پیاده‌سازی یک هسته‌ی قابل تنظیم ضرب ماتریس‌های تنک در بردار به کمک زبان آزاد محاسباتی روی پردازنده‌های گرافیکی

فرشید مسیبی

گروه مهندسی عمران، دانشکده فنی و مهندسی، دانشگاه اصفهان

(دریافت مقاله: ۱۳۹۱/۰۲/۲۴– دریافت نسخه نهایی: ۱۳۹۱/۱۰/۰۹)

چکیده – ضرب ماتریس‌هاي تنک در بردار اصلي‌ترین پایه روش‌هاي تکراري در حل دستگاه‌هاي معادلات خطي به شمار مي‌رود. تقریباً تمامي روش‌هاي عددي نیاز به حل چنین دستگاهي از معادلات خطي در فرایند حل خود دارند. تا کنون تحقیقات بسیار زیادي در این زمینه انجام گرفته و هنوز نیز در حال انجام است. یکي از مناسب‌ترین روش‌ها براي انجام سریع‌تر این گونه عملیات استفاده از پردازنده‌هاي گرافیکي است. این پردازنده‌ها در سال‌هاي اخیر رشد بسیار چشمگیري از نظر قدرت پردازش داشته‌اند. در این تحقیق روشي نو براي انجام این عملیات روي پردازنده‌هاي گرافیکي با کمک زبان آزاد محاسباتي ارائه شده است. نتایج نشان مي‌دهد با بهینه‌سازي پارامترهاي این روش مي‌توان کارایي به مراتب بالاتر از انجام این عمل روي پردازنده‌ها حتي با کمک استاندارد باز چندپردازنده‌اي به دست آورد. همچنین نتایج نشان مي‌دهد در نزدیکي پارامترهاي بهینه این روش حساسیت چنداني وجود ندارد که کار را براي تخمین آن‌ها از روي مشخصات ماتریس بسیار ساده‌تر مي‌کند.

واژگان کلیدی : **ضرب ماتریس‌هاي تنک در بردار، پردازنده‌هاي گرافیکي، زبان آزاد محاسباتي**

Implementation of a tunable sparse matrix-vector product kernel using OpenCL on graphics processing units

F. Mossaiby

1. Department of Civil Engineering, Faculty of Engineering, University of Isfahan

Abstract: ***Sparse matrix-vector multiplication (SpMV) is the key operation in the iterative methods for solving linear systems of equations. Almost all numerical methods need to solve such a system in their solution procedure. There have been a lot of researches on this subject and it is still a very hot research area. One of the best methods to increase the performance of this operation is using graphics processing units (GPUs). These processors have had a great improvement in their processing capabilities. In this research, a new method to perform this operation using open computing language (OpenCL) is presented. The results show that by optimizing the parameters of this method one can gain a much higher performance compared to CPUs,***

---

* : مسئول مکاتبات، پست الکترونیکی: mossailby@eng.ui.ac.ir

even when using the open multi-processing standard (OpenMP). Also, the results show that there is not much sensitivity near the optimal parameters, which paves the way to estimate them from the matrix properties.

Keywords: *Sparse matrix-vector product, Graphics processing unit (GPU), Open computing language (OpenCL)*

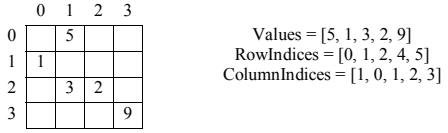
۱- مقدمه

بیشتر روش‌های عددی در قسمتی از روند حل خود نیازمند حل دستگاه‌های معادلات خطی هستند. با افزایش حجم مسئله، تعداد این معادلات نیز افزایش یافته و به سادگی می‌تواند به میلیون‌ها معادله برسد. در این موارد دیگر استفاده از روش‌های مستقیم[۱] حل دستگاه‌های معادلات خطی همانند روش چولسکی[۲] به سادگی امکانپذیر نیست و به جای آن از روش‌های تکراری[۳] استفاده می‌شود. از جمله این روش‌ها می‌توان به روش گرادیان مزدوج[۴] و یا گرادیان مزدوج دوگانه[۵] اشاره کرد که همراه با سایر روش‌هایی از این دست، در زمره روش‌های زیرفضای کرایلف[۶] جای می‌گیرند. مهم‌ترین جزء این روش‌ها که میان تمام آن‌ها مشترک است، نیاز به محاسبه ضرب یک ماتریس (که معمولاً همان ماتریس ضرایب دستگاه معادلات است) در یک بردار است. از این رو این جزء، پایه پیاده‌سازی هر یک از روش‌های زیرفضای کرایلف است.

در اکثر روش‌های عددی، ماتریس ضرایب به‌دست آمده ماتریسی تنک[۷] است، بدان معنی که بیشتر درایه‌های آن را صفرها تشکیل می‌دهند که در ضرب ماتریس در بردار بی‌تاثیرند. از این رو برای آنکه حافظه رایانه صرف نگهداری این صفرها نشده و همچنین برای جلوگیری از انجام عملیات ریاضی بیهوده و اتلاف قدرت پردازش رایانه، روش‌های متعددی برای ذخیره ماتریس‌های تنک ابداع شده است که با استفاده از آن‌ها می‌توان تنها درایه‌های غیر صفر را به کمک مقدار کمی اطلاعات جانبی برای بازسازی ماتریس اصلی نگهداری کرد. یکی از قدیمی‌ترین این روش‌ها، روش سطر تنک فشرده[۸] [۱] است. این روش کاملاً کلی بوده می‌تواند بر خلاف برخی روش‌های دیگر هر ماتریسی با هر خصوصیتی (مانند تقارن) را نگهداری کند. این خصوصیات باعث شده است با وجود برخی مشکلات که در ادامه خواهد آمد، این روش هنوز هم پرکاربردترین روش نگهداری ماتریس‌های تنک باشد و بسیاری از کتابخانه‌های توابع و برنامه‌ها از آن به عنوان مبنای پیاده‌سازی استفاده کنند.

در شکل (۱) نحوه‌ی نگهداری یک ماتریس تنک به کمک روش سطر تنک فشرده نمایش داده شده است. در این روش به جای کل ماتریس، تنها سه بردار نگهداری می‌شود. اولین بردار، بردار مقادیر نام دارد و تمام مقادیر غیر صفر ماتریس را سطر به سطر در خود جای داده است. برای عملکرد بهتر این روش، سعی می‌شود که مقادیر یک سطر نیز به ترتیب ستونی کنار یکدیگر قرار داده شود. غیر از این بردار، دو بردار دیگر از جنس اعداد صحیح تعریف می‌شوند که یکی اختصاص به نگهداری اندیس محل شروع هر سطر در بردار مقادیر داشته و دیگری شماره ستون هر یک از اعداد ذخیره شده در بردار مقادیر را نگهداری می‌کند. به این ترتیب، ابعاد بردار اندیس‌های سطری و ستونی به ترتیب برابر تعداد سطرها و تعداد عناصر غیرصفر ماتریس است. برای دسترسی به عناصر غیرصفر یک سطر که در اعمالی مانند ضرب بردار در ماتریس مورد نیاز است، می‌توان اندیس سطر دلخواه و سطر پس از آن را از بردار اندیس‌های سطری استخراج کرده و به راحتی به این عناصر دسترسی پیدا کرد. برای این کار در تمام سطرها (از جمله سطر آخر) قابل انجام باشد، مرسوم است که یک اندیس صوری به انتهای اندیس‌های سطری به نحوی اضافه می‌شود که قاعده بالا در مورد سطر آخر نیز برقرار باشد. در نتیجه ابعاد بردارهای مقادیر و اندیس‌های ستونی برابر تعداد عناصر غیر صفر ماتریس و تعداد عناصر بردار اندیس‌های سطری، یکی بیش از تعداد سطرهای ماتریس خواهد بود.

