) کرنل در برنامه میزبان مشاهده می‌شود. این فراخوانی بسیار مشابه فراخوانی توابع سی است. اما بین نام تابع و پارامترها نیاز به عرضه پارامترهای پیکربندی به کرنل وجود دارد. در واقع دو پارامتر پیکربندی وجود دارد. اولی تعداد بلاک‌ها در توری و دومی تعداد رشته‌ها در یک بلاک را تعیین می‌کند. در مثال برنامه (۲-۷) به کرنل اعلام شده که ۲۵۶ رشته در هر رشته بلاک لازم است و برای تعیین تعداد بلاک‌ها، تعداد درایه‌های هر یک از بردارهای جمع شونده (که هر دو باید مساوی باشند) تقسیم بر تعداد رشته‌های هر بلاک (۲۵۶) شده‌است و برای جلوگیری از اعشاری شدن این تقسیم، از تابع ceil استفاده شده‌است.

برنامه ۲-۷: فراخوانی کرنل جمع دو بردار در کد میزبان

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
// d_A, d_B, d_C allocations and copies omitted
// Run ceil(n/256.0) blocks of 256 threads each

vecAddKernnel<<<ceil(n/256.0), 256>>>(d_A,d_B,d_C,n);
}
```

در اینجا یک نکته مهم در نوشتن کرنل را به صورت یک مثال شرح می‌دهیم. اگر آرایه‌های ورودی ۱۰۰۰ عضوی باشند، برای بلاک‌های ۲۵۶ رشته‌ای ۴ رشته بلاک نیاز است و لذا ۱۰۲۴ رشته تخصیص داده می‌شود. آخرین رشته بلاک عضو ۹۹۹ام را در برمی‌گیرد چون شماره اجزاء از ۰ شروع می‌شود. بنابراین ۲۴ رشته خواهیم داشت که مقدار i برای آن‌ها از ۱۰۰۰ بیش‌تر خواهد بود. این رشته‌ها متناظر با هیچ یک از اجزاء ورودی‌ها و خروجی نخواهند بود و نیازی به عملیات توسط این رشته‌ها

[1] Header

وجود ندارد. بنابراین در برنامه کرنل (۲-۶) قبل از انجام عملیات جمع، مقدار i باید بررسی شود تا از ۱۰۰۰ تجاوز نکند.

برنامه (۲-۸) یک مفهوم دیگر در اجرای کرنل را نشان می‌دهد. می‌توان توری‌های دو یا سه‌بعدی داشت. در برنامه (۲-۷) تنها برای توری یک بعدی عمل می‌کند و هر یک از پارامترهای پیکربندی مقادیر صحیح هستند. اکنون پارامترهای سه‌بعدی را با استفاده از نوع dim3 تعریف می‌کنیم. در این نوع متغیر پیکربندی، سه مقدار صحیح برای هر یک از ابعاد وجود دارد. اولین مقدار مربوط به بعد x است و به همین ترتیب مقادیر دوم و سوم مربوط به ابعاد y و z می‌باشد. اگر تنها نیاز به توری‌های یک بعدی وجود داشته‌باشد، تنها بعد x مقداردهی می‌شود. بنابراین این نوع تعریف و اجرای کرنل روش عمومی‌تری است.

برنامه ۲-۸: تعریف سه‌بعدی پارامترهای پیکربندی در فراخوانی کرنل

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)                    ۱
{                                                                         ۲
dim3 DimGrid((n-1)/256+1 ,1, 1);                                          ۳
dim3 DimBlock(256, 1, 1);                                                 ۴
vecAddKernnel<<<DimGrid, DimBlock>>>(d_A, d_B, d_C, n);                   ۵
}                                                                         ۶
```

نحوه تعریف و نوشتن یک برنامه ساده به زبان کودا توضیح داده شد و سعی شد آشنایی ابتدایی از کودا در ذهن خواننده بوجود آید. راه‌کارهای بیش‌تری برای موازی‌سازی و نحوه انتخاب و تعداد رشته بلاک‌ها، مدیریت بیش‌تر حافظه و ...وجود دارد که در [۱۷] موجود است. همچنین برای عملیات ریاضی جبر خطی به زبان کودا، کتابخانه cuBLAS وجود دارد که نسخه کودا کتابخانه BLAS است. مستندات و نحوه استفاده از این کتابخانه در [۱۹] موجود می‌باشد. در فصل بعد به راه‌کارهای موازی‌سازی در متلب خواهیم پرداخت.

فصل ۳

موازی‌سازی در متلب

۳-۱ مقدمه

میلیون‌ها مهندس و دانشمند در سراسر جهان از متلب برای تجزیه و تحلیل و طراحی سامانه‌ها و محصولات با هدف تحول جهان هستی استفاده می‌کنند. زبان مبتنی بر ماتریس متلب غریزی‌ترین راه بیان ریاضیات محاسباتی است. متلب جعبه ابزار محاسبات موازی [۲] را فراهم ساخته که به کاربران امکان حل مسائل محاسباتی فشرده [۳] و اجرای سریع‌تر کدهایشان را با استفاده از پردازنده‌های چند هسته‌ای، خوشه های کامپیوتری و جی‌پی‌یوها می‌دهد.

در این فصل در مورد قابلیت‌ها و امکانات بهبود عملکرد و موازی‌سازی در متلب بحث خواهیم کرد. در مورد ابزار تجزیه تحلیل برنامه‌های متلب برای شناسایی بخش‌های زمان‌بر بحث خواهیم کرد، بردارایزه کردن کد را توضیح می‌دهیم، به برخی محدودیت‌ها و قوانین استفاده از parfor که ابزاری برای موازی‌سازی و بهره‌وری از هسته‌های سی‌پی‌یو هستند، خواهیم پرداخت و در پایان امکانات و قابلیت‌های قابل استفاده پردازنده گرافیکی (جی‌پی‌یو) در متلب را شرح می‌دهیم. عمده منابع این فصل، [۲۰] و [۱۹] می‌باشد.

۳-۲ ابزار نمایه‌ساز [۴] متلب

قبل از هر گونه اقدام جهت موازی‌سازی برنامه و تغییر کدها باید برنامه از نظر بار محاسباتی مورد تجزیه و تحلیل قرار گیرد. طبق مستندات متلب، عملیات نمایه‌سازی [۵] روشی است برای تعیین این مسئله که یک برنامه در کدام قسمت‌ها زمان بیش‌تری صرف نموده‌است. بعد از شناسایی توابعی که بیش‌ترین زمان را مصرف می‌کنند، می‌توان آن‌ها را برای امکان بهبود عملکرد ارزیابی نمود. این ابزار رابط کاربرای گرافیکی دارد و استفاده از آن بسیار آسان است. برای اجرای آن از دستور زیر

[۲] Parallel Computing Toolbox [۳] Computationally Intensive [۴] Profiler [۵] profiling

استفاده می‌کنیم:

```
profile viewer
```

در شکل (۳-۱)، نتیجه تحلیل برنامه تصاویر وضوح فوق‌العاده توسط ابزار نمایه‌ساز ملاحظه می‌شود. همان‌طور که واضح است، تابع امپی که مربوط به عملیات جستجوی تطابقی در بهینه‌سازی تنک است، بیش‌ترین زمان یعنی ۲۲ ثانیه از ۳۲ ثانیه زمان کل را به خود اختصاص داده چرا که ۱۷۶۴ بار فراخوانی شده. این تابع در تابع دیگری به نام L1SR فراخوانی

Profile Summary

Generated 05-Sep-2018 14:11:35 using performance time.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
Example	1	32.690 s	3.222 s	
L1SR	1	23.701 s	1.404 s	
MP	1764	22.136 s	22.136 s	
imshow	3	3.689 s	0.357 s	
initSize	3	2.563 s	0.161 s	
movegui	3	1.953 s	0.079 s	
legacyMoveGUI	3	1.860 s	1.757 s	
imread	1	0.457 s	0.029 s	
addpath	2	0.450 s	0.028 s	
path	2	0.412 s	0.266 s	

شکل ۳-۱: تحلیل ابزار نمایه‌ساز متلب از برنامه بهبود کیفیت تصویر

شده‌است که در آن ۲ حلقه فور تودرتو وجود دارد که تابع امپی درون این حلقه تودرتو فراخوانی می‌شود. در جدول شکل (۳-۱) اگر هر یک از توابع را انتخاب کنیم، به صفحه تحلیل آن تابع هدایت می‌شویم که در آن تحلیل دقیق‌تری از تابع مربوطه ارائه شده‌است به‌طوری که بار زمانی خطوط تابع از توابع درونی تفکیک شده و در جداول مجزا نمایش داده می‌شوند. (شکل ۳-۲) همچنین یک جدول به نام نتایج تجزیه تحلیل کد [۱] وجود دارد که برخی اشکالات در کد از قبیل متغیرهای استفاده نشده و ...در آن به تفکیک شماره خط نشان داده شده‌است. (شکل ۳-۳) همچنین جدول نتایج پوشش [۲] انواع دسته‌بندی خطوط را مانند تعداد خطوط غیر کد (مثل کامنت یا خطوط خالی)، خطوط کدی که اجرا شد و خطوطی که اجرا نشده و ...نمایش می‌دهد. در بخش انتهایی نیز خود کدها به همراه زمان صرف شده برای هر خط نمایش داده می‌شود و خطوطی که میزان مصرف زمانی بالایی داشته‌اند را باتوجه به میزان مصرف زمانی با رنگ قرمز برجسته شده‌است.

[۱] Code Analyzer Results [۲] Cverage Results

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
119	`w = MP(Dl,y,.01);`	1764	22.173 s	93.6%	
54	`lImGll = conv2(mIm,hfl,'same')...`	1	0.532 s	2.2%	
97	`fprintf('.');`	1747	0.272 s	1.1%	
101	`mmean = mean(mpatch(:));`	1764	0.225 s	0.9%	
134	`hpatch = Dh*w*mnorm;`	1734	0.130 s	0.5%	
All other lines			0.369 s	1.6%	
Totals			23.701 s	100%	

Children (called functions)

Function Name	Function Type	Calls	Total Time	% Time	Time Plot
MP	function	1764	22.136 s	93.4%	
mean	function	1764	0.119 s	0.5%	
imresize	function	2	0.042 s	0.2%	
Self time (built-ins, overhead, etc.)			1.404 s	5.9%	
Totals			23.701 s	100%	

شکل ۳-۲: تحلیل نمایه‌ساز از تابع L1SR به تفکیک خطوط و توابع

Code Analyzer results

Line number	Message
1	`Input argument 'lambda' might be unused, although a later one is used. Consider replacing it by ~.`
44	`The value assigned here to 'mhg' appears to be unused. Consider replacing it by ~.`
44	`The value assigned here to 'mwd' appears to be unused. Consider replacing it by ~.`
142	`INV(A)*b can be slower and less accurate than A\b. Consider using A\b for INV(A)*b or b/A for b*INV(A).`

شکل ۳-۳: تجزیه تحلیل کد تابع L1SR در نمایه‌ساز

۳-۳ بردارایزه کردن [1]

متلب برای عملیاتی بهینه‌سازی شده‌است که شامل ماتریس‌ها و بردارها باشد. عملیات بازنگری کدهای مبتنی بر حلقه و اسکالرگرا برای استفاده از عملیات ماتریسی و برداری متلب را بردارایزه کردن گویند. بردارایزه کردن کد به دلایل متعددی ارزشمند است.

- ظاهر : کد ریاضی بردارایزه شده شباهت بیش‌تری به عبارات ریاضی که در کتاب‌ها یافت می‌شوند دارند که فهم کد را آسان‌تر می‌سازد.
- مستعد خطای کمتر : با حذف حلقه‌ها، کد بردارایزه شده معمولا کوتاه‌تر است. خطوط کد کم‌تر به معنی احتمال

[1] Vectorization

بروز خطای برنامه‌نویسی کم‌تر است.

- عملکرد: کد برداریزه معمولا بسیار سریع‌تر از کدهای متناظر حاوی حلقه است.

مثال ۳-۳-۱ (برداریزه کردن کد برای محاسبات عمومی). کد زیر سینوس ۱۰۰۱ عدد بین ۰ تا ۱۰ را محاسبه می‌کند:

```
i = 0;                                                    ۱
for t = 0:.01:10                                          ۲
    i = i + 1;                                            ۳
    y(i) = sin(t);                                        ۴
end                                                       ۵
```

کد برداریزه شده کد بالا به صورت زیر است:

```
t = 0:.01:10;                                             ۱
y = sin(t);                                               ۲
```

نمونه کد دوم، معمولا سریع‌تر از نمونه اول اجرا می‌شود و استفاده اصولی‌تر از متلب است. هر دو کد در متلب تست شد، که اولی ۰.۰۰۸ و دومی ۰.۰۰۵ ثانیه زمان برد.

