یک الگوریتم جستجوی اول سطح کارامد گراف بر روی CPU و GPU

پریسا کشاورزی، حسین دلداری و سعید ابریشمی

چکیده: گراف‌ها نمایش داده قدرتمندی هستند که به طور گسترده در حوزه‌های متفاوتی مورد استفاده قرار می‌گیرند. در کاربردهای مبتنی بر گراف یک پیمایش قاعده‌دار از گراف مانند جستجوی اول سطح، غالباً جزء کلیدی در پردازش مجموعه داده‌های بزرگ است. در این مقاله یک روش ترکیبی ارائه شده که برای هر سطح از پیمایش گراف، بهینه‌ترین نسخه از الگوریتم‌های پیاده‌سازی شده بر روی پردازنده مرکزی و پردازنده گرافیکی را انتخاب می‌کند. این روش ترکیبی کارایی خوبی را برای هر اندازه گرافی فراهم می‌کند، در حالی که از کارایی ضعیف روی گراف‌های با میانگین درجه کم و زیاد جلوگیری می‌کند. لازم به ذکر است که این روش بهره سرعت بالاتری نسبت به کارهای پیشین ارائه می‌دهد و نتایج علمی به دست آمده این ادعا را تأیید می‌کنند.

***کلید واژه:* جستجوی اول سطح، پردازنده گرافیکی، پردازنده مرکزی، کرنل.**

۱- مقدمه

امروزه ایده استفاده از پردازنده گرافیکی برای محاسبات همه‌منظوره بسیار محبوب شده است. زمانی که از پردازنده گرافیکی برای حل مسایل مناسب استفاده می‌شود، کارایی به طور چشم‌گیری افزایش می‌یابد [۱]. نکته قابل توجه این است که کدهایی برای اجراشدن بر روی پردازنده گرافیکی مناسب هستند که در آنها نیاز بسیاری به موازی‌سازی داده وجود دارد و همچنین کد هم در زمان و هم در جریان کنترل و الگوی دسترسی باید منظم باشد. هر چند هنوز مسایلی وجود دارند که به محاسبات سریع نیاز دارند، اما طبیعت الگوریتم آنها نامنظم است و از آنجایی که پردازنده گرافیکی برای حل مسایل منظم مناسب است، توجه‌نکردن به معماری پردازنده گرافیکی، مدل برنامه‌نویسی کودا (CUDA) و ساختار الگوریتم در هنگام برنامه‌نویسی می‌تواند باعث افت شدید کارایی برنامه شود. در نتیجه این سؤال مطرح می‌شود که به چه شیوه می‌توان از قدرت پردازنده گرافیکی برای حل چنین مسایلی استفاده نمود. یک مثال مهم از چنین مسایلی، پیمایش گراف است.

گراف‌ها نمایش داده قدرتمندی هستند که به طور گسترده در حوزه‌های زیادی مانند آنالیز اطلاعات [۲]، رباتیک [۳]، تحلیل شبکه اجتماعی [۴] و سایر موارد از آنها استفاده می‌شود. به علت اندازه زیاد مجموعه داده‌ها در این کاربردها زمان پردازش آنها بسیار زیاد است. به علت این که تسریع موازی این کاربردها به شدت به طبیعت تصادفی الگوهای دسترسی به حافظه محدود شده است بنابراین با موازی‌سازی نمی‌توان زمان پردازش آنها را زیاد کاهش داد [۵].

جستجوی اول سطح یک الگوریتم گراف مهم است که به طور باقاعده گره‌های گراف را پیمایش می‌کند. جستجوی اول سطح به عنوان یکی از مهم‌ترین الگوریتم‌های گراف در نظر گرفته شده است زیرا به عنوان یک ساختمان بلاک برای بسیاری از الگوریتم‌های دیگر مانند تشخیص قطعات متصل [۶]، تشخیص ساختار جامعه [۷] و محاسبه حداکثر جریان استفاده می‌شود.

به علت چنین اهمیتی تحقیقات چشم‌گیری برای پیاده‌سازی کارامد یک الگوریتم جستجوی اول سطح موازی بر روی آرایه وسیعی از سیستم‌های محاسباتی صورت گرفته است. نتایج دو کار اخیر توجه ما را به خود جلب کرده است. یکی کار لیو و همکارانش [۸] که جستجوی اول سطح را با استفاده از صف بر روی پردازنده گرافیکی حل می‌کند. با این شیوه می‌توان گراف‌های با میانگین درجه پایین را با کارایی مناسبی پیمایش کرد. دیگری کار هانگ و همکارانش [۹] است که برای پیاده‌سازی جستجوی اول سطح از پردازنده مرکزی در کنار پردازنده گرافیکی استفاده می‌کنند تا بتوانند بدین طریق گراف‌های بزرگ و کوچک را با سرعت مناسبی پیمایش کنند و از کارایی ضعیف جلوگیری کنند.