با وجود قابلیت‌های ذکر شده برای روش سطر تنک فشرده، مشکلاتی نیز در این زمینه وجود دارد که مهم‌ترین



شکل ۱- نحوه‌ی نگهداری یک ماتریس در قالب سطر تنک فشرده

آن‌ها دسترسی نامنظم به حافظه‌ی رایانه در هنگام اعمالی مانند ضرب ماتریس در بردار است. این مورد باعث می‌شود در معماری‌هایی از رایانه که از حافظه سریع کمکی[۹] برای بالا بردن سرعت حافظه‌ی اصلی رایانه استفاده می‌کنند، تعداد عدم موفقیت در دسترسی[۱۰] بالا رفته و سرعت دسترسی به حافظه‌ی اصلی کاهش یابد. از آنجا که در عمل ضرب ماتریس تنک در بردار سرعت دسترسی به حافظه به شدت گلوگاهی است، این کاهش سرعت به کاهش سرعت محاسبات خواهد انجامید. در معماری‌های برداری نیز به علت نامنظمی دسترسی به حافظه‌ی اصلی مشکلات مشابهی روی خواهد داد. حدود ۸۰٪ از زمان حل یک دستگاه معادلات خطی به روش‌های زیرفضای کرایلف برای محاسبه حاصل‌ضرب ماتریس در بردار صرف می‌شود، از این رو انجام هر چه سریع‌تر این محاسبه، نقش بسیار عمده‌ای در حل سریع‌تر دستگاه‌های معادلات خطی دارد و این مورد هنوز موضوع تحقیقات بسیار زیادی را به خود اختصاص می‌دهد.

با افزایش بی‌سابقه حجم محاسبات مورد نیاز در مسائل مهندسی و محدودیت افزایش سرعت پردازنده‌ها به علت‌های فراوان، مهندسان برای انجام این محاسبات به روش‌های دیگری روی آوردند که اصطلاحاً به آن‌ها محاسبات با کارایی بالا[۱۱] گفته می‌شود. این روش‌ها را می‌توان به سه دسته عمده تفکیک کرد. در دسته اول که به آن‌ها حافظه گسترده[۱۲] نیز گفته می‌شود، الگوریتم مسئله مورد نظر به نحوی شکسته می‌شود که روی تعدادی رایانه مستقل یا گره محاسباتی[۱۳] قابل انجام باشد. از آنجایی که در مراحل میانی هر گره ممکن است به نتایج سایر گره‌ها نیاز داشته باشد، ارتباط میان گره‌ها باید به صورتی برقرار گردد. ساده‌ترین نوع این ارتباط، استفاده از شبکه‌های محلی[۱۴] است. در دسته دوم یا حافظه مشترک[۱۵]، از پردازنده‌های چند هسته‌ای[۱۶] استفاده شده و مسئله میان آن‌ها تقسیم می‌شود. هر یک از این دو دسته، مزایا و معایب خاص خود را دارا هستند و تحقیقات بسیار گسترده‌ای روی آن‌ها انجام شده و یا در حال انجام است. در دسته سوم که به لحاظ تاریخی بسیار جدیدتر از دو دسته قبلی است، از کمک‌پردازنده‌های محاسباتی[۱۷] برای انجام محاسبات سریع استفاده می‌شود. این کمک‌پردازنده‌ها می‌توانند شامل شتاب‌دهنده‌های محاسباتی[۱۸]، پردازنده‌های گرافیکی[۱۹] و یا سایر انواع کمک‌پردازنده‌ها باشند.

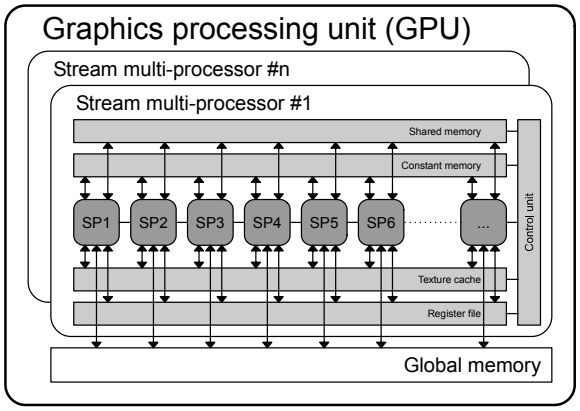
در سال‌های اخیر، قدرت محاسباتی پردازنده‌های گرافیکی به نحو بی‌سابقه‌ای بسیار بیش از پردازنده‌ها افزایش یافته است. این پردازنده‌ها در ابتدا تنها برای انجام اعمال گرافیکی مورد استفاده قرار می‌گرفتند، اما پس از مدتی محققان به استفاده از آن‌ها در انجام سریع‌تر محاسبات روی آوردند. از آنجا که نرم‌افزار و سخت‌افزار مورد استفاده تنها برای کارهای گرافیکی در نظر گرفته شده بود، امکان استفاده عمومی از آن‌ها در تمامی مسائل وجود نداشت. همچنین محققان مجبور بودند از کتابخانه‌های توابع گرافیکی مانند کتابخانه آزاد گرافیکی[۲۰] استفاده کنند که این مورد بر سختی کار می‌افزود [۲]. با وجود این مشکلات، محققان بسیاری از اعمال ابتدایی مانند عملیات ماتریس‌های پر و بردارها را روی پردازنده‌های گرافیکی پیاده‌سازی کرده و به کمک آن‌ها اعمالی مانند حل کردن دستگاه‌های معادلات خطی را انجام دادند [۳]. با فراگیرتر شدن استفاده از پردازنده‌های گرافیکی، سازندگان این پردازنده‌ها با ایجاد تغییرات گسترده در سخت‌افزار و نرم‌افزار، این پردازنده‌ها را به وسایلی ایدئال برای انجام بسیاری از محاسبات مهندسی تبدیل کردند. یکی از موفق‌ترین نمونه‌ها در این زمینه کودا[۲۱] [۴] متعلق به شرکت

انویدیا[۲۲] است. کودا پس از ارائه‌اش در فوریه ۲۰۰۷ میلادی و گسترش روزافزون خود، توانسته است بسیاری از محققان را برای استفاده از پردازنده‌های گرافیکی در محاسبات خود ترغیب کرده و به بسیاری از برنامه‌های علمی و تجاری راه پیدا کند. کودا با هدف استفاده در محاسبات با کارایی بالا به‌وجود آمده و به علت توسعه‌ی سریع، پشتیبانی از سیستم عامل‌های معروف، رایگان بودن ابزار توسعه و مستندات توانسته است جایگاه مهمی در این زمینه پیدا کند. تاکنون فعالیت‌های زیادی در زمینه استفاده از کودا در جهت فعالیت‌های علمی انجام شده است که به عنوان نمونه می‌توان به پیاده‌سازی کتابخانه‌های توابع مربوط به بردارها و ماتریس‌های اشاره کرد[۵].

مشکلی که در این هنگام به وجود آمد، اختصاصی بودن هر یک از زبان‌ها و کتابخانه‌های توابع به سخت‌افزار شرکت تولیدکننده خود بود که باعث می‌شد برنامه‌ای که بر مبنای یکی از این زبان‌ها و کتابخانه‌های نرم‌افزاری توسعه یافته است، قابل استفاده روی سایر سخت‌افزارها نباشد. برای رفع این مشکل در دسامبر سال ۲۰۰۸ میلادی، شرکت اپل[۲۳] به کمک تعدادی از شرکت‌های مهم در این زمینه، زبان آزاد محاسباتی[۲۴] را ارائه کرد [۶]. زبان آزاد محاسباتی یک استاندارد باز[۲۵] است که فارغ از نرم‌افزار و سخت‌افزار، یک زبان واحد و یک رابط کتابخانه‌های نرم‌افزاری را معرفی می‌کند که می‌تواند برای دسترسی و انجام محاسبات روی انواع پردازنده‌ها و کمک‌پردازنده‌ها مورد استفاده قرار گیرد. این استاندارد توسط برترین شرکت‌های سازنده پردازنده‌ها و پردازنده‌های گرافیکی حمایت می‌شود. مهم‌ترین ویژگی زبان آزاد محاسباتی، عدم وابستگی برنامه به نوع سخت‌افزار، نرم‌افزار و شرکت سازنده است که باعث می‌شود یک برنامه نوشته شده با این زبان بدون تغییر و یا با تغییرات جزیی قابل اجرا روی سخت‌افزارهای مختلف از شرکت‌های مختلف باشد. باید توجه داشت که این مسئله جدا از بهینه بودن برنامه روی سخت‌افزارهای مختلف است. این موضوع با توجه به پیچیدگی سخت‌افزار پردازنده‌های گرافیکی به خوبی قابل درک است. تحقیقات نشان می‌دهد در یک مقایسه متناسب، کارایی زبان آزاد محاسباتی روی سخت‌افزارهای مشابه می‌تواند با کارایی کودا رقابت کند [۷]. امروزه با فراگیر شدن زبان آزاد محاسباتی، این زبان نیز راه خود را به محاسبات عددی باز کرده است. از جمله کتابخانه‌های توابع بسیار رایج در این زمینه می‌توان به کتابخانه توابع وینا سی ال [۸] اشاره کرد. این کتابخانه علاوه بر یک مجموعه کامل از توابع مربوط به ماتریس‌های پر و بردارها، دارای توابعی برای حل دستگاه‌های معادلات خطی است.