مثال ۳-۳-۲ (برداریزه کردن کد برای برخی دستورات خاص). کد زیر جمع تجمعی یک بردار در هر پنج درایه را محاسبه می‌کند:

```
x = 1:10000;                                              ۱
ylength = ( length(x) - mod(length(x),5)/5;               ۲
y(1:ylength) = 0;                                         ۳
for n = 5:5:length(x)                                     ۴
   y(n/5) = sum(x(1:n));                                  ۵
end                                                       ۶
```

با استفاده از برداریزه کردن، می‌توان عملیات متلب بسیار مختصرتری نوشت. کد زیر یک روش برای رسیدن به این هدف را نشان می‌دهد:

```
x = 1:10000;                                              ۱
xsums = cumsum(x); % returns the cumulative sum of x      ۲
y = xsums(5:5:length(x));                                 ۳
```

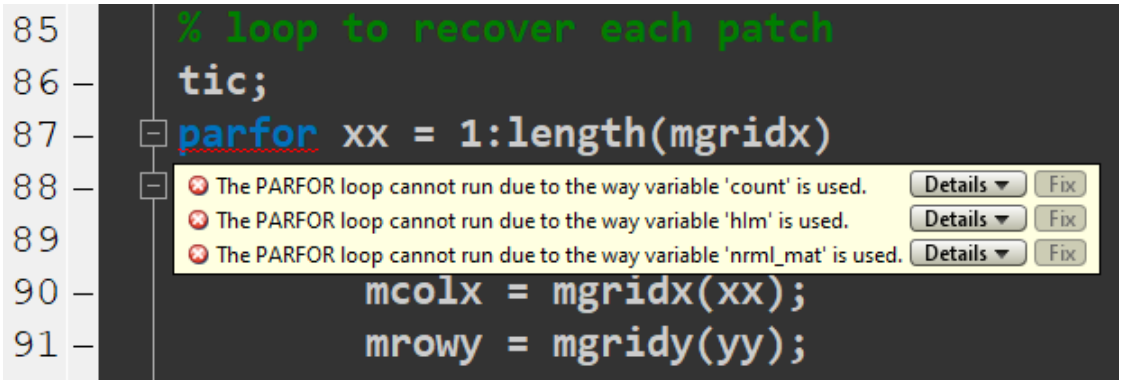
۳-۴ استفاده از حلقه parfor

parfor برای اجرای حلقه‌های فور به صورت موازی روی کارگرها [۱] در یک استخر موازی [۲] استفاده می‌شود. زمانی که کد بررسی شود و حلقه‌های فور کند شناسایی گردد، برای افزایش توان عملیاتی از parfor استفاده می‌شود.

[۱] worker [۲] parallel pool

اما به راحتی و با تغییر عبارت فور به parfor خصوصا در برنامه‌های پیچیده و دارای حلقه‌هایی که عملیات متعددی در درون آن‌ها انجام می‌شود، امکان موازی‌سازی و بهبود عملکر پدید نمی‌آید. چرا که استفاده از parfor و در کل امکانات موازی متلب دارای اصول، قواعد و محدودیت‌هایی است که باید برای استفاده از این امکانات به طور دقیق مورد مطالعه و بررسی قرار گیرد. در ادامه به تحلیل و بررسی مستندات متلب در مورد این حلقه‌ها می‌پردازیم.

۳-۴-۱ چه زمانی از parfor استفاده کنیم؟



شکل ۳-۴: خطای کارگزار متلب در تبدیل حلقه فور به parfor

کارگزار متلب [۱] به فرمان parfor رسیدگی می‌کند و در هماهنگی با کارگرهای متلب تکرارهای حلقه را در یک استخر موازی روی کارگرها به صورت موازی اجرا می‌کند. اگر یک حلقه فور آهسته و کند داشته‌باشیم، یک حلقه parfor می‌تواند مناسب باشد . موارد استفاده حلقه parfor به شرح ذیل می‌باشد:

- برخی حلقه‌های تکرار زمان زیادی برای اجرا شدن صرف می‌کنند. در این مورد، کارگرها می‌توانند تکرارها را به صورت همزمان اجرا کنند. باید اطمینان حاصل شود که تعداد تکرارها از تعداد کارگرها بیش‌تر باشد. در غیر این صورت از همه کارگرهای موجود استفاده نخواهد شد.
- تعداد تکرارهای متعدد از محاسبات ساده مثل شبیه‌سازی مونته‌کارلو. parfor حلقه‌های تکرار را به گروه‌هایی تقسیم می‌کند تا هر کارگر قسمتی از کل تعداد تکرارها را اجرا کند.

مواردی که حلقه parfor کارایی ندارد عبارتند از:

- کدی داشته‌باشیم که حلقه‌های فور را برداریزه [۲] کرده‌است. به طور کلی اگر بخواهیم کد سریع‌تر اجرا شود، اول از همه باید سعی کنیم آن را برداریزه نماییم. برداریزه کردن کد اجازه می‌دهد از موازیزاسیون فراهم‌شده توسط طبیعت چندهسته‌ای بسیاری از کتابخانه‌های اصلی متلب بهره برد. گرچه، اگر کد برداریزه شده و تنها به کارگرهای محلی

[۱] MATLAB Client [۲] vectorization

دسترسی وجود دارد، حلقه‌های parfor ممکن است آهسته‌تر از حلقه‌های فور عمل کنند. برای استفاده از حلقه‌های parfor نباید کد را از حالت برداریزه به حلقه تبدیل کرد [۱] چرا که این راهکار به خوبی کار نمی‌کند.

- حلقه‌های تکرار که زمان اندکی برای اجرا دارند. در این حالت، سرانه موازی بر محاسبات شما غالب است.

نمی‌توان زمانی که یک تکرار به تکرار قبل وابسته است از یک حلقه parfor استفاده کرد. استثنا در این قاعده جمع‌آوری مقادیر در یک حلقه با استفاده از متغیرهای کاهش است.

در تصمیم‌گیری در مورد این‌که چه زمانی از parfor استفاده شود باید سربار موازی‌سازی را درنظر گرفت. سربار موازی‌سازی شامل زمان مورد نیاز برای ارتباط، هماهنگی و انتقال داده (ارسال و دریافت اطلاعات از کارگزار تا کارگرها و برعکس) می‌شود. اگر ارزیابی‌های تکرار سریع باشند، این سربار می‌تواند بخش قابل‌توجهی از زمان کلی باشد. دو نوع مختلف از تکرارها را در نظر بگیرید:

- حلقه‌های فور دارای یک عملیات با بار محاسباتی بالا. این حلقه‌ها به طور کلی نماینده‌های خوبی برای تبدیل به حلقه‌های parfor هستند، چرا که زمان مورد نیاز برای محاسبات بر زمان انتقال داده چیره می‌شود.
- حلقه‌های فور با عملیات محاسباتی ساده. این حلقه‌ها عموما از تبدیل شدن به یک حلقه parfor سودی نمی‌برند، چرا که زمان مورد نیاز برای انتقال داده‌ها در مقایسه با زمان مورد نیاز برای محاسبات قابل‌توجه است.

۳-۴-۲ تبدیل حلقه‌های فور به حلقه‌های parfor

در برخی موارد برای تبدیل حلقه‌های فور به حلقه‌های parfor باید کد را تغییر داد.

مثال ۳-۴-۱. این مثال نشان می‌دهد چگونه اشکالات حلقه‌های parfor را با استفاده از یک حلقه ساده داخلی فور شناسایی و برطرف کنیم.

```
for x = 0:0.1:1                                    ۱
   for k = 2:10                                    ۲
    x(k) = x(k-1) + k;                             ۳
   end                                             ۴
   x                                               ۵
end                                                ۶
```

برای سرعت دادن به کد، سعی می‌کنیم حلقه فور را به parfor تبدیل کنیم. اما کد زیر خطا تولید می‌کند:

```
parfor x = 0:0.1:1                                 ۱
   parfor k = 2:10                                 ۲
    x(k) = x(k-1) + k;                             ۳
   end                                             ۴
   x                                               ۵
end                                                ۶
```

[۱] devectorize

بنابراین نمی‌توان به سادگی و بدون تغییرات حلقه فور را به parfor تغییر داد. برای رفع مشکل، باید در موارد بسیاری کد را تغییر داد. برای شناسایی مشکل، به دنبال پیام‌های کاوشگر کد در ویرایشگر متلب می‌گردیم. این کد اشکالات رایج در زمان اقدام برای تبدیل حلقه‌های فور به parfor را نشان می‌دهد:

- متغیر حلقه غیر صحیح
- حلقه‌های تودرتو موازی
- بدنه حلقه وابسته

برای رفع این مسائل، باید برای استفاده از parfor کد را تغییر داد. بدنه حلقه parfor در یک استخر موازی با استفاده کارگرهای متعدد در یک ترتیب غیر قطعی اجرا می‌شوند. بنابراین، باید این الزامات را برای بدنه حلقه parfor برآورده کرد:

۱. بدنه حلقه parfor باید مستقل باشد. یک تکرار حلقه نمی‌تواند به یک تکرار قبلی وابسته باشد، چرا که تکرارها به طور موازی در یک ترتیب غیرقطعی اجرا می‌شوند. در مثال زیر:

```
x(k) = x(k-1) + k;
```

استقلال وجود ندارد، و لذا نمی‌توان از parfor استفاده کرد.

۲. نمی‌توان از یک حلقه parfor در داخل یک حلقه parfor دیگر استفاده کرد. مثال ۳-۴-۱ دو حلقه parfor تودرتو دارد، و لذا فقط می‌توان یک حلقه فور را به حلقه parfor تبدیل کرد. در عوض می‌توان یک تابع که از یک حلقه parfor در درونش استفاده شده را در داخل بدنه یکه حلقه parfor دیگر به کار برد. گرچه، چنین حلقه‌های تودرتو parfor هیچ امتیاز محاسباتی ندارند چرا که همه کارگرها برای موازی‌سازی خارجی‌ترین حلقه استفاده می‌شوند.

۳. حلقه‌های parfor باید اعداد صحیح متوالی باشند. در مثال زیر:

```
parfor x = 0:0.1:1
```

متغیرهای حلقه غیرصحیح هستند. پس در این‌جا نمی‌توان از parfor استفاده کرد. می‌توان با تغییر متغیر حلقه به مقادیر صحیح مورد نیاز الگوریتم، این مشکل را حل کرد.

۴. برخلاف یک حلقه فور نمی‌توان از یک حلقه parfor زودتر از موعد خارج شد. در بدنه حلقه parfor نباید از عبارات `break` یا `return` استفاده کرد. بدون برقراری ارتباط، سایر نمونه‌های متلب اجرا کننده حلقه نمی‌دانند در چه زمانی متوقف شوند. می‌توان به عنوان جایگزین `parfeva` را در نظر گرفت.

۳-۴-۳ اطمینان از مستقل بودن تکرارها در حلقه parfor

اگر زمانی که حلقه‌های فور را به حلقه‌های parfor تبدیل می‌کنیم، با خطا مواجه شدیم، باید اطمینان حاصل کنیم که تکرارهای parfor مستقل هستند. تکرارهای حلقه parfor برخلاف تکرارهای حلقه‌های فور که به صورت ترتیبی هستند، هیچ ترتیب تضمین شده‌ای ندارند. همچنین تکرارهای حلقه parfor در کارگرهای متلب مختلفی در استخر موازی اجرا می‌شوند، بنابراین هیچ اطلاعات مشترکی بین تکرارها وجود ندارد. تنها استثناء برای این قانون جمع‌کردن مقادیر در یک حلقه با استفاده از متغیرهای کاهشی است.

مثال ۳-۴-۲. در کد زیر، هر عضو A برابر با شاخص متناظرش است. به همین دلیل حلقه parfor هم برای این مثال کار می‌کند. حلقه‌های فور با وظایف مستقل کاندیدهای ایده‌آلی برای حلقه‌های parfor هستند.

```
for i = 1:8
   A(i) = i;
end
A
```

اما در کد زیر از یک متغیر غیرشاخص یا متغیری که شاخص‌دهی آن به متغیر حلقه i وابسته نیست در داخل حلقه استفاده شده.

```
d = 0; i = 0;
for i = 1:4
   d = i*2;
   A(i) = d;
end
A
d
i
```

خروجی برای حلقه فور به صورت زیر است:

```
A =

     2     4     6     8

d =

     8

i =

     4
```

در حالی‌که خروجی برای حلقه parfor به صورت زیر است:

```
A =
```

```
     2     4     6     8

d =

     0

i =

     0
```

با وجود این‌که مقادیر A در هر دو مثال یکسان است، در مورد متغیر d این گونه نیست. در حلقه فور تکرارها به صورت سریالی اجرا شده‌اند. لذا در پایان، d مقداری که در آخرین تکرار در حلقه داشته را در خود دارد. در حلقه parfor اما تکرارها به صورت موازی اجرا می‌شوند پس غیرممکن است که یک مقدار تعریف شده به d در پایان حلقه اختصاص یابد. این وضعیت همچنین برای متغیر i نیز صدق می‌کند. بنابراین رفتار حلقه parfor به گونه‌ای تعریف شده که تاثیری در مقادیر d و i در خارج از حلقه ندارد. مقادیر آن‌ها قبل و بعد از حلقه یکسان باقی می‌ماند. اگر متغیرها در حلقه parfor مستقل نباشند، آن‌گاه ممکن است پاسخ‌های متفاوتی نسبت به حلقه فور دریافت شود. همه کدهایی که در ادامه یک عبارت parfor می‌آیند، نباید به ترتیب تکرار حلقه وابسته باشند. آنالیزگر کد فقط می‌تواند در تشخیص مستقل بودن تکرارهای حلقه کمک کند.

۳-۴-۴ متغیرهای برش داده‌شده [۱]

یک متغیر برش داده‌شده متغیری‌ست که مقدارش می‌تواند به بخش‌ها یا قطعه‌هایی تقسیم شود که هر یک بعدا به طور مستقل توسط کارگرها مورد پردازش قرار می‌گیرند. هر تکرار حلقه روی قطعه مجزایی از آن آرایه کار می‌کند. استفاده از متغیرهای قطعه‌شده می‌تواند ارتباط بین کارگزار و کارگرها را کاهش دهد. وقتی که یک کارگر شروع به کار بر روی یک محدوده اندیس می‌کند، تنها آن قطعاتی که توسط آن کارگر مورد نیاز است به آن ارسال می‌شود.