در این مقاله ما ایده‌های خود را از دو کار قبل کسب کرده‌ایم و یک راه حل عمومی ارائه کرده‌ایم که هم از پردازنده مرکزی و هم از پردازنده گرافیکی استفاده می‌کند تا بدین ترتیب از کارایی ضعیف روی گراف‌های بزرگ و کوچک با هر میانگین درجه‌ای جلوگیری کنیم و همچنین تا جایی که امکان دارد استفاده از پردازنده مرکزی را کاهش دهیم.

قبل از معرفی الگوریتم پیشنهادی ابتدا در بخش ۲ اصول برنامه‌نویسی بر روی پردازنده گرافیکی بیان می‌شود و سپس در بخش ۳ کارهای پیشین به صورت مختصر شرح داده می‌شوند. در بخش‌های ۴ و ۵ به ترتیب الگوریتم پیشنهادی و نتایج آن ارائه خواهند شد. در نهایت در بخش ۶ نتیجه‌گیری ارائه می‌شود.

۲- معماری پردازنده گرافیکی

یک پردازنده گرافیکی سازگار با کودا از آرایه‌ای از چندپردازنده‌های جریانی[1] سریع تشکیل شده است. هر دو چندپردازنده جریانی، یک ساختمان بلاک را تشکیل می‌دهند، هرچند در نسل‌های دیگر تعداد چندپردازنده‌های جریانی در این بلاک‌های ساختمانی می‌تواند فرق داشته باشد. همچنین هر چندپردازنده جریانی دارای تعدادی هسته[2] است که دارای حافظه نهان دستور مشترکی هستند. هر پردازنده گرافیکی دارای یک DRAM است که به آن حافظه سراسری گفته می‌شود.

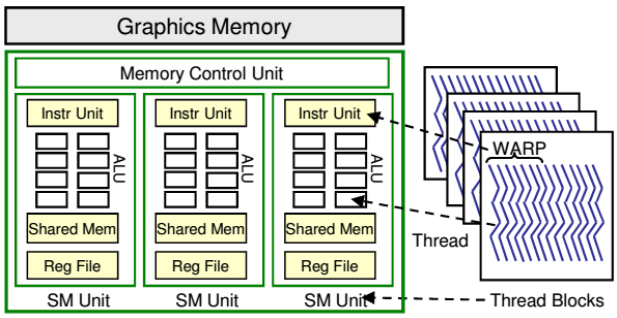
این مقاله در تاریخ ۲۵ بهمن ماه ۱۳۹۲ دریافت و در تاریخ ۲۱ آبان ماه ۱۳۹۳ بازنگری شد.

پریسا کشاورزی، دانشکده مهندسی کامپیوتر، دانشگاه آزاد اسلامی واحد مشهد، مشهد، (email: parisa.keshavarzi@yahoo.com).

حسین دلداری، دانشکده مهندسی کامپیوتر، دانشگاه آزاد اسلامی واحد مشهد، مشهد، (email: hdeldari@yahoo.com).

سعید ابریشمی، دانشکده مهندسی کامپیوتر، دانشگاه فردوسی، مشهد، (email: s-abrishami@um.ac.ir).

1. Streaming Multiprocessor
2. Core



شکل ۱: معماری یک پردازنده گرافیکی و مدل اجرای نخ در کودا.

روش مدیریت اجرای کودا تقسیم گروه‌هایی از نخ‌ها[1] در میان بلاک‌ها است. گریدها از مجموعه‌ای از بلاک‌ها[2] تشکیل شده‌اند. نخ‌ها به گروه‌های ۳۲بیتی که کلاف[3] نامیده می‌شود گروه‌بندی می‌شوند و به هسته‌ها نگاشت می‌شوند. یک کلاف واحد پایه‌ای برای زمان‌بندی است. همه ۳۲ نخ یک کلاف باید دستور یکسانی را اجرا کنند اما داده‌های آنها می‌تواند متفاوت باشد. هر نخ به یک هسته نگاشت می‌شود. در شکل ۱ معماری یک پردازنده گرافیکی و مدل اجرای نخ در کودا به تصویر کشیده شده است.

شش دسته فضای حافظه‌ای متداول در پردازنده‌های گرافیکی وجود دارد: ۱) حافظه ثباتی، ۲) حافظه محلی، ۳) حافظه مشترک، ۴) حافظه داده، ۵) حافظه ثابت و ۶) حافظه بافت.