۲- سخت‌افزار پردازنده‌های گرافیکی

همان گونه که پیش‌تر اشاره شد، پردازنده‌های گرافیکی در سال‌های اخیر رشد بی‌سابقه‌ای از لحاظ قدرت پردازش داشته‌اند. امروزه توان محاسباتی یک پردازنده گرافیکی چندین برابر سریع‌ترین پردازنده‌های موجود است. دلایل این رشد را می‌توان در چند عامل جستجو کرد. ابتدا باید گفت پردازنده‌های گرافیکی تعداد هسته‌های بسیار بیشتری از یک پردازنده عادی دارند. به عنوان نمونه یک پردازنده بسیار پیشرفته فعلی حداکثر دارای ۸ هسته است، در صورتی که یک پردازنده گرافیکی می‌تواند دارای ۵۱۲ هسته باشد. بدین سبب پردازنده‌های گرافیکی در زمره پردازنده‌های با تعداد هسته‌های زیاد[۲۶] طبقه‌بندی می‌شوند. باید دانست که هر هسته یک پردازنده گرافیکی بسیار ضعیف‌تر از یک هسته پردازنده عادی است، اما به لحاظ تعداد بسیار بیشتر و ساختار خاص پردازنده‌های گرافیکی، این پردازنده‌ها می‌توانند توان محاسباتی بسیار بالایی از خود بروز دهند. دومین عامل در بالا بودن قدرت محاسباتی این پردازنده‌ها را باید در نوع معماری آن‌ها جستجو کرد. در این معماری، یک پردازنده گرافیکی مشتمل بر تعدادی واحد چندپردازنده[۲۷] است که هر یک به نوبه خود به تعدادی واحد محاسباتی تقسیم می‌شوند. شکل (۲) نمایی از این معماری را نشان می‌دهد. در هر یک از



شکل ۲- نمایی از معماری پردازنده‌های گرافیکی

چندپردازنده‌ها در هر زمان تنها یک دستور مشابه روی تمام واحدهای محاسباتی با داده‌های گوناگون اجرا می‌شود. به همین دلیل در این معماری، محاسباتی که شامل اجرای یک سری دستور خاص روی حجم بالایی از داده‌ها باشد، با سرعت بسیار بالایی اجرا می‌شود. به عنوان نمونه می‌توان به جمع دو بردار با تعداد بسیار زیاد درایه‌ها اشاره کرد. اگر به دلیلی همانند وجود دستورات شرطی لازم باشد برخی از واحدهای محاسباتی دستورات دیگری را انجام دهند، واحد کنترل شاخه‌های مختلف دستورات شرطی را به صورت نوبتی در صف اجرا قرار می‌دهد. واضح است که این کار عملکرد کلی را به شدت تحت تاثیر قرار می‌دهد و باید تا حد امکان از نوشتن برنامه‌ها به این صورت خودداری کرد. در اینجا باید اشاره کرد در مقایسه با پردازنده‌ها، هزینه به‌وجود آوردن یک رشته اجرا[۲۸] در پردازنده‌های گرافیکی بسیار کمتر است. بدین سبب یک رشته عملیات را می‌توان به هزاران رشته اجرایی تقسیم کرد که واحد کنترل پردازنده گرافیکی آن‌ها را به ترتیب به چندپردازنده‌ها و در نتیجه واحدهای محاسباتی تخصیص داده و آن‌ها را اجرا می‌کند. عامل سوم را می‌توان ساختار حافظه در پردازنده‌های گرافیکی دانست. در مقایسه با پردازنده‌ها، ساختار حافظه پردازنده‌های گرافیکی بسیار پیچیده‌تر است. چهار نوع حافظه مختلف در پردازنده‌های گرافیکی به کار می‌رود که هر یک کاربرد خاص خود را دارند. حافظه اصلی[۲۹] پردازنده گرافیکی که میان همه چندپردازنده‌ها مشترک است، بیشترین حجم را دارا بوده ولی دسترسی به آن بسیار کندتر از سایر انواع حافظه است. این تنها نوعی از انواع حافظه‌های موجود روی پردازنده‌های گرافیکی است که امکان انتقال داده‌ها از حافظه اصلی رایانه به آن وجود دارد. این انتقال از طریق درگاه داده‌ها[۳۰] انجام می‌گیرد و با وجود سرعت بالا، چنانچه برنامه‌ای نیاز به انتقال مکرر داده‌ها از حافظه اصلی رایانه به حافظه اصلی پردازنده گرافیکی داشته باشد، کارایی آن به شدت کاهش خواهد یافت. از دیگر انواع حافظه در پردازنده‌های گرافیکی می‌توان به حافظه ثابت[۳۱] برای مقادیری که در طول اجرای یک رشته‌محاسباتی تغییر نمی‌کنند اشاره کرد. همچنین می‌توان از حافظه محلی[۳۲] که میان واحدهای محاسباتی یک چندپردازنده به صورت مشترک قابل دسترسی است و نیز حافظه خصوصی[۳۳] که مختص هر واحد محاسباتی است، نام برد. حجم این حافظه‌ها بسیار کمتر از حافظه اصلی بوده و عمدتاً سرعت دسترسی به آن‌ها بسیار بیشتر است. به عنوان مثال دسترسی به حافظه محلی می‌تواند ۶۰۰ تا ۷۰۰ برابر سریع‌تر از حافظه اصلی باشد. نکته بسیار مهم در این‌جا شناخت بسیار دقیق انواع حافظه و استفاده صحیح از آن‌ها در الگوریتم

```
for (unsigned int i = 0; i < NumberOfRows; i++)
{
double Sum = 0.00;

for (unsigned int j = 0; j < NumberOfColumns; j++)
{
Sum += A[i][j] * X[j];
}

Y[i] = Sum;
}
```

شکل ۳- پیاده‌سازی الگوریتم ضرب ماتریس‌های متراکم در بردار روی پردازنده‌ها

```
for (unsigned int i = 0; i < NumberOfRows; i++)

{
  double Sum = 0.00;

  for (unsigned int j = RowIndices[i]; j < RowIndices[i + 1]; j++)
  {
    Sum += Values[j] * X[ColumnIndices[j]];
  }

  Y[i] = Sum;
}
```

شکل ۴- پیاده‌سازی الگوریتم ضرب ماتریس‌های تنک در بردار روی پردازنده‌ها

محاسباتی است که این موضوع می‌تواند سرعت انجام محاسبات را به حداکثر نزدیک کند. آنچه ذکر شد تنها گوشه‌ای از پیچیدگی‌های سخت‌افزار پردازنده‌های گرافیکی جدید است. شناخت دقیق سخت‌افزار می‌تواند راهگشای انجام محاسبات سریع با استفاده از پردازنده‌های گرافیکی باشد.