مثال ۳-۴-۳. در کد زیر یک قطعه از A شامل یک درایه تکی از این آرایه است:

```
parfor i = 1:length(A)
   B(i) = f(A(i));
end
```

۳-۴-۴-۱ خصوصیات یک متغیر برش داده‌شده

اگر یک حلقه parfor همه خصوصیات زیر را دارا باشد، آن‌گاه متغیر برش‌داده‌شده خواهد بود:

[۱] Sliced

- نوع اندیس‌دهی سطح اول – سطح اول اندیس‌دهی یا پرانتز است و یا براکت.
- فهرست اندیس ثابت – در پرانتز یا براکت سطح اول، فهرست اندیس‌ها برای همه رخدادهای یکه متغیر ثابت است.
- نحوه اندیس‌دهی – در فهرست اندیس‌ها برای یک متغیر، دقیقا یک اندیس شامل متغیر حلقه است.
- شکل آرایه – آرایه شکل ثابتی را حفظ می‌کند. در تخصیص به یک متغیر برش داده‌شده، سمت راست عبارت نمی‌تواند تهی [ ] یا '' باشد، چون این عملگرها اقدام به حذف درایه‌ها می‌کنند.

۳-۴-۴-۲ نوع اندیس‌دهی سطح اول

بعد از سطح اول می‌توان از هر نوع اندیس دهی مجاز متلب در سطح دوم و سطوح بعدی استفاده کرد. متغیر `A.q{i,12}` برش‌داده‌شده نیست اما `A{i,12}.q` برش داده شده است.

۳-۴-۴-۳ فهرست اندیس ثابت

در اندیس‌دهی سطح اول یک متغیر قطعه‌شده، فهرست اندیس‌ها برای همه رخدادهای یک متغیر یکسان است.

مثال ۳-۴-۴. متغیر `A` در کد زیر قطعه شده نیست چون `A` توسط `i` و `i+1` در محل‌های متفاوت اندیس‌دهی شده.

```
parfor i = 1:k                                                    ۱
   B(:) = h(A(i), A(i+1));                                        ۲
end                                                               ۳
```

اما در کد زیر، متغیر `A` به درستی قطعه شده است:

```
 parfor i = 1:k                                                   ۱
     B(:) = f(A(i));                                              ۲
     C(:) = g(A{i});                                              ۳
 end                                                              ۴
```

مثال ۳-۴-۵. در کد زیر ، `A` قطعه نمی‌شود چون اندیس‌دهی در همه جا یکسان نیست.

```
parfor i=1:10                                                     ۱
  b = A(1,i) + A(2,i)                                             ۲
end                                                               ۳
```

کد زیر هم `A` و هم `B` را قطعه می‌کند. اندیس‌دهی `A` با اندیس‌دهی `B` یکسان نیست. گرچه اندیس‌دهی برای هر یک به طور مجزا ثابت است.

```
A = [1 2 3 4 5 6 7 8 9 10 20 30 40 50 60 70 80 90 100];
B = zeros(1,10);
parfor i=1:10
      for n=1:2
          B(i) = B(i)+A(n,i)
      end
end
```

۳-۴-۴-۴ نحوه اندیس‌دهی

در فهرست اندیس‌ها برای یک متغیر برش داده‌شده، یکی از این اندیس‌ها به فرم i ، i+k ، i-k ، k+i یا k-i است. اندیس i متغیر حلقه است و k یک ثابت یا یک متغیر پخش است. هر اندیس دیگری یک ثابت اسکالر است، یک متغیر پخش، یک حلقه فور تودرتو، ، کولون یا end است.

مثال ۳-۴-۶. در کد زیر، با i به عنوان متغیر حلقه، متغیرهای A نشان داده‌شده در بخش اول برش داده‌شده نیستند، در حالی‌که متغیرهای A در قسمت پایین برش داده‌شده هستند:

```
%% NOT Sliced:
A(i+f(k),j,:,3) % f(k) invalid for slicing
A(i,20:30,end)  % 20:30 not scalar
A(i,:,s.field1) % s.field1 not simple broadcast var
%% Sliced:
A(i+k,j,:,3)
A(i,:,end)
A(i,:,k)
```

زمانی که متغیرهای دیگری همراه متغیر حلقه برای اندیس‌دهی یک آرایه استفاده می‌شود، نمی‌توان این متغیرها را در داخل حلقه تنظیم کرد. در واقع، چنین متغیرهایی در طول اجرای کل دستور parfor ثابت هستند. شما نمی‌توانید متغیر حلقه را با خودش ترکیب کنید تا یک عبارت اندیس را تشکیل دهید.

۵-۴-۴-۴ شکل آرایه

یک متغیر برش داده‌شده باید یک شکل ثابتی را حفظ کند. متغیر A در مثال زیر در هر دو خط برش داده‌شده نیست:

```
A(i,:) = [];
A(end + 1) = i;
```

در خط اول، A برش داده‌شده نیست زیرا تغییر شکل آرایه قطعه شده می تواند مفروضاتی را که در ارتباط بین کارگزار و کارگران قرار دارد را نقض کند. در خط دوم، A یک خروجی قطعه شده نیست چون روی متغیر حلقه اندیس‌دهی نشده.

۳-۴-۴-۶ متغیرهای برش داده شده ورودی و خروجی

تمامی متغیرهای قطعه‌شده، خصوصیات ورودی و خروجی بودن را دارا هستند. یک متغیر قطعه شده گاهی هر دو خصوصیات را دارد. متلب متغیرهای قطعه شده ورودی را از کارگزار به کارگرها منتقل می‌کند و متغیرهای قطعه‌شده خروجی را از کارگرها به کارگزار بازمی‌گرداند. اگر متغیری هم ورودی و هم خروجی باشد، در هر دو جهت رفت و آمد دارد. در حلقه parfor زیر، r یک متغیر قطعه‌شده وردی‌ست و b یک متغیر قطعه‌شده خروجی:

```
a = 0;                                                    ۱
z = 0;                                                    ۲
r = rand(1,10);                                           ۳
parfor ii = 1:10                                          ۴
   a = ii;                                                ۵
   z = z + ii;                                            ۶
   b(ii) = r(ii);                                         ۷
end                                                       ۸
```

گرچه، اگر در هر تکرار، هر اعلان به یک عضو آرایه قبل از آن‌که استفاده شود صورت بگیرد، متغیر یک متغیر قطعه‌شده ورودی نیست. در مثال زیر، همه درایه‌های A ابتدا مشخص شده و بعد فقط آن مقادیر ثابت مورد استفاده قرار گرفته:

```
parfor ii = 1:n                                           ۱
   if someCondition                                       ۲
      A(ii) = 32;                                         ۳
   else                                                   ۴
      A(ii) = 17;                                         ۵
   end                                                    ۶
   loop code that uses A(ii)                              ۷
end                                                       ۸
```

متغیرهای خروجی قطعه‌شده می توانند به صورت پویا از طریق تخصیص های نمایه شده با مقادیر پیش فرض در موقعیت های شاخص میانگین میانه رشد کنند. در این مثال متغیر قطعه‌شده، می‌توانید ببینید که مقدار پیش‌فرض صفر در بسیاری محل‌ها در a قرار داده‌شده:

```
a = [];                                                   ۱
parfor idx = 1:10                                         ۲
    if rand < 0.5                                         ۳
        a(idx) = idx;                                     ۴
    end                                                   ۵
end                                                       ۶
```

```
0     2     0     4     5     0     0     8     9     10
```

حتی اگر یک متغیر قطعه شده به صورت صریح به عنوان یک ورودی ارجاع داده نشود، استفاده ضمنی آن را می توان انجام داد. برای مثال کد زیر اقدام به گسترش مقدار idx

برای تخصیص به هر درایه بردار تعریف شده توسط x(:, idx) می‌کند. این کار خطا تولید می‌کند.

```
x = zeros(10,12);                                                    ۱
parfor idx = 1:12                                                    ۲
  x(:,idx) = idx;                                                    ۳
end                                                                  ۴
```

کد زیر یک راه حل پیشنهادی برای این محدودیت ارائه می دهد:

```
x = zeros(10,12);                                                    ۱
parfor idx = 1:12                                                    ۲
   x(:,idx) = repmat(idx,10,1);                                      ۳
end                                                                  ۴
```

۳-۵ استفاده از جی‌پی‌یو در متلب

علاوه بر کامپیوترهای چندهسته‌ای، چندپردازنده‌ای و خوشه‌های کامپیوتری، جعبه ابزار موازی متلب به کاربران اجازه می‌دهد مسائل محاسباتی سنگین را روی جی‌پی‌یوهای با قابلیت کودا حل کنند. محاسبات جی‌پی‌یو در متلب برای اولین بار در نسخه R2010b معرفی شد.

با پیشرفت‌هایی که در صنعت سخت‌افزار بوجود آمده، جی‌پی‌یوها محبوبیت بسیاری در دهه اخیر کسب کرده‌اند و به‌طور گسترده در کاربردهای محاسباتی فشرده استفاده شده‌اند. در حال حاضر دو روش برای برنامه‌نویسی روی جی‌پی‌یوها وجود دارد. کودا [۱] و اوپن‌سی‌ال [۲]. کودا بالغ‌تر و باثبات‌تر است. برای دسترسی به معماری کودا، یک برنامه‌نویس می‌تواند از زبان سی یا سی++ با استفاده از کوداسی یا فرترن استفاده کند. راه‌کارهایی برای استفاده از کودا در متلب با استفاده از اجرای کِرنل‌های کودا [۳] و همچنین فایل‌های مِکس [۴] وجود دارد که به آن‌ها خواهیم پرداخت.

ابتدایی‌ترین راه استفاده از gpuArray می‌باشد. با استفاده از این دستور می‌توان داده‌ها را به جی‌پی‌یو منتقل نمود. در واقع gpuArray یک نوع داده است و متغیری که از این نوع تعریف شود، روی جی‌پی‌یو بارگذاری می‌گردد. همچنین عملیات درایه‌ای با استفاده از توابع arrayfun, bsxfun و pagefun در صورتی که داده‌ها روی جی‌پی‌یو تعریف شده‌باشند و از نوع gpuArray باشند، سرعت اجرای برنامه بهبود می‌یابد. پس از آن راه‌کارهایی برای استفاده از کودا در متلب تعبیه شده که یکی استفاده از کِرنل‌های کودا در متلب است و دیگری استفاده از توابع MEX. در ادامه هر یک از این روش‌ها را مورد بررسی قرار می‌دهیم.

---

[۱] Compute Unified Device Architecture (CUDA) [۲] OpenCL [۳] CUDA Kernels [۴] MEX files

۳-۵-۱ شناسایی و انتخاب یک جی‌پی‌یو

برای استفاده از امکانات جی‌پی‌یو متلب، طبیعتا باید یک جی‌پی‌یو که تحت پشتیبانی متلب باشد، داشته‌باشیم. برای شناسایی تعداد جی‌پی‌یوهای تحت پشتیبانی، از دستور gpuDeviceCount استفاده می‌شود. تعداد جی‌پی‌یوها در سیستمی که از آن برای انجام آزمایش‌ها در این پایان‌نامه استفاده شده‌است، یک عدد می‌باشد. اگر تعداد جی‌پی‌یو ها بیش از یکی باشد، برای انتخاب هر یک از آن‌ها از دستور زیر استفاده می‌شود:

```
parallel.gpu.GPUDeviceManager.instance
```

حاصل استفاده از این دستور در خروجی به صورت زیر است:

```
ans =

  GPUDeviceManager with properties:

    SelectedDevice: [1×1 parallel.gpu.CUDADevice]

>>
```

با وارد کردن شماره جی‌پی‌یو مورد نظر انتخاب می‌شود.

۳-۵-۱-۱ بررسی اجمالی مشخصات جی‌پی‌یو

با استفاده از دستور gpuDevice می‌توان اطلاعات لازم درباره دستگاه جی‌پی‌یو انتخاب شده را در متلب بدست آورد. مشخصات جی‌پی‌یو استفاده شده در این پایان‌نامه به صورت زیر است:

برنامه ۳-۱: مشخصات جی‌پی‌یو در متلب GeForce GTX 1050

```
  CUDADevice with properties:

                      Name: 'GeForce GTX 1050'
                     Index: 1
         ComputeCapability: '6.1'
            SupportsDouble: 1
             DriverVersion: 9.2000
            ToolkitVersion: 9
         MaxThreadsPerBlock: 1024
           MaxShmemPerBlock: 49152
         MaxThreadBlockSize: [1024 1024 64]
                MaxGridSize: [2.1475e+09 65535 65535]
                  SIMDWidth: 32
                TotalMemory: 4.2950e+09
            AvailableMemory: 3.4633e+09
       MultiprocessorCount: 5
              ClockRateKHz: 1493000
               ComputeMode: 'Default'
```

```
    GPUOverlapsTransfers: 1
  KernelExecutionTimeout: 1
       CanMapHostMemory: 1
        DeviceSupported: 1
         DeviceSelected: 1
```

در ادامه برخی از این خواص را طبق مستندات شرکت NVIDIA و مستندات متلب شرح می‌دهیم.

- توانایی محاسبه [۱] یک جی‌پی‌یو، مشخصات عمومی و ویژگی‌های در دسترس آن را آشکار می‌سازد. توانایی محاسبه گاهی با عنوان نسخه SM هم خطاب می‌گردد. نسخه توانایی محاسبه، ویژگی‌هایی که توسط سخت‌افزار جی‌پی‌یو در زمان اجرا پشتیبانی می‌شود را مشخص می‌کند تا تعیین شود چه ویژگی‌های سخت‌افزاری و دستوراتی برای یک جی‌پی‌یو وجود دارد.

  توانایی محاسبه از یک شماره ویرایش [۲] X و یک شماره اصلاحات جزئی [۳] Y تشکیل شده که به صورت $X.Y$ نشان داده می‌شود.