حافظه ثباتی و حافظه محلی حاوی متغیرهای خصوصی هر نخ هستند و تنها خود نخ می‌تواند به آنها دسترسی پیدا کند. حافظه ثباتی سریع‌ترین نوع حافظه است که در ثبات پرونده نگهداری می‌شود. حافظه محلی برای نگهداری ثبات‌هایی که به علت محدودیت حجم ثبات پرونده باید خارج از آن نگهداری شوند به کار می‌رود. حافظه محلی عموماً در DRAM خارج از تراشه ذخیره می‌شود. این دو حافظه از نوع خواندنی/ نوشتنی هستند.

حافظه مشترک برای هر بلوک تعریف شده و نخ‌های هر بلوک تنها می‌توانند به حافظه مشترک بلوک خود دسترسی پیدا کنند. این حافظه از نوع خواندنی/ نوشتنی است. این حافظه کندتر از ثبات پرونده و سریع‌تر از DRAM خارج از تراشه است.

حافظه داده، ثابت و بافت بین تمامی نخ‌های گرید قابل رؤیت هستند و همگی به آنها دسترسی دارند. حافظه داده، ثابت و بافت حافظه سراسری نیز نام دارند. حافظه داده، ثابت و بافت در تراشه DRAM که خارج از پردازنده گرافیکی است ذخیره می‌شوند و هر یک بر روی سلسله مراتب جداگانه‌ای از تراشه هستند. برای دسترسی به حافظه اشتراکی یک پردازنده گرافیکی مدل GTX۲۸۰ تنها به دو سیکل ساعت نیاز است در حالی که زمان دسترسی به حافظه سراسری این پردازنده تقریباً ۳۰۰ سیکل ساعت است. در نتیجه باید در دسترسی به این حافظه‌ها دقت کرد.

یک برنامه کودا شامل یک یا چند مرحله است که روی میزبان یا یک دستگاه اجرا می‌شود. منظور از میزبان پردازنده مرکزی و دستگاه همان پردازنده گرافیکی است. یک برنامه کودا یک کد یک‌پارچه است که شامل کد میزبان و دستگاه است. به کد دستگاه کرنل[4] نیز گفته می‌شود و از این پس هر جا صحبت از کرنل شد منظور کد دستگاه است.

اجرای برنامه با اجرای کد میزبان آغاز می‌شود. وقتی که یک تابع کرنل فراخوانی می‌شود فرایند اجرا به دستگاه سپرده می‌شود. زمانی که کرنل فراخوانی می‌شود تعداد زیادی نخ تولید می‌شود. همه نخ‌های ایجادشده در طی فراخوانی یک کرنل، گرید نامیده می‌شوند. زمانی که همه نخ‌های یک کرنل اجرای خود را کامل کنند گرید مربوط خاتمه می‌یابد و اجرا روی میزبان ادامه می‌یابد تا زمانی که کرنل دیگری فراخوانی شود. برای اطلاعات بیشتر در این زمینه می‌توانید به [۱۰] مراجعه کنید.

۳- مطالعه پیشین

برای اولین بار الگوریتم جستجوی اول سطح توسط هریش و همکارانش در سال ۲۰۰۷ بر روی پردازنده گرافیکی پیاده‌سازی شد [۱۱]. در ادامه این الگوریتم با نام الگوریتم پایه شناخته می‌شود. در پیاده‌سازی این الگوریتم از شیوه برنامه‌نویسی مشابه به PRAM استفاده شده است. بدین صورت که هر گره به یک نخ تخصیص می‌یابد و در صورتی که آن گره به سطح جاری تعلق داشته باشد، آن نخ همسایه‌های ملاقات‌نشده آن گره را به روز رسانی خواهد کرد. با این الگوریتم می‌توان گراف‌های بزرگ و متراکم یا گراف‌هایی که تعداد سطوح کمی دارند را به خوبی پیمایش کرد. هریش و همکارانش از روش همگامی ضمنی پردازنده مرکزی برای همگام‌کردن بلاک‌های نخ هر سطح کرنل استفاده کرده‌اند و بین نخ‌های کرنل امکان عدم تعادل بار وجود دارد و الحاق حافظه[5] نیز در این الگوریتم تضمین نمی‌شود.