۳- پیاده‌سازی الگوریتم ضرب ماتریس تنک در بردار روی پردازنده‌های گرافیکی و تحقیقات مرتبط

پیاده‌سازی الگوریتم ضرب ماتریس‌های متراکم در بردار روی پردازنده‌ها بسیار ساده است. شکل (۳) الگوریتم این کار را در قالب یک برنامه به زبان C++ نشان می‌دهد. هنگامی که ماتریس تنک بوده و به روش سطر تنک متراکم نگهداری شده باشد، باید تغییراتی در این الگوریتم صورت گیرد. واضح است که در این حالت اعمال ضرب و جمع تنها باید روی مقادیر غیرصفر انجام گیرد. برای این کار با توجه به شکل (۴) و با استفاده از بردار کمکی اندیس سطرها، ابتدا و انتهای هر سطر به‌دست می‌آید. سپس با انجام یک حلقه روی این سطر مقادیر از بردار مقادیر ماتریس استخراج شده و با کمک بردار اندیس‌های ستونی، در درایه مربوطه از بردار x ضرب می‌شود. چنانچه تمامی این مقادیر برای یک سطر با یکدیگر جمع شوند، یک درایه بردار حاصل‌ضرب نهایی به دست می‌آید. چند نکته در این الگوریتم قابل توجه است. نخست این که دسترسی به بردار x بسیار نامنظم است. همان‌طور که پیش‌تر اشاره شد، این مورد کارایی کلی را به شدت تحت تاثیر قرار می‌دهد. نکته دوم نیاز به حاصل‌جمع تمام حاصل‌ضرب‌های میانی[۳۴] در هر سطر برای به‌دست آوردن یک درایه از بردار حاصل‌ضرب نهایی است که قسمت دوم الگوریتم را تشکیل می‌دهد. این مورد به عنوان یکی از اعمال ابتدایی[۳۵] در بحث پردازنده‌های گرافیکی مطرح است و به نام عملیات کاهش[۳۶]

شناخته می‌شود. پیاده‌سازی یک عملیات کاهش با کارایی مناسب در پردازنده‌های گرافیکی کار ساده‌ای نیست و نیاز به دقت فراوان دارد. به عنوان آخرین مورد باید توجه داشت که تعداد عناصر غیرصفر در هر سطر در یک ماتریس لزوماً با یکدیگر برابر نیست. همچنین دلیلی ندارد این تعداد بر تعداد واحدهای محاسباتی در یک چندپردازنده بخش‌پذیر باشد. بنابراین همواره تعدادی از واحدهای محاسباتی بدون عملیات خواهند ماند که این مورد از کارایی کلی الگوریتم به شدت خواهد کاست و در نتیجه، میزان بهینه بودن الگوریتم ارتباط تنگاتنگی با نحوه‌ی توزیع مقادیر غیرصفر ماتریس خواهد داشت. برخی از محققان تلاش کرده‌اند به نحوی با تغییر تعداد سطرهایی که به یک چندپردازنده سپرده می‌شود، الگوریتم خود را برای دسته‌ای از ماتریس‌ها بهینه کنند [۹-۱۱]. در مواردی نیز این کار توسط یک برنامه خارجی و به صورت خودکار انجام شده است [۱۲ و ۱۳]. البته لازم به ذکر است تقریباً تمامی کارهای انجام شده در این زمینه بر مبنای کودا انجام شده است و در نتیجه منحصر به سخت‌افزارهای شرکت انویدیاست. در دسته دیگری از تحقیقات انجام شده، سعی شده است با تغییر نحوه‌ی نگهداری ماتریس بر کارایی الگوریتم افزوده شود [۹ و ۱۱]. با توجه به کاربرد بسیار زیاد روش نگهداری سطر تنک فشرده، استفاده از چنین الگوریتم‌هایی مستلزم تغییرات کلی در ساختار برنامه‌های قدیمی و یا تبدیل نوع ماتریس است که هیچ کدام چندان مطلوب نیست. همچنین سایر روش‌های نگهداری ماتریس‌های تنک نیز مشکلاتی خاص خود دارا بوده و نمی‌توان آن‌ها را حل نهایی مشکل دانست. در این مقاله سعی می‌شود با استفاده از چندین تکنیک، از جمله تعیین تعداد سطرها در هر چندپردازنده در زمان اجرای برنامه، دادن آگاهی اولیه به کامپایلر برای بهینه‌سازی هر چه بیشتر و استفاده از یک قسمت عملیات کاهش با حداکثر کارایی، کارایی کلی الگوریتم بدون تغییر در ساختار ماتریس تا حد امکان افزایش یابد. این تکنیک‌ها در قسمت‌های آینده به اختصار بیان خواهد شد.

۴- مبانی کلی یک برنامه به زبان آزاد محاسباتی

مدل برنامه‌نویسی در زبان آزاد محاسباتی با آنچه در سایر مدل‌های برنامه‌نویسی وجود دارد، قدری متفاوت است. در این زبان الگوریتم به صورت یک سری رشته اجرایی در نظر گرفته می‌شود که همگی یک سری دستورات یکسان را اجرا می‌کنند. این دستورات به صورت یک تابع در نظر گرفته می‌شود که آن را هسته[۳۷] می‌نامند. می‌توان تصور کرد تمام واحدهای محاسباتی این تابع را فراخوانی و اجرا می‌کنند. به عنوان نمونه، برنامه جمع دو بردار با یکدیگر در نظر گرفته می‌شود. شکل (۵) الگوریتم لازم برای این کار را روی یک پردازنده و نیز هسته معادل آن را در زبان آزاد محاسباتی نشان می‌دهد. مهم‌ترین تفاوت این دو برنامه در حذف شدن حلقه خارجی در هسته است که با اختصاص قسمت داخلی حلقه در هر شمارنده به واحدهای محاسباتی، عملاً کار انجام شده در دو برنامه معادل خواهد بود. توضیح چند نکته در این‌جا ضروری به نظر می‌رسد. نخست این که هر واحد محاسباتی نیاز دارد موقعیت خود را در میان واحدهای محاسباتی دیگر بداند که این امر معادل استفاده از شمارنده حلقه در برنامه مربوط به پردازنده در شکل(۵) است. همان گونه که در شکل مشاهده می‌شود، برای این کار هسته می‌تواند از توابع موجود در زبان آزاد محاسباتی استفاده کند. نکته دیگری که باید بدان اشاره کرد، این است که به دلایل بسیاری، ترتیب اجرای هسته روی واحدهای محاسباتی در چندپردازنده‌های مختلف قابل پیش‌بینی نیست. از این رو هنگامی یک حلقه خارجی را بدین صورت می‌توان به یک برنامه زبان آزاد محاسباتی تبدیل کرد که هر بار اجرای حلقه از دفعات دیگر کاملاً مستقل باشد. برنامه ساده‌ای که در این قسمت بدان پرداخته شد، دارای این خصوصیت است، اما در برنامه‌های پیچیده‌تر مانند عملیات کاهش این کار امکانپذیر نیست و باید با ارائه‌ی یک الگوریتم سازگار با مدل برنامه‌نویسی زبان آزاد محاسباتی، عملیات

```
void AddVec(
  const double *x,
  const double *y,
  double *z,
  unsigned int N)
{
  for (unsigned int i = 0; i < N; i++)
  {
    z[i] = x[i] + y[i];
  }
}


__kernel void AddVec(
  __global const double *x,
  __global const double *y,
  __global double *z,
  unsigned int N)
{
  unsigned int i = get_global_id(0);

  if (i < N)
  {
    z[i] = x[i] + y[i];
  }
}
```

شکل ۵- برنامه جمع دو بردار با یکدیگر روی پردازنده و هسته‌ی معادل آن در زبان آزاد محاسباتی

```
__kernel void SpMV_Naive(
  __global unsigned int const *RowIndices,
  __global unsigned int const *ColumnIndices,
  __global unsigned int const *Values,
  __global double const *X,
  __global double *Y,
  unsigned int N)
{
  unsigned int i = get_global_id(0);

  if (i < N)
  {
    double Sum = 0.00;

    for (unsigned int j = RowIndices[i]; j < RowIndices[i + 1]; j++)
    {
      Sum += Values[j] * X[ColumnIndices[j]];
    }

    Y[i] = Sum;
  }
}
```