  دستگاه‌های با شماره ویرایش یکسان، متعلق به معماری هسته یکسانی هستند. عدد ۶ برای دستگاه‌های با معماری پاسکال [۴] است. شماره اصلاحات جزئی مربوط به یک بهبود افزایشی می‌باشد که احتمالا شامل ویژگی های جدید است.

- طبق مستندات متلب، گزینه SupportsDouble مربوط به پشتیبانی از عملیات با دقت دو برابر [۵] است. ممیز شناور به روشی گفته می‌شود که برای نمایش اعداد حقیقی به طوری که محدوده‌ای وسیع از مقادیر را بپذیرند، بکار می‌رود. در واقع محاسبات ریاضی با دقت شناوری تکی [۶] با اعداد ممیز شناور ۳۲ بیتی سر و کار دارد در حالی که عملیات با دقت دو برابر، با اعداد ۶۴ بیتی سر و کار دارند.

- نسخه درایور [۷] در واقع نسخه درایور جی‌پی‌یو کودا در حال استفاده را نشان می‌دهد. این نسخه باید تحت پشتیبانی نسخه متلب مورد استفاده باشد.

- گزینه بعدی نسخه ابزار [۸] است. این آیتم مربوط به نسخه ابزار کودا [۹] ست که توسط نسخه فعلی متلب استفاده می‌شود.

- گزینه MaxThreadsPerBlock : بیش‌ترین تعداد رشته‌هایی که در یک بلاک در طول اجرای کِرنل کودا می‌تواند استفاده شود .

- گزینه MaxShmemPerBlock : بیش‌ترین مقدار حافظه اشتراکی پشتیبانی شده که توسط یک رشته بلاک می‌تواند در طول اجرای کِرنل کودا استفاده شود.

---

[۱] Compute Capability [۲] Revision Number [۳] Minor Revision Number [۴] Pascal Architecture [۵] Double Precision Operations [۶] Single Precision Floating [۷] Driver Version [۸] Toolkit Version [۹] CUDA® Toolkit

- گزینه MaxThreadBlockSize : حداکثر اندازه یک بلاک در هر بُعد است. هر کدام از بُعدهای یک رشته بلاک نباید از این ابعاد تجاوز کند. همچنین، ضرب همه ابعاد رشته بلاک نباید از MaxThreadsPerBlock تجاوز کند. طبیعتا مقداری که در ۳-۱ برای هر بعد نشان داده شده، به شرطی قابل پشتیبانی است که ضربش در سایر ابعاد بلاک از حداکثر تعداد رشته‌های یک بلاک بیش‌تر نشود.

- گزینه MaxGridSize : حداکثر تعداد توری‌های متشکل از رشته بلاک ها است.

- گزینه MaxGridSize : تعداد رشته‌های اجرا شده به صورت همزمان است. همان‌طور که در فصل قبل توضیح داده شده، در واقعیت تعداد رشته‌هایی که همزمان اجرا می‌شوند محدود است. در کودا این گزینه به نام lrWarp شناخته می‌شود. یک lrWarp همواره به صورت همزمان روی یک SM (چندپردازنده جریانی) اجرا می‌شود. البته به شرطی که تعداد SPها (پردازنده‌های جریانی) در یک اس‌ام بیش‌تر از ۳۲ تا باشد.

- گزینه TotalMemory : حداکثر حافظه در دستگاه (جی‌پی‌یو) به بایت

- گزینه AvailableMemory : حداکثر میزان حافظه (به بایت) در دسترس برای داده است.

- گزینه MultiprocessorCount : تعداد هسته‌های برداری (چندپردازنده جریانی) موجود روی دستگاه

- گزینه ClockRateKHz : حداکثر نرخ ساعت جی‌پی‌یو را به کیلوهرتز نشان می‌دهد که مقدار نشان داده شده در ۳-۱، معادل ۱.۴۲ گیگاهرتز است. این بدین معناست که جی‌پی‌یو می‌تواند در ۱ ثانیه ۱۴۲ میلیون چرخه ساعت داشته‌باشد.

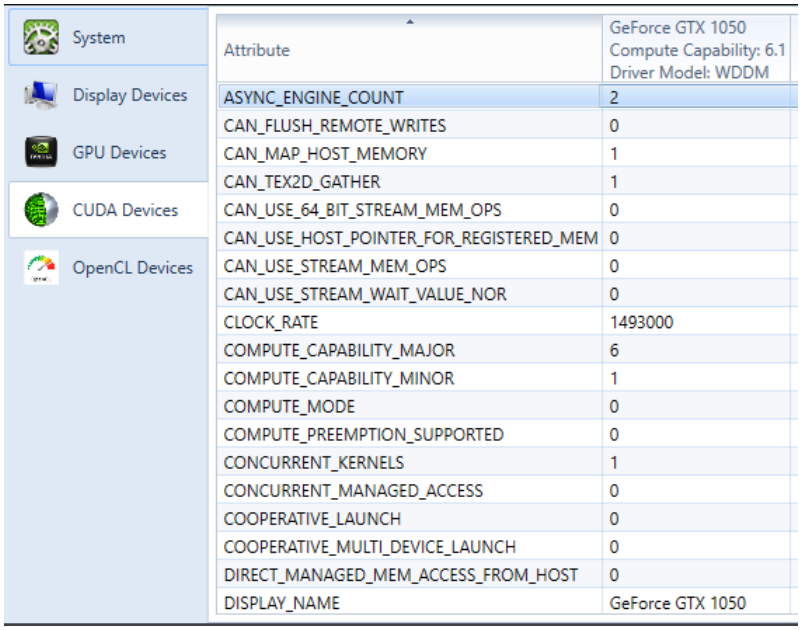
- گزینه ComputeMode : حالت محاسبه دستگاه یکی از مقادیر زیر است:

  - 'Default' : دستگاه در این حالت محدود شده نیست و می‌تواند توسط چندین برنامه به صورت هم‌زمان اجرا شود. متلب می‌تواند دستگاه را با سایر برنامه‌ها از جمله جلسات متلب [1] یا کارگرها، به اشتراک بگذارد.
  - 'Exclusive thread' یا 'Exclusive process' : دستگاه می‌تواند فقط توسط یک برنامه در یک زمان استفاده شود. وقتی دستگاه در متلب انتخاب شده، نمی‌تواند توسط سایر برنامه‌ها، شامل سایر جلسات و کارگرهای متلب، استفاده شود.
  - 'Prohibited' : دستگاه نمی‌تواند استفاده شود.

برخی مشخصات دیگر را هم می‌توان در قسمت system information در Nvidia Control Panel یافت. یک بخش مهم در این قسمت، تعداد هسته‌هاست که برای سیستم تست ۶۴۰ تاست. همچنین منو Nsight در Visual Studio، قسمت ویندوز، گزینه System Info هم اطلاعات کامل‌تری درباره مشخصات جی‌پی‌یو وجود دارد که بخشی از آن در

[1] MATLAB session

تصویر ۳-۵ ملاحظه می‌گردد. در منو GPU Devices (منوهای سمت چپ)، تعداد SMها ذکر شده که برای GeForce GTX 1050، ۵ تاست.



Attribute	GeForce GTX 1050 Compute Capability: 6.1 Driver Model: WDDM
ASYNC_ENGINE_COUNT	2
CAN_FLUSH_REMOTE_WRITES	0
CAN_MAP_HOST_MEMORY	1
CAN_TEX2D_GATHER	1
CAN_USE_64_BIT_STREAM_MEM_OPS	0
CAN_USE_HOST_POINTER_FOR_REGISTERED_MEM	0
CAN_USE_STREAM_MEM_OPS	0
CAN_USE_STREAM_WAIT_VALUE_NOR	0
CLOCK_RATE	1493000
COMPUTE_CAPABILITY_MAJOR	6
COMPUTE_CAPABILITY_MINOR	1
COMPUTE_MODE	0
COMPUTE_PREEMPTION_SUPPORTED	0
CONCURRENT_KERNELS	1
CONCURRENT_MANAGED_ACCESS	0
COOPERATIVE_LAUNCH	0
COOPERATIVE_MULTI_DEVICE_LAUNCH	0
DIRECT_MANAGED_MEM_ACCESS_FROM_HOST	0
DISPLAY_NAME	GeForce GTX 1050

شکل ۳-۵: بخشی از مشخصات جی‌پی‌یو در Nsight System Information

۳-۵-۲ ایجاد یک آرایه در جی‌پی‌یو

یک gpuArray در متلب، نشان دهنده یک آرایه است که روی جی‌پی‌یو ذخیره شده. از تابع gpuArray برای برای انتقال یک آرایه از متلب به جی‌پی‌یو استفاده می‌شود.

۳-۵-۳ عملیات درایه‌ای روی جی‌پی‌یو

یک روش دیگر برای استفاده از قابلیت‌های GPU استفاده از توابع arrayfun, bsxfun و pagefun است. به طور خلاصه این توابع عملیات ثابت که با یک تابع تعریف می‌شود را روی تک‌تک اعضای ماتریس‌ها یا صفحات ماتریس‌ها انجام می‌دهند. این توابع در حالت معمولی روی CPU اجرا می‌شوند اما اگر متغیرهای فراخوانی شده در آن‌ها از نوع gpuArray باشند، کلیه عملیات روی GPU انجام می‌شود که برای داده‌های حجیم در تئوری باید عملکرد بهتری داشته‌باشند.

۳-۵-۴ استفاده از کودا در متلب

برای استفاده از کودا در متلب دو روش وجود دارد که یکی انتقال کِرنل و اجرای آن در متلب است و دیگری تولید تابع MEX می‌باشد. برای استفاده از کرنل، باید فایل PTX کرنل تولید شود. کرنل در فایل با فرمت cu. ذخیره می‌شود. در متلب دستوری برای تولید فایل PTX وجود دارد. همچنین برای تولید فایل MEX، تابع به زبان سی در درون یک تابع MEX قرار داده می‌شود.

توضیحات کامل در این خصوص در [۱۸] موجود است.

فصل ۴

کارهای انجام شده

۴-۱ مقدمه

هدف اصلی این پایان‌نامه استفاده از روش‌ها و راه‌کارهای موازی‌سازی برای بهبود عملکرد عملیات بهینه‌سازی تنک و کاربرد آن در پردازش تصویر است. در این راستا تعدادی مقالات و پایان‌نامه‌های مرتبط مورد مطالعه و بررسی قرار گرفت و همچنین سعی شد کلیه روش‌ها و راه‌کارهای موازی‌سازی در عمل مورد آزمایش و بررسی قرار گیرد.

در خصوص به کارگیری عملی روش‌ها، برنامه تصاویر وضوح فوق‌العاده در مقاله [۲۱] که به زبان متلب نوشته شده‌است به عنوان هدف انتخاب شد. در این برنامه یک تصویر ورودی ابتدا کوچک می‌شود تا کیفیت آن افت کند. سپس در هر تکرار یک حلقه فور [۲] تودرتو، یک قطعه [۳] از تصویر جدا می‌شود و بعد عملیات بهینه‌سازی تنک با استفاده از یک دیکشنری با وضوح پایین در یک تابع روی قطعه انجام می‌شود. سپس با استفاده از دیکشنری وضوح بالا، قطعه با کیفیت بالا تولید می‌گردد. در پایان حلقه، این قطعه در محل مربوط به خود در تصویر خروجی قرار می‌گیرد. کلیه روش‌های موازی‌سازی در جعبه ابزار محاسبات موازی متلب مورد مطالعه دقیق قرار گرفت و سپس سعی شد این روش‌ها روی برنامه تصاویر وضوح فوق‌العاده اعمال شوند تا نتیجه بهبود در زمان اجرای این برنامه مورد تحلیل و بررسی قرار گیرد.

همچنین با توجه به این‌که یکی از جدیدترین و موثرترین روش‌های موازی‌سازی برنامه‌نویسی به روش کودا می‌باشد، مقدمات برنامه‌نویسی به زبان کودا سی مطالعه و فرا گرفته شد. سعی شد با استفاده از این روش زمان اجرای برنامه تصاویر وضوح فوق‌العاده بهتر شود. همچنین مقالات مرتبط در زمینه موازی‌سازی در بهینه‌سازی تنک مورد مطالعه و بررسی قرار گرفت و کدهای مربوط به مقاله [۲۲] پیاده‌سازی شد.

کلیه عملیات بهبود عملکرد برنامه در این پایان‌نامه با دیکشنری و قطعات ثابت انجام شد، چرا که عملیات ساخت و تولید دیکشنری با اندازه‌های متفاوت بسیار زمان‌بر است. عملیات موازی‌سازی خصوصا در نوع دستورات ثابت با داده‌های مختلف، اغلب برای داده‌های حجیم کارایی خود را نشان می‌دهد. اما در این پایان نامه سعی بر این بوده تا تاثیر موازی‌سازی

[۲] for [۳] patch

با روش‌های مختلف روی داده‌های یکسان بررسی و تحلیل گردد. لذا در برنامه تصاویر وضوح فوق‌العاده، از یک دیکشنری با ابعاد ۱۴۴ در ۱۰۲۲ استفاده شده است.

۴-۲ موازی‌سازی سی‌پی‌یو برنامه تصاویر وضوح فوق‌العاده

روش‌های موازی‌سازی را می‌توان به دو بخش عمده تقسیم کرد. بخش اول موازی‌سازی روی سی‌پی‌یو است و بخش دیگر موازی‌سازی جی‌پی‌یو. در این بخش به فعالیت‌های انجام شده برای موازی‌سازی برنامه تصاویر وضوح فوق‌العاده روی سی‌پی‌یو (میزبان) پرداخته می‌شود.