شکل ۶- ساده‌ترین روش پیاده‌سازی الگوریتم ضرب ماتریس تنک در بردار در زبان آزاد محاسباتی

```
// WORKGROUP_SIZE_BITS and ROWS_PER_WORKGROUP_BITS are to be defined
// on the command-line

#define WORKGROUP_SIZE (1 << WORKGROUP_SIZE_BITS)
#define ROWS_PER_WORKGROUP (1 << ROWS_PER_WORKGROUP_BITS)

#define LOCAL_WORKGROUP_SIZE_BITS (WORKGROUP_SIZE_BITS - \
ROWS_PER_WORKGROUP_BITS)
#define LOCAL_WORKGROUP_SIZE (1 << LOCAL_WORKGROUP_SIZE_BITS)

__kernel void __attribute__((reqd_work_group_size(WORKGROUP_SIZE, 1, 1))) SpMV(
  __global unsigned int const *RowIndices,
  __global unsigned int const *ColumnIndices,
  __global double const *Values,
  __global double const *X,
  __global double *Y,
  unsigned int N,
  __local double *Buffer)
{
  const unsigned int gid = get_group_id(0);
  const unsigned int tid = get_local_id(0);

  const unsigned int lgid = tid >> LOCAL_WORKGROUP_SIZE_BITS;
  const unsigned int ltid = tid & (LOCAL_WORKGROUP_SIZE - 1);

  const unsigned int Row = (gid << ROWS_PER_WORKGROUP_BITS) + lgid;

  if (Row < N)
  {
    Buffer[tid] = 0.00;

    const unsigned int Start = RowIndices[Row];
    const unsigned int End = RowIndices[Row + 1];

    // Actual multiplication
    for (unsigned int i = Start + ltid; i < End; i += LOCAL_WORKGROUP_SIZE)
    {
      Buffer[tid] += Values[i] * X[ColumnIndices[i]];
    }
  }

  // __local memory barrier
  barrier(CLK_LOCAL_MEM_FENCE);

  // Reduction of results

#if LOCAL_WORKGROUP_SIZE > 512

  if (Row < N)
  {
    if (ltid < 512)
    {
      Buffer[tid] += Buffer[tid + 512];
    }
  }

  barrier(CLK_LOCAL_MEM_FENCE);
```

```
#endif

#if LOCAL_WORKGROUP_SIZE > 256

  if (Row < N)
  {
    if (ltid < 256)
    {
      Buffer[tid] += Buffer[tid + 256];
    }
  }

  barrier(CLK_LOCAL_MEM_FENCE);

#endif

#if LOCAL_WORKGROUP_SIZE > 128

  if (Row < N)
  {
    if (ltid < 128)
    {
      Buffer[tid] += Buffer[tid + 128];
    }
  }

  barrier(CLK_LOCAL_MEM_FENCE);

#endif

#if LOCAL_WORKGROUP_SIZE > 64

  if (Row < N)
  {
    if (ltid < 64)
    {
      Buffer[tid] += Buffer[tid + 64];
    }
  }

  barrier(CLK_LOCAL_MEM_FENCE);

#endif

#if LOCAL_WORKGROUP_SIZE > 32

  if (Row < N)
  {
    if (ltid < 32)
    {
      Buffer[tid] += Buffer[tid + 32];
    }
  }

  barrier(CLK_LOCAL_MEM_FENCE);

#endif
```

```
#if LOCAL_WORKGROUP_SIZE > 16

  if (Row < N)
  {
    if (ltid < 16)
    {
      Buffer[tid] += Buffer[tid + 16];
    }
  }

  barrier(CLK_LOCAL_MEM_FENCE);

#endif

#if LOCAL_WORKGROUP_SIZE > 8

  if (Row < N)
  {
    if (ltid < 8)
    {
      Buffer[tid] += Buffer[tid + 8];
    }
  }

  barrier(CLK_LOCAL_MEM_FENCE);

#endif

#if LOCAL_WORKGROUP_SIZE > 4

  if (Row < N)
  {
    if (ltid < 4)
    {
      Buffer[tid] += Buffer[tid + 4];
    }
  }

  barrier(CLK_LOCAL_MEM_FENCE);

#endif

#if LOCAL_WORKGROUP_SIZE > 2

  if (Row < N)
  {
    if (ltid < 2)
    {
      Buffer[tid] += Buffer[tid + 2];
    }
  }

  barrier(CLK_LOCAL_MEM_FENCE);

#endif

#if LOCAL_WORKGROUP_SIZE > 1
```

```
  if (Row < N)
  {
    if (ltid < 1)
    {
      Buffer[tid] += Buffer[tid + 1];
    }
  }

  barrier(CLK_LOCAL_MEM_FENCE);

#endif

  if (Row < N)
  {
    // Store final result
    if (ltid == 0)
    {
      Y[Row] = Buffer[tid];
    }
  }
}
```

شکل ۷ - پیاده‌سازی الگوریتم پیشنهادی ضرب ماتریس تنک در بردار در زبان آزاد محاسباتی

```
#pragma omp parallel for

for (unsigned int i = 0; i < NumberOfRows; i++)
{
  double Sum = 0.00;

  for (unsigned int j = RowIndices[i]; j < RowIndices[i + 1]; j++)
  {
    Sum += Values[j] * X[ColumnIndices[j]];
  }

  Y[i] = Sum;
}
```

شکل ۸- پیاده‌سازی الگوریتم ضرب ماتریس‌های تنک در بردار روی پردازنده‌ها با استفاده از استاندارد باز چندپردازنده

مورد نظر را انجام داد. همچنین در صورتی که یک قسمت از برنامه به نتایج قسمت قبل نیاز داشته باشد، هر یک از این قسمت‌ها باید در قالب یک هسته مستقل معرفی و به ترتیب فراخوانی شوند. فراخوانی هر هسته جدا از زمان اجرای آن، به مقداری زمان نیاز دارد و بنابراین باید برنامه‌ها به شکلی نوشته شوند که کمترین تعداد فراخوانی هسته‌ها را داشته باشند. واضح است با وجود بالاتر بودن قدرت محاسباتی پردازنده‌های گرافیکی، نمی‌توان انتظار داشت عملکرد هر برنامه‌ای در زبان آزاد محاسباتی بهتر از پردازنده‌ها باشد. حتی گاهی بهتر است بخشی از برنامه روی پردازنده و بخشی روی پردازنده گرافیکی اجرا شود.

۵- پیاده‌سازی الگوریتم ضرب ماتریس تنک در بردار توسط زبان آزاد محاسباتی

ساده‌ترین روش پیاده‌سازی الگوریتم ضرب ماتریس تنک در بردار در شکل (۶) نمایش داده شده است. در این روش،همان گونه که پیش‌تر توضیح داده شد، تنها حلقه خارجی در برنامه شکل (۵) حذف و محتویات حلقه به عنوان هسته مورد نظر معرفی شده است. واضح است که هر واحد محاسباتی، محاسبه حاصل‌ضرب یک سطر از ماتریس را بر عهده خواهد داشت. در صورتی که تعداد عناصر غیرصفر در هر سطر با یکدیگر مساوی نباشد، همواره تعدادی از واحدهای محاسباتی بدون عملیات خواهند ماند. همچنین الگوی دسترسی به حافظه بسیار درهم ریخته است. توضیح بیشتر در مورد بهترین نوع دسترسی به حافظه در پردازنده‌های گرافیکی خارج از حوصله این نوشتار است و خواننده می‌تواند به مراجع [۱۴ و ۱۵] مراجعه کند. این موارد باعث می‌شود کارایی این هسته بسیار پایین باشد.

در این تحقیق از چندین تکنیک برای افزایش کارایی این هسته استفاده شده است. اولین مورد به کارگیری یک الگوریتم جدید برای استفاده از چندین واحد محاسباتی برای محاسبه حاصل‌ضرب یک سطر از ماتریس در بردار است. از این دیدگاه می‌توان الگوریتم‌های ارائه شده در مراجع [۹ و ۱۱] را حالت خاصی از الگوریتم ارائه شده در این تحقیق دانست. تا حد اطلاع نگارنده ارائه چنین الگوریتمی برای اولین بار انجام شده است. چنانچه تمام واحدهای محاسباتی یک گروه کاری[۳۸] بدون عملیات بماند، واحد کنترل عملیات جدیدی را به آن‌ها تخصیص می‌دهد و در نتیجه بر کارایی الگوریتم اضافه می‌شود. در این روش واحدهای محاسباتی متوالی مربوط به یک سطر به خانه‌های حافظه متوالی دسترسی پیدا می‌کنند. این الگوی دسترسی به حافظه برای پردازنده‌های گرافیکی بسیار مناسب بوده و سرعت دسترسی به حافظه و در نتیجه کارایی کلی را به شکل مناسبی بهبود می‌بخشد. واضح است که بهترین تعداد واحد محاسباتی برای هر سطر به نحوه‌ی توزیع عناصر غیرصفر در ماتریس بستگی دارد. آنچه در این تحقیق برای اولین بار پیاده‌سازی شده است، نحوه‌ی خاص نوشتن هسته به شکلی است که با کمک ماکروهای پیش‌پردازنده[۳۹] می‌توان به صورت کاملاً بهینه تعداد واحد محاسباتی در هر گروه کاری را تغییر داد. پارامتر بسیار مهم دیگر در این زمینه تعداد سطر اختصاص‌یافته به هر گروه کاری است. این پارامتر نیز به صورت بهینه قابل تغییر با کمک ماکروهای پیش‌پردازنده است. بیشتر محاسبات لازم وابسته به این پارامترها در زمان ترجمه‌ی هسته انجام می‌گیرد و حاصل آن یک هسته بسیار بهینه است. در بسیاری از کاربردها مانند حل یک دستگاه معادلات خطی، ماتریس با الگوی یکسان عناصر غیرصفر بارها و بارها باید در بردار ضرب شوند. یک برنامه می‌تواند به نحوی نوشته شود که در زمان اجرا با تغییر این پارامترها، بهینه‌ترین حالت ممکن را برای این الگوی عناصر غیرصفر انتخاب و استفاده کند. از آنجا که این محاسبات بخش بسیار عمده‌ای از زمان کلی حل یک مسئله را تشکیل می‌دهد، این بهینه‌سازی کاملاً به صرفه است.

تکنیک دیگری که برای افزایش کارایی هسته‌ی ضرب ماتریس در بردار به کار رفته است، استفاده از یک بخش عملیات کاهش با کارایی بالاست. این بخش با کمک ماکروهای پیش‌پردازنده به شکلی نوشته شده است که با پارامترهای گفته شده در بالا سازگار بوده و تغییر پارامترها به شکل بهینه‌ای در نظر گرفته می‌شود. عملیات کاهش در چند مرحله انجام می‌شود. هنگامی که قسمت اول الگوریتم پایان می‌یابد، به ازای هر یک از اعضای گروه کاری یک حاصل‌جمع میانی به‌دست آمده است. نحوه کار عملیات کاهش بدین صورت است که ابتدا هر گروه کاری به دو قسمت مساوی تقسیم می‌شود. نیمه اول حاصل‌جمع میانی خود را با حاصل جمع میانی نیمه دوم جمع کرده و جایگزین حاصل جمع میانی خود می‌کنند. با این کار تعداد محاسباتی

جدول ۱- مشخصات سیستم‌های رایانه‌های مورد استفاده

ردیف	پردازنده	تعداد هسته‌ها	حافظه اصلی	سیستم عامل	پردازنده گرافیکی	حافظه‌ی اصلی	بستر نرم‌افزاری
۱	AMD Phenom Quad core 9950	4	4 GB	Ubuntu 10.10 (64 bit Linux)	NVIDIA GeForce GTX 280	1 GB	NVIDIA CUDA 4.1
۲	Intel Core2 Quad Q8300	4	4 GB	Ubuntu 10.10 (64 bit Linux)	AMD Radeon HD 6970	2 GB	AMD APP SDK 2.6

جدول ۲- سرعت حافظه سیستم‌های رایانه‌های مورد استفاده

ردیف	پردازنده	پردازنده گرافیکی	
	Add	Add	Copy
۱	۷/۶۲	۳۶/۴۱	۱۲۰/۹۹
۲	۴/۸۶	۶۷/۲۷	۱۳۸/۵۶

نیز نتیجه نهایی را در محل مربوطه در حافظه اصلی ذخیره می‌کند. از آنجا که عملیات کاهش حداکثر در یک گروه کاری انجام می‌گیرد، از حافظه محلی که میان واحدهای محاسباتی مشترک است، برای ذخیره این حاصل‌جمع‌های میانی استفاده می‌شود. این مورد کارایی این الگوریتم را به شدت افزایش می‌دهد.

۶- مثال‌های عددی

شکل (۷) متن کامل هسته پیاده‌سازی شده در زبان آزاد محاسباتی را نشان می‌دهد. برای بررسی کارایی هسته پیاده‌سازی شده، دو سیستم رایانه‌ای انتخاب شد. مشخصات پردازنده، حافظه اصلی، پردازنده گرافیکی، حافظه اصلی گرافیکی و مشخصات نرم‌افزاری این دو سیستم در جدول(۱) آمده است. همان گونه که پیش‌تر اشاره شد، سرعت عمل ضرب ماتریس در بردار، بسیار وابسته به سرعت حافظه‌ی مورد استفاده است. برای بررسی سرعت حافظه در سیستم‌های مورد استفاده، از آزمون استریم[۴۰] [۱۶] استفاده شده است. این آزمون را می‌توان یکی از معروف‌ترین آزمون‌ها در این زمینه دانست. در این آزمون سرعت چند عملیات ساده مانند جمع دو بردار و یا کپی کردن بخشی از حافظه به عنوان نمودی از سرعت قابل دسترسی حافظه به‌دست می‌آید. نظیر همین مورد توسط زبان آزاد محاسباتی روی پردازنده گرافیکی پیاده‌سازی شد. جدول (۲) نتایج حاصل از انجام آزمون را نشان می‌دهد. واضح است که سرعت حافظه در پردازنده‌های گرافیکی به مراتب بیش از سرعت حافظه در پردازنده است و بنابراین می‌توان انتظار داشت که عمل ضرب ماتریس در بردار روی پردازنده‌های گرافیکی سریع‌تر انجام گیرد.

پیش‌تر اشاره شد که خصوصیات ماتریس مورد استفاده و از آن جمله نحوه توزیع مقادیر غیر صفر، تاثیر بسیاری در نتایج به‌دست آمده دارد. از این رو، برای به‌دست آوردن نتایجی که بتواند گویای عملکرد واقعی روش باشد، از یک

جدول ۳- مشخصات ماتریس‌ها

ماتریس	تعداد سطرها	تعداد ستون‌ها	تعداد عناصر غیر صفر	متوسط تعداد عناصر غیر صفر در هر سطر
Dense	۲۰۰۰	۲۰۰۰	۴۰۰۰۰۰۰	۲۰۰۰
Protein	۳۶۴۱۷	۳۶۴۱۷	۴۳۴۴۷۶۵	۱۱۹
FEM / Spheres	۸۳۳۳۴	۸۳۳۳۴	۶۰۱۰۴۸۰	۷۲
FEM / Cantilever	۶۲۴۵۱	۶۲۴۵۱	۴۰۰۷۳۸۳	۶۴
Wind Tunnel	۲۱۷۹۱۸	۲۱۷۹۱۸	۱۱۶۳۴۴۲۴	۵۳
FEM / Harbor	۴۶۸۳۵	۴۶۸۳۵	۲۳۷۴۰۰۱	۵۱
QCD	۴۹۱۵۲	۴۹۱۵۲	۱۹۱۶۹۲۸	۳۹
FEM / Ship	۱۴۰۸۷۴	۱۴۰۸۷۴	۷۸۱۳۴۰۴	۵۵
Economics	۲۰۶۵۰۰	۲۰۶۵۰۰	۱۲۷۳۳۸۹	۶
Epidemiology	۵۲۵۸۲۵	۵۲۵۸۲۵	۲۱۰۰۲۲۵	۴
FEM / Accelerator	۱۲۱۱۹۲	۱۲۱۱۹۲	۲۶۲۴۳۳۱	۲۲
Circuit	۱۷۰۹۹۸	۱۷۰۹۹۸	۹۵۸۹۳۶	۶
Webbase	۱۰۰۰۰۰۵	۱۰۰۰۰۰۵	۳۱۰۵۵۳۶	۳
LP	۴۲۸۴	۱۰۹۲۶۱۰	۱۱۲۷۹۷۴۸	۲۶۳۳

مجموعه ماتریس‌های تنک که در بسیاری از مقالات به آن‌ها استناد می‌شود، استفاده شد. این مجموعه شامل ۱۴ ماتریس است که طیف بسیار گسترده‌ای از مسائل را پوشش می‌دهد. مشخصات این ماتریس‌ها در جدول (۳) آمده است. توضیحات بیشتر در مورد این ماتریس‌ها و منشاء آن‌ها در مرجع [۱۷] موجود است.