۴-۲-۱ بهبود عملکرد با استفاده از بردارایزه کردن

برنامه تصاویر وضوح فوق‌العاده برای امکان بردارایزه شدن بررسی شد، در بخش اول تابع امپی، جایی که عملیات تقسیم برای نرمالیزه کردن دیکشنری A انجام می‌شود، به صورت زیر است:

```
for k=1:1:m                                                    ۱
   A(:,k)=A(:,k)/W(k);                                         ۲
 end                                                           ۳
```

این حلقه فور به صورت زیر بردارایزه شد:

```
A = bsxfun(@rdivide,A,W);
```

با استفاده از ابزار نمایه‌ساز متلب مشخص می‌شود که استفاده از تابع bsxfun موجب بهبود نسبی عملکرد برنامه می‌گردد. همان‌طور که در شکل ۴-۱ مشاهده می‌شود، خط ۷ برنامه که از این تابع در آن استفاده شده، تنها ۶.۲ درصد از زمان را به خود اختصاص داده، در حالی که در تصویر ۴-۲ ، ۱۵ درصد از زمان تابع امپی در خط ۹ برنامه که مربوط به حلقه فوری است که عملیات تقسیم در آن انجام شده‌است، زمان صرف شده‌است. همچنین برای بررسی تغییر در عملکرد کلی برنامه بهبود کیفیت تصویر، در هر حالت پنج بار برنامه اجرا شد و زمان میانگین برای هر دو حالت محاسبه شد که در جدول ۴-۱ ملاحظه می‌شود. بهبود عملکرد کلی حدود ۱.۵ ثانیه است که قطعا در صورت بزرگ‌تر شدن دیکشنری و

	زمان میانگین (ثانیه)
استفاده از bsxfun	۲۱.۲۱۷۹۸۹
استفاده از حلقه فور	۲۲.۶۷۷۲۳۷

جدول ۴-۱: مقایسه میانگین عملکرد کلی برنامه در حالت بردارایزه و غیر بردارایزه

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
20	`r=r-A(:,posZ)*A(:,posZ)'*r;`	142265	8.614 s	39.8%	
16	`Z=abs(A'*r);`	142265	5.678 s	26.2%	
5	`W=sum(A.*A).^.5;`	1764	2.103 s	9.7%	
7	`A = bsxfun(@rdivide,A,W);`	1764	1.350 s	6.2%	
18	`posZ=find(Z==max(Z),1);`	142265	1.156 s	5.3%	
All other lines			2.741 s	12.7%	
Totals			21.641 s	100%	

شکل ۴-۱: تحلیل تابع امپی در حالت استفاده از دستور bsxfun به جای حلقه فور

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
20	`r=r-A(:,posZ)*A(:,posZ)'*r;`	142265	8.338 s	36.0%	
16	`Z=abs(A'*r);`	142265	5.110 s	22.1%	
9	`A(:,k)=A(:,k)/W(k);`	1802808	3.474 s	15.0%	
5	`W=sum(A.*A).^.5;`	1764	2.329 s	10.1%	
18	`posZ=find(Z==max(Z),1);`	142265	1.062 s	4.6%	
All other lines			2.820 s	12.2%	
Totals			23.133 s	100%	

شکل ۴-۲: تحلیل تابع امپی در حالت عدم استفاده از دستور bsxfun به جای حلقه فور

قطعات تصویر، این بهبود تاثیر بیش‌تری خواهد داشت.

۴-۲-۲ استفاده از حلقه parfor برای بهبود سرعت برنامه

با توجه به این‌که برنامه امپی در درون دو حلقه فور تودرتو قرار گرفته، یکی از روش‌های معمول موازی‌سازی برای این حلقه‌ها استفاده از حلقه parfor است. در فصل قبل در مورد قوانین و محدودیت‌های استفاده از حلقه‌های parfor بحث شد و توضیح داده‌شد که تنها با تغییر کلمه for به parfor نمی‌توان از امکانات و مزایای این روش موازی‌سازی بهره برد.

در برنامه تصاویر وضوح فوق‌العاده (تصاویر با وضوح فوق‌العاده)، دو حلقه فور تودرتو وجود دارد. درون این دو حلقه ابتدا یک قطعه از تصویر با وضوح پایین جدا می‌شود، عملیات بهینه‌سازی تنک با استفاده از الگوریتم جستجوی تطابقی

انجام می‌شود. سپس با استفاده از دیکشنری با وضوح بالا، قطعه با وضوح بالا تولید شده و در پایان حلقه، این قطعه در سر جای خود در تصویر نهایی قرار می‌گیرد. این عملیات تا زمانی که همه قطعات تصویر پردازش شوند، ادامه می‌یابد.

همان‌طور که می‌دانیم، در استفاده از حلقه‌های parfor بهتر است خارجی‌ترین حلقه فور را تغییر دهیم. لذا حلقه خارجی در برنامه را به parfor تغییر می‌دهیم. در این لحظه کارگزار متلب شروع به جستجوی خطاها نموده و آن‌ها را گزارش می‌دهد. برای اجرای صحیح برنامه، باید خطاها برطرف شود. خطاهایی که کارگزار متلب در مورد برنامه تصاویر با وضوح بالا می‌دهد در تصویر ۳-۴ در فصل ۳ نشان داده شده است.

اولین خطا مربوط به متغیر count در حلقه است. این متغیر در قسمتی از برنامه که برای بهتر نشان دادن خروجی استفاده می‌شود به کار گرفته شده است.

```
count = count + 1;                                    ۱
if ~mod(count, 100)                                   ۲
   fprintf('.\n');                                    ۳
else                                                  ۴
   fprintf('.');                                      ۵
end                                                   ۶
```

بنابراین استفاده از آن لزومی ندارد و تاثیری در نتیجه خروجی نخواهد داشت. در نتیجه می‌توان از این قسمت به طور کلی صرف نظر کرد.

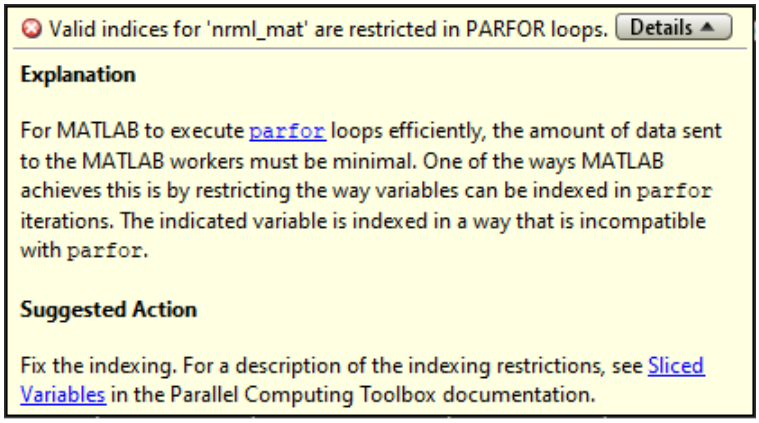
دو خطای بعدی مربوط به دو متغیر hIm و nrml_mat است که در انتهای حلقه استفاده شده‌اند:

برنامه ۴-۱: قسمت پایانی حلقه فور برنامه که خطا دارد

```
hIm(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+hpatch_size-1)...
        = hIm(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+hpatch_size-1)
          + hpatch;
nrml_mat(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+hpatch_size-1)...
        = nrml_mat(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+
          hpatch_size-1) + 1;
```

hIm در واقع همان تصویر با وضوح بالاست. درپایان حلقه قطعه وضوح بالا تولید شده در سر جای خود در این متغیر قرار می‌گیرد. متغیر nrml_mat یک ماتریس دقیقا هم اندازه hIm است که در هربار اضافه شدن یک قطعه به hIm ، محل پیکسل‌های آن قطعه را کنترل می‌کند. خطایی که کارگزار متلب برای این دو متغیر اعلام می‌کند، در تصویر ۴-۳ نشان داده شده است. این خطا بیان می‌دارد که برای افزایش عملکرد، میزان داده‌ای که به کارگرهای متلب ارسال می‌شود محدود شده است. یکی از این محدودیت‌ها مربوط به نحوه اندیس‌دهی متغیرهای درون حلقه است. همچنین برای بررسی بیش‌تر این خطا، مطالعه قوانین و محدودیت‌های مربوط به متغیرهای قطعه شده پیشنهاد شده است که در فصل ۳ بررسی شده است.

در بخش ۳-۴-۴-۴ و در مثال ۳-۴-۶ دیدیم که باید در اندیس‌دهی از مقادیر اسکالر استفاده شود و اندیس‌دهی به شکل hrowy:hrowy+hpatch_size-1 در داخل حلقه parfor قابل قبول نیست. همچنین در اندیس‌دهی از متغیرهای حلقه xx و yy به طور مستقیم استفاده نشده است. برای قرار دادن قطعات تولید شده در سر جایشان در تصویر ناچار به این



شکل ۴-۳: خطای اندیس‌دهی متغیر برش داده شده برای متغیر nrml_mat

نوع اندیس‌دهی در درون حلقه فور هستیم. از طرف دیگر همان‌طور که در تصویر ۴-۴ مشاهده می‌شود، خط ۱۵۵ برنامه مربوط به hIm، تنها ۰.۲ درصد یا ۰.۰۵۴ ثانیه از زمان برنامه را تلف می‌کند که زمان بسیار ناچیزی است و درنتیجه تنها راه باقیمانده خارج کردن این قسمت از برنامه از حلقه و قرار دادن آن در حلقه مجزاست.

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
119	`w = MP(D1,y,.01);`	1764	29.869 s	97.0%	
97	`fprintf('.');`	1747	0.280 s	0.9%	
101	`mmean = mean(mpatch(:));`	1764	0.164 s	0.5%	
134	`hpatch = Dh*w*mnorm;`	1734	0.148 s	0.5%	
155	`= hIm(hrowy:hrowy+hpatch_size-...`	1764	0.054 s	0.2%	
All other lines			0.263 s	0.9%	
Totals			30.779 s	100%	

شکل ۴-۴: زمان ناچیز قرار دادن قطعات در تصویر hIm

برای جدا کردن این بخش از برنامه، تمامی قطعات تولید شده در پایان حلقه داخلی در یک ماتریس چهاربعدی به نام Hpatch به صورت `Hpatch(:,:,:,yy) = hpatch` قرار داده می‌شود و پس از پایان حلقه داخلی، کلیه قطعات تولید شده در حلقه داخلی یعنی Hpatch در ماتریس دیگری به نام hPatch به صورت `hPatch(:,:,xx,:) = Hpatch`، جمع‌آوری می‌شود. در پایان متغیر hPatch ماتریسی حاوی تمامی قطعات تولید شده می‌باشد. اکنون می‌توان حلقه را بدون خطا به parfor تبدیل کرد. همچنین از hPatch در یک حلقه عادی فور برای قرار دادن قطعات در سرجای خودشان استفاده کرد. لازم به توضیح است که چون متغیر nrml_mat هیچ گونه وابستگی به عملیات درون حلقه ندارد و تنها در محل قطعه اضافه شده به hIm مقادیر ۱ قرار می‌دهد، می‌توان آن را به کلی از حلقه parfor حذف کرد.

برنامه ۴-۲: استفاده از متغیر hIm در حلقه تودرتو جداگانه

```
for xx = 1:length(mgridx)
    for yy = 1:length(mgridy)

        hcolx = hgridx(xx);
        hrowy = hgridy(yy);

        hIm(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+hpatch_size-1)...
            = hIm(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+hpatch_size
                -1) + hPatch(:,:,xx,yy);
        nrml_mat(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+hpatch_size
            -1)...
            = nrml_mat(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+
                hpatch_size-1) + 1;
    end
end
```

۴-۲-۲-۱ نتایج عملیات موازی‌سازی با استفاده از parfor

پس از اجرا شدن حلقه parfor بدون خطا، زمان اجرای برنامه بهبود قابل توجهی دارد به طوری که زمان اجرا نصف و حتی کم‌تر می‌گردد. نتایج اجرای برنامه با حلقه parfor و تابع امپی بعد از میانگین‌گیری ۵ بار اجرا در جدول ۴-۲ نشان داده شده‌است.

	زمان میانگین (ثانیه)
حالت عادی	۲۱.۲۱۷۹۸۹
استفاده از حلقه parfor	۱۰.۵۱۵۰۷۳

جدول ۴-۲: مقایسه میانگین عملکرد کلی برنامه در حالت عادی با حالت استفاده از حلقه parfor

۴-۳ موازی‌سازی جی‌پی‌یو برنامه تصاویر وضوح فوق‌العاده

یکی از روش‌های استفاده از قابلیت‌های GPU تعریف همه متغیرها از نوع gpuArray است. چرا که کلیه عملیات انجام شده بر روی نوع داده gpuArray به صورت خودکار روی GPU انجام می‌شود. اما این بدین معنا نیست که برنامه با سرعت بیش‌تری انجام شود. در تابع L1SR در مبحث استفاده از نمایش تنک در تصاویر با وضوح بالا بعد از تبدیل نوع داده تمامی متغیرها به نوع gpuArray، نه تنها بهبودی در سرعت و زمان اجرا حاصل نشد، بلکه مدت اجرا چندین برابر شد. دلیل این مسئله این است که تنها برخی از توابع از پیش تعریف شده متلب با ساختار جی‌پی‌یو سازگاری دارند و در صورتی که روی جی‌پی‌یو اجرا شوند، بهبود در سرعت حاصل خواهد شد. لیست این توابع در [۲۰] موجود است. برای آزمایش تصویر با

اندازه بزرگ‌تری به برنامه داده شد، اما باز هم شرایط تغییری نکرد. گرچه این احتمال وجود دارد که برنامه عملکرد بهتری برای دیکشنری با اندازه بزرگ‌تر و یا پچ‌های متفاوت داشته‌باشد چرا که می‌دانیم GPU برای محاسبات با حجم بالا مناسب است.