پیاده‌سازی برنامه اصلی توسط زبان برنامه‌نویسی C++ انجام شد. برای ترجمه برنامه‌ها نیز از کامپایلر g++ 4.4.5 که به همراه سیستم عامل عرضه می‌شود استفاده شده است. زمان اجرای هسته بر مبنای زمان روی پردازنده (و نه پردازنده گرافیکی) به کمک دقیق‌ترین زمان‌سنج موجود در سیستم اندازه‌گیری شده است. این مورد از آن جهت واجد اهمیت است که بسیاری از مولفان تنها زمان اجرای هسته روی پردازنده گرافیکی را گزارش می‌کنند که در نتیجه زمان صرف شده برای آماده‌سازی هسته برای اجرا در آن منظور نمی‌شود. در تمامی برنامه‌ها، برای بررسی درستی جواب‌ها، نتایج به‌دست آمده با نتایج برنامه مرجع روی پردازنده مقایسه شده و همگی مورد تایید قرار گرفته است. در مورد روش پیشنهادی پارامترها با زمان‌گیری به نحوی انتخاب شده‌اند که سریع‌ترین حل ممکن به دست آید.

جدول‌های (۴) و (۵) نتایج به‌دست آمده را به تفکیک ماتریس‌های مورد استفاده نشان می‌دهند. در ستون مربوط به پردازنده، زمان اجرای الگوریتم مرجع شکل (۴) روی پردازنده نشان داده شده است. این زمان برای مقایسه سایر روش‌ها به کاربرده شده است و افزایش سرعت‌ها نسبت به آن سنجیده شده‌اند. در ستون‌های بعد کارایی استفاده از

جدول ۴- زمان انجام عمل ضرب بر حسب میلی ثانیه و نسبت افزایش سرعت در سیستم شماره ۱

ماتریس	CPU	CPU (OpenMP)		GPU (Naïve)		GPU (Proposed)	
	زمان اجرا	زمان اجرا	سرعت نسبی	زمان اجرا	سرعت نسبی	زمان اجرا	سرعت نسبی
Dense	۱۲/۰۹	۵/۷۰	۲/۱۲	۱۰/۴۲	۱/۱۶	۰/۷۹	۱۵/۲۳
Protein	۱۱/۵۳	۶/۴۱	۱/۸۰	۵/۶۷	۲/۰۳	۱/۳۱	۸/۷۷
FEM / Spheres	۱۶/۵۱	۹/۰۷	۱/۸۲	۶/۵۶	۲/۵۲	۱/۸۱	۹/۱۱
FEM / Cantilever	۱۱/۰۰	۵/۸۸	۱/۸۷	۴/۵۷	۲/۴۰	۱/۲۹	۸/۵۳
Wind Tunnel	۳۲/۱۰	۱۶/۱۹	۱/۹۸	۱۱/۳۷	۲/۸۲	۳/۱۹	۱۰/۰۵
FEM / Harbor	۶/۵۹	۴/۲۷	۱/۵۴	۲/۹۶	۲/۲۳	۰/۸۲	۸/۰۶
QCD	۵/۶۳	۳/۰۳	۱/۸۶	۱/۸۴	۳/۰۵	۰/۵۹	۹/۵۲
FEM / Ship	۲۲/۰۰	۱۱/۷۸	۱/۸۷	۸/۵۰	۲/۵۹	۲/۱۱	۱۰/۴۱
Economics	۶/۱۱	۳/۳۴	۱/۸۳	۱/۵۶	۳/۹۳	۰/۹۱	۶/۷۳
Epidemiology	۷/۶۹	۴/۶۶	۱/۶۵	۱/۲۱	۶/۳۶	۰/۷۴	۱۰/۳۸
FEM / Accelerator	۱۰/۷۷	۵/۲۱	۲/۰۷	۳/۲۹	۳/۲۷	۱/۰۷	۱۰/۱۰
Circuit	۵/۲۷	۳/۱۴	۱/۶۸	۱/۶۰	۳/۳۰	۰/۸۵	۶/۲۰
Webbase	۱۷/۳۱	۸/۹۶	۱/۹۳	۱۱/۷۸	۱/۴۷	۶/۶۶	۲/۶۰
LP	۶۹/۱۳	۳۷/۲۹	۱/۸۵	۸۱/۲۳	۰/۸۵	۵/۳۲	۱۳/۰۰

استاندارد چندپردازنده باز[۴۱] شکل (۸) نسبت به الگوریتم مرجع نشان داده شده است. دیده می‌شود استفاده از این روش که جزو روش‌های حافظه‌ی مشترک است، در برخی موارد سرعت اجرای الگوریتم را افزایش داده و گاهی آن را کاهش می‌دهد. در توجیه این نتایج باید گفت که از آنجا که مهم‌ترین عامل در زمینه سرعت الگوریتم ضرب ماتریس‌های تنک در بردار سرعت دسترسی به حافظه است، صرف‌نظر از تعداد هسته‌های پردازنده به کار رفته در اجرای الگوریتم، همواره یک حد بالا برای سرعت وجود دارد که باید هزینه‌ی نسبتاً زیاد ایجاد و از بین بردن رشته‌های اجرایی مورد استفاده در این روش را نیز بدان افزود که ممکن است چنین نتایجی را به دنبال داشته باشد. در ستون‌های بعد کارایی الگوریتم ساده ضرب ماتریس‌های تنک در بردار روی پردازنده‌های گرافیکی شکل (۶) مورد بررسی قرار گرفته است. هسته به کار رفته برای استخراج این نتایج دقیقاً همان هسته ضرب ماتریس در بردار مورد استفاده در کتابخانه توابع وینا سی ال [۸] است و بنابراین می‌تواند براوردی از کارایی نسبی روش ارائه شده به‌دست دهد. دیده می‌شود این الگوریتم با وجود سادگی و عدم استفاده بهینه از منابع موجود توانسته است سرعت قابل قبولی در مقایسه با دو مورد قبلی کسب کند. ستون‌های آخر نیز کارایی روش پیشنهادی در این تحقیق را نشان می‌دهد. همان‌طور که دیده می‌شود کارایی روش بسیار بالاتر از تمامی موارد قبل بوده و در برخی موارد نزدیک ۲۰ برابر سریع‌تر از الگوریتم مشابه در پردازنده عمل کرده است. باید در نظر داشت هر چند مقادیر گزارش شده برای زمان اجرا با تغییر پارامترها و انتخاب بهترین مورد به دست آمده‌اند، اما می‌توان

جدول ۵- زمان انجام عمل ضرب بر حسب میلی ثانیه و نسبت افزایش سرعت در سیستم شماره ۲

ماتریس	CPU	CPU (OpenMP)		GPU (Naïve)		GPU (Proposed)	
	زمان اجرا	زمان اجرا	سرعت نسبی	زمان اجرا	سرعت نسبی	زمان اجرا	سرعت نسبی
Dense	۸/۳۲	۱۶/۰۰	۰/۵۲	۲/۶۶	۳/۱۳	۰/۴۲	۱۹/۶۲
Protein	۹/۰۰	۹/۳۲	۰/۹۷	۶/۷۴	۱/۳۴	۰/۶۹	۱۲/۹۸
FEM / Spheres	۱۲/۹۱	۱۱/۱۶	۱/۱۶	۹/۵۰	۱/۳۶	۱/۰۸	۱۱/۹۷
FEM / Cantilever	۸/۵۹	۷/۴۲	۱/۱۶	۶/۳۸	۱/۳۵	۰/۸۴	۱۰/۲۹
Wind Tunnel	۲۳/۶۵	۲۱/۰۲	۱/۱۳	۱۷/۴۶	۱/۳۵	۲/۰۶	۱۱/۴۸
FEM / Harbor	۵/۴۱	۴/۸۰	۱/۱۳	۳/۲۵	۱/۶۶	۰/۵۳	۱۰/۲۰
QCD	۴/۳۱	۴/۰۲	۱/۰۷	۲/۷۵	۱/۵۷	۰/۴۷	۹/۱۳
FEM / Ship	۱۶/۱۷	۱۴/۵۳	۱/۱۱	۱۱/۴۰	۱/۴۲	۱/۴۱	۱۱/۴۸
Economics	۶/۵۴	۳/۶۶	۱/۷۸	۱/۱۵	۵/۶۹	۰/۵۸	۱۱/۳۱
Epidemiology	۶/۹۳	۶/۴۸	۱/۰۷	۰/۹۱	۷/۶۰	۰/۴۹	۱۴/۰۸
FEM / Accelerator	۹/۸۰	۹/۸۷	۰/۹۹	۳/۷۳	۲/۶۳	۰/۷۹	۱۲/۴۳
Circuit	۵/۵۲	۳/۵۶	۱/۵۵	۱/۳۳	۴/۱۶	۱/۱۷	۴/۷۴
Webbase	۱۵/۲۸	۱۲/۳۶	۱/۲۴	۱۰/۱۴	۱/۵۱	۶/۵۲	۲/۳۴
LP	۶۴/۲۲	۵۹/۵۰	۱/۰۸	۵۱/۳۳	۱/۲۵	۳/۸۲	۱۶/۸۲