۴-۳-۱ استفاده از ابزار GPU Coder

از نسخه R2017b نرم‌افزار متلب، یک برنامه جدید به نام GPU Coder در قسمت APPS این برنامه فراهم شده که به صورت خودکار یک تابع متلب را به تابع کودا تبدیل می‌کند و در واقع تابع جدید تماما بر روی GPU اجرا می‌شود. اما نسخه R2017b متلب با نسخه ۱۴.۵ برنامه Visual Studio Community 2017 سازگار نیست و در همان مراحل ابتدایی برنامه GPU Coder خطایی در این زمینه دارد که البته با نصب آخرین نسخه کنونی متلب یعنی R2018a مشکل برطرف می‌شود.

سعی شد با استفاده از این ابزار، تابع ام‌پی و برخی توابع بهینه‌سازی دیگر تبدیل به توابع کودا شوند تا عملکرد برنامه بهبود یابد. نه تنها هیچ یک از توابع تولید شده منجر به بهتر شدن زمان برنامه نشد، بلکه حتی این زمان به مقدار زیادی افزایش یافت. دلیل این اتفاق می‌تواند بهینه نبودن کد باشد چرا که کدهای کودا در این روش به صورت خودکار و بوسیله ماشین تولید شده‌است. اما دلیل اصلی این است که عملیات انتقال داده‌ها از میزبان به GPU بسیار زمان‌بر است. توابع بهینه‌سازی تنک در برنامه تصاویر با وضوح فوق‌العاده از قبیل ام‌پی در درون حلقه‌های فور فراخوانی می‌شوند و بنابراین به دفعات اجرا می‌شوند. این امر بدین معناست که در هربار تکرار حلقه، باید یک عملیات انتقال داده به جی‌پی‌یو و همچنین بازگشت نتایج از جی‌پی‌یو به میزبان رخ دهد. این مسئله موجب غلبه سربار انتقال داده بر زمان اجرای خود برنامه می‌شود. در نتیجه استفاده از توابع کودا در درون حلقه‌های فور غیرمنطقی و ناکارآمد است. لذا حتی اگر کد به صورت بهینه و دستی نوشته شود، باز هم بدلیل فراخوانی آن در درون حلقه‌های فور، هیچ بهبودی حاصل نخواهد شد.

۴-۳-۲ چالش‌های نصب و راه‌اندازی کودا

ابتدا آخرین نسخه Visual Studio Community 2017 15.6.7 ، با افزونه Desktop development with C++ نصب شد و بعد آخرین نسخه CUDA Toolkit 9.1.85 نصب شد. برای شروع کدنویسی CUDA در Visual Studio یک پروژه جدید از نوع CUDA 9.1 ایجاد می‌کنیم. به صورت پیش‌فرض یک اسکریپت حاوی یک مثال ساده ایجاد می‌شود. برای اجرای برنامه‌های کودا ابتدا باید از منوی Build اولین گزینه Build Solution را انتخاب کنیم. اما در این مرحله با خطا مواجه می‌شوید که با جستجو در مورد این خطا متوجه خواهید شد که آخرین نسخه Visual Studio با آخرین نسخه CUDA Toolkit همخوانی ندارد و برای کارکرد صحیح باید نسخه ۱۴.۵ برنامه Visual Studio Community 2017 را نصب کنید. برای نصب این نسخه می‌توانید از ایـــنجا اقدام نمایید. پس از Build کردن کد که این‌بار با موفقیت انجام می‌شود، برای اجرای آن روی GPU باید Start CUDA Debugging را بزنید که گزینه دوم در منوی Nsight می‌باشد. برای دسترسی آسان‌تر به این گزینه می‌توانید با راست‌کلیک در فضای خالی نوارابزار در منوی باز شده گزینه

Nsight Connections را فعال کنید تا از این به بعد این گزینه را در نوار ابزار خود داشته‌باشید. پس از کلیک بر روی Start CUDA Debugging کد پیش‌فرض روی GPU اجرا می‌شود اما خروجی ندارد و برای این‌که مطمئن شویم کد روی GPU اجرا شده، می‌توانیم یک break point در میان کد مثلا در خط ۹ قراردهیم و بعد در منوی Nsight زیرمنوی Windows زیرمنوی CUDA Warp Watch یکی از گزینه‌های CUDA Warp Watch 1 را انتخاب می‌کنیم. این پنجره مشابه پنجره Watch در Visual Studio می‌باشد. تنها تفاوت آن این است که شما می‌توانید وضعیت متغیرها را در یک Warp ببینید. حالا اگر یکی از متغیرها، مثلا $a[i]$ را در قسمت Name پنجره CUDA Warp Watch 1 بنویسید، چیزی به شما نشان نمی‌دهد چرا که تا خط ۹ هنوز عملیاتی روی متغیرها انجام نشده، بنابراین اگر تا خط ۱۲ برنامه را اجرا کنید، مقادیر متغیر a را خواهید دید. اما وقتی در یک برنامه کودا break point قرار می‌دهیم، تنها محاسبات انجام شده روی یک thread خاص به ما نشان داده می‌شود. برای دیدن علیات روی سایر threadها از ابزاری به نام CUDA Debug Focus استفاده می‌نماییم که در همان مسیر منوی Nsight زیرمنوی Windows قرار دارد. در این پنجره شماره بلاک و threadای که در آن قرار داریم را نشان می‌دهد که البته در این مثال فقط یک بلاک داریم که حاوی ۵ thread است. که می‌توانیم شماره thread را از ۰۰۰ به ۱۰۰ تغییر دهیم تا بتوانیم عملیات انجام شده روی این thread را ببینیم.

۴-۴ پیاده‌سازی تابع جستجوی تطابقی با کتابخانه cuBLAS

در [۲۲] مسئله بازسازی سیگنال روی داده‌های تصادفی ابتدا روی میزبان و سپس پیاده‌سازی آن روی جی‌پی‌یو با استفاده از کودا انجام شده‌است. در پیاده‌سازی کودا از کتابخانه cuBLAS و در پیاده‌سازی میزبان از کتابخانه BLAS بهره برده شده‌است. پیاده‌سازی کودا بسیار سریع‌تر و در حدود ۳۱ برابر سرعت پیاده‌سازی BLAS است. مراحل اجرای برنامه به شرح زیر است:

- مقداردهی اولیه:
  - تعداد سطرها و ستون‌های دیکشنری تعیین می‌گردد،
  - تعداد مقادیر غیرصفر بردار تنک یا همان میزان تنکی بردار مشخص می‌شود.
  - مقدار خطای باقیمانده هدف مشخص می‌شود.
  - حداکثر تعداد تکرارها تعیین می‌شود.
- در برنامه اصلی
  - تعریف متغیرها
  - مقداردهی اولیه توابع تولید اعداد تصادفی و تابع زمان و همچنین تابع کوبلاس
  - تولید دیکشنری تصادفی روی میزبان

- تولید یک سیگنال تنک به صورت تصادفی
- تخصیص حافظه برای جواب در میزبان
- انتقال داده‌های لازم (دیکشنری و سیگنال) از میزبان به جی‌پی‌یو
- رمزگذاری : ضرب سیگنال تنک در دیکشنری و تولید سیگنال اصلی (جواب)
- رمزگشایی: بازتولید سیگنال تصادفی تولید شده با استفاده از جستجوی تطابقی
- محاسبه زمان سپری شده برای اجرا و همچنین مقایسه خروجی جستجوی تطابقی با سیگنال اصلی
- پاک کردن حافظه جی‌پی‌یو

کدهای مربوط به این پیاده‌سازی در ؟؟ قسمت آ-۵ موجود است.

در این پیاده‌سازی :

- از توابع `cublasSgemv` و `cublasDgemv` برای عملیات در ضرب و جمع ماتریس در بردار ممیز شناوری با دقت دو برابر استفاده شده‌است:

$$y = \alpha Ax + \beta \quad or \quad y = \alpha A^T x = \beta y$$

- از توابع `cublasIsamax` و `cublasIdamax` برای محاسبه کوچک‌ترین اندیس عضو با مقدار بیشینه بردار x استفاده شده‌است:

$$m = \arg\max |x_m|$$

- از توابع `cublasSaxpy` و `cublasDxpy` برای محاسبه جمع با دقت دوبرابری ممیز شناور استفاه شده‌است.

$$y = ax + y$$

- از توابع `cublasSnrm2` و `cublasDnrm2` برای محاسبه نُرم اُقلیدسی یک بردار ممیز شناور با دقت تکی استفاده شده‌است:

$$\| x \|_۲ = \sqrt{\sum_{m=۰}^{M} {x_m}^۲}$$

برای اجرای این برنامه در Visual Studio Community 2017 باید دستور زیر اضافه شود تا برنامه قابل اجرا باشد:

```
#pragma comment(lib,"cublas.lib")
```

در خروجی زمان لازم برای تخصیص داده، زمان انتقال داده از میزبان به جی‌پی‌یو، زمان برای رمزگذاری و زمان لازم برای رمزگشایی نشان داده شده‌است. نکته قابل توجه این است که زمان برای تخصیص حافظه که در میزبان انجام می‌شود، تقریبا با زمان اجرای عملیات جستجوی تطابقی که روی جی‌پی‌یو اجرا می‌گردد برابر است. و همچنین زمان عملیات رمزگذاری صفر است.

۴-۵ جمع‌بندی

در این پایان‌نامه ابتدا به مبحث بهینه‌سازی تُنک و الگوریتم‌های جستجو برای حل آن‌ها پرداختیم. در مورد مفاهیم پایه موازی‌سازی بحث نمودیم. مقدماتی از برنامه‌نویسی کودا را بیان کردیم. امکانات و روش‌های موازی‌سازی در متلب را بررسی نمودیم و در پایان سعی کردیم با پیاده‌سازی عملی روش‌ها و ابزارهای موازی‌سازی عملیات بهینه‌سازی تُنک و جستجوی تطابقی را در برنامه تصاویر با وضوح فوق‌العاده پیاده‌سازی نماییم. همچنین الگوریتم جستجوی تطابقی را با استفاده از توابع کتابخانه جبر خطی کودا روی داده‌های تصادفی پیاده‌سازی کردیم.

۴-۶ تحقیقات در آینده

با توجه به این‌که کودا پتانسیل قابل توجهی در بهبود سرعت برنامه‌ها دارد، می‌توان برنامه تصاویر با وضوح بالا را با استفاده از آن بازنویسی کرد تا نتایج بهتری حاصل گردد. همچنین سایر امکانات و قابلیت‌های کودا بررسی شود. از کودا برای سایر کاربردها در پردازش تصویر و دیگر مباحث استفاده شود. بهبود عملکرد روش‌های موازی‌سازی جی‌پی‌یو با دیکشنری‌های متفاوت (از نظر اندازه) و همچنین به کارگیری موازی‌سازی در تولید دیکشنری از دیگر فعالیت‌هایی است که می‌توان در این راستا انجام داد.

فهرست منابع

[1] Averbuch, A., Coifman, R., Donoho, D.L., Elad, M., and Israeli, M. Fast and accurate polar fourier transform. *Journal on Applied and Computational Harmonic Analysis*, 21:145–167, September 2006.

[2] Ahmed, N., Natarajan, T., and Rao, K. R. Discrete cosine transform. *IEEE Transactions on Computers*, C-23(1):90–93, Jan 1974.

[3] Ram, Idan, Elad, Michael, and Cohen, Israel. Generalized tree-based wavelet transform. *CoRR*, abs/1011.4615, 2010.

[4] Jolliffe, I.T. *Principal Component Analysis*. Springer Verlag, 1986.

[5] Elad, Michael. *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer Publishing Company, Incorporated, 1st ed. , 2010.

[6] Koskinas, S. *Denoising Using Randomized Matching Pursuit: Algorithmic Improvements and GPU Implementation*. McGill theses. McGill University Libraries, 2013.

[7] Zhang, Zheng, Xu, Yong, Yang, Jian, Li, Xuelong, and Zhang, David. A survey of sparse representation: algorithms and applications. *CoRR*, abs/1602.07017, 2016.

[8] Jain, A. K. Advances in mathematical models for image processing. *Proceedings of the IEEE*, 69(5):502–528, May 1981.

[9] Raid, A. M., Khedr, W. M., El-dosuky, M. A., and Ahmed, Wesam. Jpeg image compression using discrete cosine transform - A survey. *CoRR*, abs/1405.6147, 2014.

[10] Sulam, J. and Elad, M. Large inpainting of face images with trainlets. *IEEE Signal Processing Letters*, 23(12):1839–1843, Dec 2016.

[11] Papyan, Vardan and Elad, Michael. Multi-scale patch-based image restoration - super resolution. `https://www.codeocean.com/`, January 2017.

[12] Ren, Yi, Romano, Yaniv, and Elad, Michael. Example-based image synthesis via randomized patch-matching. *CoRR*, abs/1609.07370, 2016.

[13] Shi, Yunhui, Qi, Na, Yin, Baocai, and Ding, Wenpeng. Two dimensional k-svd for the analysis sparse dictionary. in Lin, Weisi, Xu, Dong, Ho, Anthony, Wu, Jianxin, He, Ying, Cai, Jianfei, Kankanhalli, Mohan, and Sun, Ming-Ting, eds. , *Advances in Multimedia Information Processing – PCM 2012*, pp. 861–871, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[14] Nasrollahi, Kamal and Moeslund, Thomas B. Super-resolution: A comprehensive survey. *Mach. Vision Appl.*, 25(6):1423–1468, August 2014.