جدول ۶- تاثیر پارامترها در زمان اجرای الگوریتم بر حسب میلی ثانیه

تعداد سطر اختصاص‌یافته به هر گروه کاری / تعداد واحد محاسباتی در هر گروه کاری	۱	۲	۴	۸	۱۶	۳۲	۶۴	۱۲۸	۲۵۶
۱	۲۴/۲۸	-	-	-	-	-	-	-	-
۲	۱۴/۰۶	۱۳/۱۶	-	-	-	-	-	-	-
۴	۸/۳۰	۷/۵۵	۷/۲۱	-	-	-	-	-	-
۸	۵/۳۹	۴/۵۴	۴/۱۹	۴/۱۸	-	-	-	-	-
۱۶	۳/۸۳	۲/۹۷	۲/۵۸	۲/۵۲	۳/۲۲	-	-	-	-
۳۲	۲/۷۵	۲/۰۵	۱/۶۸	۱/۵۴	۲/۱۷	۷/۶۸	-	-	-
۶۴	۲/۵۷	۱/۵۲	۱/۲۵	۱/۱۸	۱/۶۵	۴/۳۳	۹/۲۸	-	-
۱۲۸	۴/۸۶	۲/۲۳	۱/۲۳	۱/۰۸	۱/۳۱	۲/۴۹	۴/۹۱	۹/۵۳	-
۲۵۶	۸/۷۰	۵/۳۱	۲/۲۶	۱/۲۶	۱/۳۱	۱/۳۲	۲/۴۲	۴/۸۱	۹/۵۰

بر اساس خصوصیات ماتریس نیز مقادیر بهینه و یا نزدیک به بهینه برای هر ماتریس را انتخاب کرد. شبیه این کار در برخی از تحقیقات دیگر انجام گرفته و می‌تواند به عنوان مکمل این تحقیق مورد نظر قرار گیرد. همچنین اجرای الگوریتم به دفعات نشان می‌دهد که پارامترهای بهینه تنها به نوع ماتریس بستگی داشته و در اجراهای مختلف ثابت هستند.

برای بررسی میزان تاثیر این پارامترها در زمان اجرای الگوریتم، ماتریس FEM / Spheres روی سیستم شماره (۲) در نظر گرفته شده و زمان اجرای تمامی حالات ممکن پارامترها در جدول (۶) نمایش داده شده است. مشاهده می‌شود هر چند استفاده از پارامترها تاثیرات بسیار مهمی در سرعت اجرای الگوریتم دارند، اما در نزدیکی پارامتر بهینه (که در جدول با خط زیر نشان داده شده است)، حساسیت نسبت به پارامترها نسبتاً پایین است و می‌توان امیدوار بود در صورتی که امکان بهینه‌سازی سرعت اجرا با زمان‌گیری وجود نداشته باشد، انتخاب نسبتاً خوب پارامترهای بهینه منجر به نتایج نسبتاً مناسبی شود.

۷- نتیجه‌گیری

در این تحقیق یک هسته ضرب ماتریس‌های تنک در بردار توسط زبان آزاد محاسباتی روی پردازنده‌های گرافیکی پیاده‌سازی شد. روش پیشنهادی از قالب استاندارد سطر تنک فشرده استفاده می‌کند که این مورد موجب سادگی استفاده از آن در برنامه‌های قبلی می‌شود. نتایج حاصله از این روش با حل مرجع روی پردازنده، استفاده از استاندارد باز چندپردازنده و نیز پیاده‌سازی ساده الگوریتم روی پردازنده‌های گرافیکی (مشابه با کتابخانه توابع وینا سی ال) روی یک طیف گسترده از ماتریس‌های تنک مقایسه شد. نتایج به‌دست آمده نشان می‌دهد روش پیشنهادی از کارایی بسیار بالایی برخوردار بوده و می‌تواند تمامی روش‌های گفته شده را به راحتی و با فاصله زیاد پشت سر بگذارد

واژه‌نامه

1. direct methods
2. Cholesky
3. iterative methods
4. conjugate gradient
5. bi-conjugate gradient
6. Krylov sub-space methods
7. sparse
8. compressed sparse row (csr)
9. cache
10. cache miss
11. high performance computing (hpc)
12. distributed memory
13. compute node
14. local area network (lan)
15. shared memory
16. multi-core
17. computational co-processor
18. computational accelerator
19. graphics processing unit (gpu)
20. open graphics library (opengl)
21. cuda
22. NVIDIA
23. Apple
24. open computing language (opencl)
25. open standard
26. many-core
27. multi-processor
28. execution thread
29. main memory
30. data bus
31. constant memory
32. local memory
33. private memory
34. intermediate products
35. primitive operation
36. reduction operation
37. kernel
38. work group
39. pre-processor macros
40. STREAM
41. open multi-processing (openmp)

مراجع

1. Saad, Y., *Numerical Methods for Large Eigenvalue Problems,* Manchester University Press, 1992.
2. Pasenau, M. and Jiménez, A. F., "Implementación de Algoritmos Numéricos en una Tarjeta Gráfica," *Centro Internacional de Métodos Numéricos en Ingeniería* (CIMNE), 2006.
3. Krüger, J. and Westermann, R., "Linear Algebra Operators for GPU Implementation of Numerical Algorithms", *Proceeding SIGGRAPH '03*, pp. 908-916, ACM, New York, USA, 2003.

4. NVIDIA CUDA, http://www.nvidia.com/object/cuda_home_new.html
5. Humphrey, J. R., Price, D. K., Spagnoli, K. E., Paolini, A. L. and Kelmelis, E. J., "CULA: Hybrid GPU Accelerated Linear Algebra Routines," *Proceedings SPIE 7705, Modeling and Simulation for Defense Systems and Applications V*, 770502, 2010.
6. OpenCL, http://www.khronos.org/opencl.
7. Fang, J., Varbanescu, A. L., and Sips, H, "A Comprehensive Performance Comparison of CUDA and OpenCL," *Proceedings International Conference on Parallel Processing*, pp. 216-225, 2011.
8. ViennaCL Project Homepage, http://viennacl.sourceforge.net.
9. Bell, N., and Garland, M., Efficient Sparse Matrix-Vector Multiplication on CUDA, NVIDIA Technical Report NVR-2008-004, NVIDIA, 2008.
10. Wu, T., Wang, B., Shan, Y., Yan, F., Wang, Y. and Xu, N., "Efficient PageRank and SpMV Computation on AMD GPUs", *Proceedings 39th International Conference on Parallel Processing (ICPP)*, pp. 81-89, 2010.
11. Mehri Dehnavi, M., Fernández, D. M., and Giannacopoulos, D., "Finite-Element Sparse Matrix Vector Multiplication on Graphic Processing Units", *IEEE Transactions on Magnetics*, Vol. 46, No. 8, 2010.
12. El Zein, A. H., and Rendell, A. P., "Generating Optimal CUDA Sparse Matrix-Vector Product Implementations for Evolving GPU Hardware," *Concurrency and Computation: Practice and Experience*, Vol. 24, pp. 3-13, 2012.
13. Grewe, D., and Lokhmotov, A., "Automatically Generating and Tuning GPU Code for Sparse Matrix-Vector Multiplication from a High-Level Representation", *Proceedings the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU4)*, 2011.
14. AMD Accelerated Parallel Processing OpenCL, AMD, 2011.
15. NVIDIA CUDA C Programming Guide, NVIDIA, 2012.
16. McCalpin, J. D., "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19-25, 1995.
17. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J., "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," *Proceedings the 2007 ACM/IEEE conference on Supercomputing*, ACM New York, NY, USA, 2007.