[15] Mallat, Stéphane G and Zhang, Zhifeng. Matching pursuits with time-frequency dictionaries. *Signal Processing, IEEE Transactions on*, 41(12):3397–3415, 1993.

[۱۶] حامدی، راضیه. بررسی نرم‌های مختلف در نمایش تنک. پایان‌نامه کارشناسی‌ارشد، دانشگاه حکیم سبزواری، ۱۳۹۵.

[17] Kirk, David B. and Hwu, Wen-mei W. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st ed. , 2010.

[18] Ploskas, Nikolaos and Samaras, Nikolaos. *GPU Programming in MATLAB*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st ed. , 2016.

[19] NVIDIA. *CUDA Toolkit Documentation v9.2.148*, 2018.

[20] MATLAB. *Matlab version 9.4.0.813654 (R2018a) Documentation*. Natick, Massachusetts, 2018.

[21] Yang, Jianchao, Wright, J., Huang, T., and Ma, Yi. Image super-resolution as sparse representation of raw image patches. in *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, June 2008.

[22] Andrecut, M. Fast gpu implementation of sparse signal recovery from random projections. *Engineering Letters*, abs/1202.4347, 2009.

پیوست آ

برنامه‌های استفاده شده در این پایان‌نامه

آ-۱ برنامه‌های متلب

در این بخش برنامه‌های متلب استفاده شده و نوشته شده برای این پایان‌نامه آورده شده است.

آ-۱-۱ برنامه جستجوی تطابقی

برنامه آ-۱: جستجو تطابقی

```
function xMP = MP(A,b,thrMP)
%% Normalization
%A = gpuArray(A);
m=size(A,2);
W=sum(A.*A).^.5;
%W=gpuArray(sum(A.*A).^.5);
% A = bsxfun(@rdivide,A,W);
for k=1:1:m
A(:,k)=A(:,k)/W(k);
end
%%
%A = gather(A);
r=b;
        xMP=zeros(m,1);
        while r'*r>thrMP  %ta zamani k norm baqimande bozorgtar az
            thr ast
            Z=abs(A'*r);
            %Z=abs(mexMatrixMultiplication(A',r));
            posZ=find(Z==max(Z),1);
            xMP(posZ)=xMP(posZ)+A(:,posZ)'*r;
            r=r-A(:,posZ)*A(:,posZ)'*r;
        end
```

```
end
```

آ-۱-۲ برنامه‌های مربوط به تصاویر با وضوح فوق‌العاده [۲۱]

آ-۱-۲-۱ برنامه اصلی تصاویر با وضوح فوق‌العاده

برنامه آ-۲: برنامه اصلی

```
% Image super-resolution using sparse representation
% Example code
%
% Nov. 2, 2007. Jianchao Yang
% IFP @ UIUC
%
% Revised version. April, 2009.
%
% Reference
% Jianchao Yang, John Wright, Thomas Huang and Yi Ma. Image
    superresolution
% via sparse representation of raw image patches. IEEE Computer
    Society
% Conference on Computer Vision and Pattern Recognition (CVPR),
    2008.
%
% For any questions, email me by jyang29@illinois.edu

clear all;
clc;

addpath('Solver');
addpath('Sparse coding');

% ========================================================
% specify the parameter settings

patch_size = 3; % patch size for the low resolution input image
overlap = 1; % overlap between adjacent patches
lambda = 0.1; % sparsity parameter
zooming = 3; % zooming factor, if you change this, the dictionary
    needs to be retrained.

tr_dir = 'Data/training'; % path for training images
skip_smp_training = true; % sample training patches
skip_dictionary_training = true; % train the coupled dictionary
num_patch = 50000; % number of patches to sample as the dictionary
codebook_size = 1024; % size of the dictionary

regres = 'L1'; % 'L1' or 'L2', use the sparse representation
    directly, or use the supports for L2 regression
% ========================================================
```

```
% training coupled dictionaries for super-resolution

if ~skip_smp_training,
    disp('Sampling image patches...');
    [Xh, Xl] = rnd_smp_dictionary(tr_dir, patch_size, zooming,
       num_patch);
    save('Data/Dictionary/smp_patches.mat', 'Xh', 'Xl');
    skip_dictionary_training = false;
end;

if ~skip_dictionary_training,
    load('Data/Dictionary/smp_patches.mat');
    [Dh, Dl] = coupled_dic_train(Xh, Xl, codebook_size, lambda);
    save('Data/Dictionary/Dictionary.mat', 'Dh', 'Dl');
else
    load('Data/Dictionary/Dictionary.mat');
end;
% ===================================================================
% Process the test image

fname = 'Data/Test/1.bmp';
testIm = imread(fname); % testIm is a high resolution image, we
   downsample it and do super-resolution

if rem(size(testIm,1),zooming) ~=0,
    nrow = floor(size(testIm,1)/zooming)*zooming;
    testIm = testIm(1:nrow,:,:);
end;
if rem(size(testIm,2),zooming) ~=0,
    ncol = floor(size(testIm,2)/zooming)*zooming;
    testIm = testIm(:,1:ncol,:);
end;

imwrite(testIm, 'Data/Test/high.bmp', 'BMP');

lowIm = imresize(testIm,1/zooming, 'bicubic');
imwrite(lowIm,'Data/Test/low.bmp','BMP');

interpIm = imresize(lowIm,zooming,'bicubic');
imwrite(uint8(interpIm),'Data/Test/bb.bmp','BMP');

% work with the illuminance domain only
lowIm2 = rgb2ycbcr(lowIm);
lImy = double(lowIm2(:,:,1));

% bicubic interpolation for the other two channels
interpIm2 = rgb2ycbcr(interpIm);
hImcb = interpIm2(:,:,2);
hImcr = interpIm2(:,:,3);

% ===================================================================
% Super-resolution using sparse representation

disp('Start superresolution...');

%Dl = gpuArray(Dl);
```

```
%[hImy] = L1SR(lImy, zooming, patch_size, overlap, Dh, Dl, lambda,
    regres) ;
%[hImy] = L1SR2(lImy, zooming, patch_size, overlap, Dh, Dl, lambda,
    regres) ;
[hImy] = L1SR_parfor(lImy, zooming, patch_size, overlap, Dh, Dl,
    lambda, regres) ;
%[hImy] = L1SR_functionalized(lImy, zooming, patch_size, overlap, Dh
    , Dl, lambda) ;



ReconIm(:,:,1) = uint8(hImy);
ReconIm(:,:,2) = hImcb;
ReconIm(:,:,3) = hImcr;

nnIm = imresize(lowIm, zooming, 'nearest');
figure, imshow(nnIm);
title('Input image');
pause(1);
figure, imshow(interpIm);
title('Bicubic interpolation');
pause(1)

ReconIm = ycbcr2rgb(ReconIm);
figure,imshow(ReconIm,[]);
title('Our method');
imwrite(uint8(ReconIm),'Data/Test/BNMZGT01.bmp','BMP');

% compute PSNR for the illuminance channel
bb_rmse = compute_rmse(testIm, nnIm);
sp_rmse = compute_rmse(testIm, ReconIm);

bb_psnr = 20*log10(255/bb_rmse);
sp_psnr = 20*log10(255/sp_rmse);

fprintf('PSNR for Bicubic Interpolation: %f dB\n', bb_psnr);
fprintf('PSNR for Sparse Representation Recovery: %f dB\n', sp_psnr)
    ;
```

آ-۱-۳ برنامه L1SR

این برنامه در درون برنامه اصلی فراخوانی شده و شامل حلقه تودرتو که عملیات بهینه‌سازی تنک روی قطعات تصویر را انجام می‌دهد. است.

برنامه آ-۳: برنامه L1SR

```
function [hIm] = L1SR(lIm, zooming, patch_size, overlap, Dh, Dl,
    lambda, regres)
% Use sparse representation as the prior for image super-resolution
```

```
% Usage
%       [hIm] = L1SR(lIm, zooming, patch_size, overlap, Dh, Dl,
    lambda)
%
% Inputs
%   -lIm:           low resolution input image, single channel, e.g.
%   illuminance
%   -zooming:       zooming factor, e.g. 3
%   -patch_size:    patch size for the low resolution image
%   -overlap:       overlap among patches, e.g. 1
%   -Dh:            dictionary for the high resolution patches
%   -Dl:            dictionary for the low resolution patches
%   -regres:       'L1' use the sparse representation directly to
    high
%                   resolution dictionary;
%                   'L2' use the supports found by sparse
    representation
%                   and apply least square regression coefficients
    to high
%                   resolution dictionary.
% Ouputs
%   -hIm:           the recovered image, single channel
%
% Written by Jianchao Yang @ IFP UIUC
% April, 2009
% Webpage: http://www.ifp.illinois.edu/~jyang29/
% For any questions, please email me by jyang29@uiuc.edu
%
% Reference
% Jianchao Yang, John Wright, Thomas Huang and Yi Ma. Image
    superresolution
% as sparse representation of raw image patches. IEEE Computer
    Society
% Conference on Computer Vision and Pattern Recognition (CVPR),
    2008.
%

%k = parallel.gpu.CUDAKernel('matrixMultiplication.ptx', '
    matrixMultiplication.cu');
% k.ThreadBlockSize = [144 1 1];
% k.GridSize = [10, 10];
% C = zeros(1022,1);


[lhg, lwd] = size(lIm);
hhg = lhg*zooming;
hwd = lwd*zooming;

mIm = imresize(lIm, 2,'bicubic');
[mhg, mwd] = size(mIm);
hpatch_size = patch_size*zooming;
mpatch_size = patch_size*2;

% extract gradient feature from lIm
hf1 = [-1,0,1];
vf1 = [-1,0,1]';
```

```
hf2 = [1,0,-2,0,1];
vf2 = [1,0,-2,0,1]';

lImG11 = conv2(mIm,hf1,'same');
lImG12 = conv2(mIm,vf1,'same');
lImG21 = conv2(mIm,hf2,'same');
lImG22 = conv2(mIm,vf2,'same');


lImfea(:,:,1) = lImG11;
lImfea(:,:,2) = lImG12;
lImfea(:,:,3) = lImG21;
lImfea(:,:,4) = lImG22;

lgridx = 2:patch_size-overlap:lwd-patch_size;
lgridx = [lgridx, lwd-patch_size];
lgridy = 2:patch_size-overlap:lhg-patch_size;
lgridy = [lgridy, lhg-patch_size];

mgridx = (lgridx - 1)*2 + 1;
mgridy = (lgridy - 1)*2 + 1;

% using linear programming to find sparse solution
bhIm = imresize(lIm, 3, 'bicubic');
%hIm = zeros([hhg, hwd],'gpuArray');
hIm = zeros([hhg, hwd]);
nrml_mat = zeros([hhg, hwd]);

hgridx = (lgridx-1)*zooming + 1;
hgridy = (lgridy-1)*zooming + 1;

disp('Processing the patches sequentially...');
count = 0;

% loop to recover each patch
tic;
for xx = 1:length(mgridx)
    for yy = 1:length(mgridy)

        mcolx = mgridx(xx);
        mrowy = mgridy(yy);

        count = count + 1;
        if ~mod(count, 100)
            fprintf('.\n');
        else
            fprintf('.');
        end

        mpatch = mIm(mrowy:mrowy+mpatch_size-1, mcolx:mcolx+
           mpatch_size-1);
        mmean = mean(mpatch(:));

        mpatchfea = lImfea(mrowy:mrowy+mpatch_size-1, mcolx:mcolx+
           mpatch_size-1, :);
        mpatchfea = mpatchfea(:);
```

```
mnorm = sqrt(sum(mpatchfea.^2));

if mnorm > 1
    y = mpatchfea./mnorm;
else
    y = mpatchfea;
end


% w = SolveLasso(Dl, y, size(Dl, 2), 'nnlasso', [], lambda);
% w = feature_sign(Dl, y, lambda);
% w = OMP01(Dl,y,.0001);
% w = MP_CUDA_MEX(Dl',y);
 w = MP(Dl,y,.01);
% w = MP_CUDA(Dl,y,.01,k,C);
% w = MP_gpuArrA(Dl,y,.01);
% w = MP_mex(Dl,y,.01);
% w = DAS(Dl,y,.9);
% w = norm1MZ(Dl, y, 1e-4);
% w=OMP01(Dl,y,.01);
% w=LSMP(Dl,y,.01);

if isempty(w)
    w = zeros(size(Dl, 2), 1);
end
switch regres
    case 'L1'
        if mnorm > 1
            hpatch = Dh*w*mnorm;
        else
            hpatch = Dh*w;
        end
    case 'L2'
        idx = find(w);
        lsups = Dl(:, idx);
        hsups = Dh(:, idx);
        w = inv(lsups'*lsups)*lsups'*mpatchfea;
        hpatch = hsups*w;
    otherwise
        error('Unknown fitting!');
end

hpatch = reshape(hpatch, [hpatch_size, hpatch_size]);
hpatch = hpatch + mmean;

hcolx = hgridx(xx);
hrowy = hgridy(yy);

hIm(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+hpatch_size-1)...
    = hIm(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+hpatch_size
       -1) + hpatch;
nrml_mat(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+hpatch_size
   -1)...
    = nrml_mat(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+
       hpatch_size-1) + 1;
```

```
    end
end
toc;
fprintf('done!\n');

% fill the empty
hIm(1:3, :) = bhIm(1:3, :);
hIm(:, 1:3) = bhIm(:, 1:3);

hIm(end-2:end, :) = bhIm(end-2:end, :);
hIm(:, end-2:end) = bhIm(:, end-2:end);

nrml_mat(nrml_mat < 1) = 1;
hIm = hIm./nrml_mat;
% hIm = uint8(hIm);
```

آ-۱-۴ برنامه L1SR_parfor

این برنامه نسخه موازی شده برنامه L1SR است که در بخش ۴-۲-۲ توضیحات کامل در مورد آن ارائه شده‌است.

برنامه آ-۴: قسمت موازی شده برنامه L1SR

```
function xMP = MP(A,b,thrMP)
function [hIm] = L1SR_parfor(lIm, zooming, patch_size, overlap, Dh,
    Dl, lambda, regres)

parfor xx = 1:length(mgridx)
    Hpatch = zeros(hpatch_size,hpatch_size,1,length(mgridy));
    for yy = 1:length(mgridy)

        mcolx = mgridx(xx);
        mrowy = mgridy(yy);

%         count = count + 1;
%         if ~mod(count, 100)
%           fprintf('.\n');
%         else
%             fprintf('.');
%         end

        mpatch = mIm(mrowy:mrowy+mpatch_size-1, mcolx:mcolx+
            mpatch_size-1);
        mmean = mean(mpatch(:));

        mpatchfea = lImfea(mrowy:mrowy+mpatch_size-1, mcolx:mcolx+
            mpatch_size-1, :);
        mpatchfea = mpatchfea(:);

        mnorm = sqrt(sum(mpatchfea.^2));

        if mnorm > 1
            y = mpatchfea./mnorm;
```

```
        else
            y = mpatchfea;
        end


        % w = SolveLasso(Dl, y, size(Dl, 2), 'nnlasso', [], lambda);
        % w = feature_sign(Dl, y, lambda);
        % w = OMP01(Dl,y,.0001);
        w = MP(Dl,y,.01);
        % w = MP_gpuArrA(Dl,y,.01);
%           w = MP_mex(Dl,y,.01);
        % w = DAS(Dl,y,.9);
        % w = norm1MZ(Dl, y, 1e-4);
        % w=OMP01(Dl,y,.01);
        % w=LSMP(Dl,y,.01);

        if isempty(w)
            w = zeros(size(Dl, 2), 1);
        end
%         switch regres
%             case 'L1'
                if mnorm > 1
                    hpatch = Dh*w*mnorm;
                else
                    hpatch = Dh*w;
                end
%             case 'L2'
%                 idx = find(w);
%                 lsups = Dl(:, idx);
%                 hsups = Dh(:, idx);
%                 w = inv(lsups'*lsups)*lsups'*mpatchfea;
%                 hpatch = hsups*w;
%             otherwise
%                 error('Unknown fitting!');
%         end

        hpatch = reshape(hpatch, [hpatch_size, hpatch_size]);
        hpatch = hpatch + mmean;

           Hpatch(:,:,:,yy) = hpatch;

    end
    hPatch(:,:,xx,:) = Hpatch;
end
%[hIm,nrml_mat] = HimageAssemble_mex(hPatch,hpatch_size,hgridx,
    hgridy,mgridx,mgridy,hhg,hwd) ;
for xx = 1:length(mgridx)
    for yy = 1:length(mgridy)

        hcolx = hgridx(xx);
        hrowy = hgridy(yy);

        hIm(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+hpatch_size-1)...
            = hIm(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+hpatch_size
                -1) + hPatch(:,:,xx,yy);
        nrml_mat(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+hpatch_size
```

```
            -1)...
             = nrml_mat(hrowy:hrowy+hpatch_size-1, hcolx:hcolx+
                 hpatch_size-1) + 1;

    end
end
```

آ-۲ پیاده سازی جستجوی تطابقی با استفاده از کتابخانه cuBlas

برنامه آ-۵: پیاده‌سازی جستجوی تطابقی با استفاده از کتابخانه cuBlas

```
/*                                                                  ۱
*  Double precision CUBLAS (NVIDIA) implementation                  ۲
*  of Matching Pursuit algorithm for                                ۳
*  sparse signal recovery from random projections.                  ۴
*/                                                                  ۵
/* Includes, system */                                              ۶
                                                                    ۷
#include <stdio.h>                                                  ۸
#include <stdlib.h>                                                 ۹
#include <string.h>                                                 ۱۰
#include <time.h>                                                   ۱۱
                                                                    ۱۲
/* Includes, cuda */                                               ۱۳
#include <cublas.h>                                                 ۱۴
#pragma comment(lib,"cublas.lib")                                   ۱۵
                                                                    ۱۶
/* Number of columns and rows in dictionary */                      ۱۷
#define M (2000)                                                    ۱۸
#define N ((int)(M/2))                                             ۱۹
                                                                    ۲۰
/* Number of non-zero elements in sparse signal */                  ۲۱
int K = 0.07*M;                                                     ۲۲
                                                                    ۲۳
/* Residual error */                                               ۲۴
double epsilon = 1.0e-7;                                            ۲۵
                                                                    ۲۶
/* Maximum number of iterations */                                  ۲۷
int T = N;                                                          ۲۸
                                                                    ۲۹
/* Sign function */                                                ۳۰
double sign(double x) { return (x >= 0) - (x<0); }                  ۳۱
                                                                    ۳۲
/* Matrix indexing convention */                                   ۳۳
#define id(m, n, ld) (((n) * (ld) + (m)))                          ۳۴
                                                                    ۳۵
                                                                    ۳۶
int main(int argc, char** argv)                                     ۳۷
{                                                                   ۳۸
        cublasStatus status;                                        ۳۹
```

```
double *h_D, *h_X, *h_C, *c;
double *d_D = 0, *d_S = 0, *d_R = 0;
int MN = M*N, m, n, k, q, t;
double norm = sqrt(N), normi, normf, a, dtime;
printf("\nProblem dimensions : NxM = %dx%d, K = %d", N, M, K
    );

/* Initialize srand and clock */
srand(time(NULL));
clock_t start = clock();

/* Initialize cublas */
status = cublasInit();
if (status != CUBLAS_STATUS_SUCCESS)
{
        fprintf(stderr, "!CUBLAS initialization error\n");
        return EXIT_FAILURE;
}

/* Initialize dictionary on host */
h_D = (double*)malloc(MN * sizeof(h_D[0]));
if (h_D == 0)
{
        fprintf(stderr, "!host memory allocation error(
            dictionary)\n");
        return EXIT_FAILURE;
}
for (n = 0; n < N; n++)
{
        for (m = 0; m < M; m++)
        {
                a = sign(2.0*rand() / (double)RAND_MAX -
                    1.0) / norm;
                h_D[id(m, n, M)] = a;
        }
}

/* Create a random K-sparse signal */
h_X = (double*)calloc(M, sizeof(h_X[0]));
if (h_X == 0)
{
        fprintf(stderr, "!host memory allocation error(
            signal)\n");
        return EXIT_FAILURE;
}
for (k = 0; k<K; k++)
{
        a = 2.0*rand() / (double)RAND_MAX - 1.0;
        h_X[rand() % M] = a;
}

/* Allocate solution memory on host */
h_C = (double*)calloc(M, sizeof(h_C[0]));
if (h_C == 0)
{
```

```
        fprintf(stderr, "!host memory allocation error(
            solution)\n");
        return EXIT_FAILURE;
}
c = (double*)calloc(1, sizeof(c));
if (c == 0)
{
        fprintf(stderr, "!host memory allocation error(c)\n
            );
        return EXIT_FAILURE;
}
dtime = ((double)clock() - start) / CLOCKS_PER_SEC;
printf("\nTime for host data allocation : %f", dtime);
start = clock();

/* Host to device data transfer: dictionary */
status = cublasAlloc(MN, sizeof(d_D[0]),
        (void**)&d_D);
if (status != CUBLAS_STATUS_SUCCESS)
{
        fprintf(stderr, "!device memory allocation error(
            dictionary)\n");
        return EXIT_FAILURE;
}
status = cublasSetVector(MN, sizeof(h_D[0]),
        h_D, 1, d_D, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
        fprintf(stderr, "!device access error(write
            dictionary)\n");
        return EXIT_FAILURE;
}

/* Host to device data transfer: signal */
status = cublasAlloc(M, sizeof(d_R[0]),
        (void**)&d_R);
if (status != CUBLAS_STATUS_SUCCESS)
{
        fprintf(stderr, "!device memory allocation error(
            signal)\n");
        return EXIT_FAILURE;
}
status = cublasSetVector(M, sizeof(h_X[0]),
        h_X, 1, d_R, 1);
if (status != CUBLAS_STATUS_SUCCESS)
{
        fprintf(stderr, "!device access error(write signal)
            n");
        return EXIT_FAILURE;
}
status = cublasAlloc(N, sizeof(d_S[0]),(void**)&d_S);
if (status != CUBLAS_STATUS_SUCCESS)
{
        fprintf(stderr, "!device memory allocation error(
            projected vector)\n");
        return EXIT_FAILURE;
```

```
}
dtime = ((double)clock() - start) / CLOCKS_PER_SEC;
printf("\nTime for Host to Device data transfer : %f(s)",
    dtime);

/* Encoding the signal on device*/
start = clock();
cublasDgemv('t', M, N, 1.0, d_D, M, d_R, 1, 0.0, d_S, 1);
status = cublasGetError();
if (status != CUBLAS_STATUS_SUCCESS)
{
        fprintf(stderr, "!kernel execution error(encoding)\n
            ");
        return EXIT_FAILURE;
}
dtime = ((double)clock() - start) / CLOCKS_PER_SEC;
printf("\nTime for encoding: %f(s)", dtime);

/* Decoding the signal on device*/
start = clock();
normi = cublasDnrm2(N, d_S, 1);
epsilon = sqrt(epsilon*normi);
normf = normi;
t = 0;
while (normf > epsilon && t < T)
{
        cublasDgemv('n', M, N, 1.0, d_D, M, d_S, 1, 0.0, d_R
            , 1);
        q = cublasIdamax(M, d_R, 1) - 1;
        cublasGetVector(1, sizeof(c), &d_R[q], 1, c, 1);
        h_C[q] = *c + h_C[q];
        cublasDaxpy(N, -(*c), &d_D[q], M, d_S, 1);
        normf = cublasDnrm2(N, d_S, 1);
        t++;
}
status = cublasGetError();
if (status != CUBLAS_STATUS_SUCCESS)
{
        fprintf(stderr, "!kernel execution error(decoding)\n
            ");
        return EXIT_FAILURE;
}
dtime = ((double)clock() - start) / CLOCKS_PER_SEC;
printf("\nTime for decoding: %f(s)", dtime);
a = 100.0*(normf*normf) / (normi*normi);
printf("\nComputation residual error : %f", a);

/* Check the solution */
printf("\nSolution(first column), Reference(second column) \n
    ");
getchar();
for (m = 0; m<M; m++)
{
        printf("\n%f\t%f", h_C[m], h_X[m]);
}
normi = 0; normf = 0;
```

```
        for (m = 0; m<M; m++)
        {
                normi = normi + h_X[m] * h_X[m];
                normf = normf +
                        (h_C[m] - h_X[m])*(h_C[m] - h_X[m]);
        }
        printf("\nSolution residual error :%f", 100.0*normf / normi)
            ;
        /* Memory clean up */
        free(h_D);
        free(h_X);
        free(h_C);
        status = cublasFree(d_D);
        status = cublasFree(d_S);
        status = cublasFree(d_R);
        if (status != CUBLAS_STATUS_SUCCESS)
        {
                fprintf(stderr,"!device memory free error\n");
                return EXIT_FAILURE;
        }
        /* Shutdown */
        status = cublasShutdown();
        if (status != CUBLAS_STATUS_SUCCESS)
        {
                fprintf(stderr,"!cublas shutdown error\n");
                return EXIT_FAILURE;
        }
        if (argc <= 1 || strcmp(argv[1], " - noprompt"))
        {
                printf("\nPress ENTER to exit...\n");
                getchar();
        }
        return EXIT_SUCCESS;
}
```

Hakim Sabzevari University

An Outline of MSc. Thesis

Surname: Khoshnevis	Name: Babak	Student No.: 9513137037

Supervisor: Dr. Mahmood Amintoosi

Advisor: Dr. Mehdi Zaferanieh

Faculty of Mathematics and Computer Science

Program: Decision Science and Knowledge Engineering

Title of thesis: Parallel Computing in Sparse Optimization

Keywords: Sparse Optimization, Parallel Computing, GPU Computing, CUDA, Super-Resolution

Abstract: Nowadays, Sparse Optimization is being used widely in most of the data modeling problems as a novel and efficient method. Mostly, solving these problems especially for large data increases computational complexity which ends in lack of performance. In such situations, Parallel Computing is an inevitable solution.

In this thesis, different Parallel Computing methods is analyzed in order to improve performance of Sparse Representation problems. First, basics of Sparse Representation is described. Then, Matching Pursuit and Orthogonal Matching Pursuit algorithms are described in detail. In parallel sections, Architecture of CPUs and GPUs are compared and therefore Heterogeneous Parallel Computing is defined as a modern method in Parallel Computing. Afterwards, basics of CUDA programming which is one of the most efficient programming platforms for Heterogeneous Parallel Programming are expressed.

In the end, most of the Parallel Computing methods and tools such as Parfor Loops, GPU Arrays, CUDA Kernels and CUDA MEX Functions are tried to be implemented and analysed in practice (in MATLAB). In one case, using Parfor in a Super-Resolution program has resulted in 50 percent of improvement in performance. Also, a Matching Pursuit CUDA code on random data is debugged and run in Visual Studio and then is tried to be used in the Super-Resolution program as a MEX function.

Hakim Sabzevari University
Faculty of Mathematics and Computer Science

A Thesis Submitted in Partial Fulfilment of the Requirement for the Degree of Master of Science in Decision Science and Knowledge Engineering

Parallel Computing in Sparse Optimization

Supervisor:
Dr. Mahmood Amintoosi

Advisor:
Dr. Mehdi Zaferanieh

By:
Babak Khoshnevis

September 2018