در سال ۲۰۱۰ لیو و همکارانش از صف و روش برنامه‌نویسی PRAM برای پیاده‌سازی الگوریتم جستجوی اول سطح استفاده کردند. به دلیل استفاده از توابع اتمی برای پیاده‌سازی صف بر روی پردازنده گرافیکی، سربار زیادی به پردازنده گرافیکی تحمیل می‌شود. به همین دلیل در پیاده‌سازی [۱۱] از صف استفاده نشده است. برای این که سربار پیاده‌سازی صف کاهش یابد لیو و همکارانش از دو سطح حافظه سراسری و اشتراکی با توجه به تعداد گره‌هایی که پردازش می‌شوند استفاده کرده‌اند. الگوریتم آنها برای پیمایش گراف‌های منظم مناسب است. به علت پیاده‌سازی مناسب صف، الحاق حافظه در این الگوریتم تضمین می‌شود اما هنوز هم امکان عدم تعادل بار بین نخ‌های کرنل وجود دارد. برای کاهش سربار همگامی بلاک‌های نخ در هر سطح، در این الگوریتم سه کرنل ارائه شده است که با توجه به تعداد گره‌های سطح جاری از روش‌های همگامی مناسب در هر کرنل استفاده شده است.

در دو کار [۸] و [۱۱] با توجه به داده‌های هر نخ ممکن است که نخ‌های موجود در یک کلاف مسیر اجرایی متفاوتی را دنبال کنند، بنابراین این الگوریتم‌ها با واگرایی مسیر اجرا[6] نیز روبرو هستند.

هانگ و همکارانش برای این که بتوانند گراف‌های خلوت و متراکم را در زمان مناسبی پیمایش کنند در کنار پردازنده گرافیکی از پردازنده مرکزی استفاده کرده‌اند [۹]. روش کار آنها به این صورت است که زمانی که تعداد گره‌های سطح جاری گراف کم باشد از پردازنده مرکزی و در غیر این صورت از پردازنده گرافیکی برای پیمایش گراف استفاده می‌کنند. آنها از دو تابع که بر اساس روش‌های مبتنی بر صف و خواندن پیاده‌سازی شده‌اند برای پیمایش گراف بر روی پردازنده مرکزی استفاده کرده‌اند. لازم به ذکر است که آنها از شیوه برنامه‌نویسی مبتنی بر کلاف برای پیاده‌سازی الگوریتم بر روی پردازنده گرافیکی استفاده کرده‌اند. این روش به این

1. Threads
2. Blocks
3. Warp
4. Kernel
5. Memory Coalescing
6. Execution-Path Divergence

```
1:  Void CPU-GPUBFS ()
2:  {Curr=0; finished=false;
3:     Do{
4:     Finished=true;
5:     if (no_nodes < Blocksz)
6:        Function CPU-BFS(int N, int curr, int*level, int*nodes, int*edges, bool*finished)
7:     else if ((no_nodes > Blocksz) and (no_nodes <= No_SM × Blocksz))
8:        Quee-bfs-kernel (int*q, int N, int curr, int*level, int*nodes, int*edges, bool*finished)
9:     else if ((no_nodes > NUM_SM × Blocksz) and (mean_graph<5)
10:       Quee-bfs-mutli-block-kernel (int*q, int N, int curr, int*level, int*nodes, int*edges, bool*finished)
11:    else
12:       warp-bfs-kernel (int N, int curr, int*level, int*nodes, int*edges, bool*finished)
13:    while(!finished)
14: }
```

شکل ۲: شبه‌کد فراخوانی و کرنل‌های الگوریتم.

صورت است که به جای تخصیص هر گره به یک نخ، هر گره به یک کلاف تخصیص می‌یابد. در نتیجه هیچ‌گاه نخ‌های یک کلاف با واگرایی مسیر اجرا روبرو نمی‌شوند و به دلیل این که کل نخ‌های کلاف همسایه‌های یک گره را بررسی و در صورت نیاز آنها را به روز رسانی می‌کنند الحاق حافظه نیز تضمین می‌شود. در این الگوریتم از روش همگامی ضمنی پردازنده مرکزی برای همگامی بلاک‌های نخ هر سطح کرنل استفاده شده است.

در بخش بعد الگوریتم پیشنهادی به نحوی ارائه شده است که به طور پویا بهترین پیاده‌سازی را از الگوریتم‌های موازی پیاده‌سازی شده بر روی پردازنده مرکزی و پردازنده گرافیکی انتخاب می‌کند. برای کاهش زمان اجرای الگوریتم پیاده‌سازی شده بر روی پردازنده گرافیکی سلسله مراتبی از کرنل‌های ارائه شده استفاده گردیده است که هر کدام با شیوه برنامه‌نویسی و روش‌های همگامی مناسب با آن کرنل پیاده‌سازی شده‌اند. بدین ترتیب علاوه بر کاهش زمان اجرای الگوریتم تا حد ممکن از واگرایی مسیر اجرا و حافظه نیز جلوگیری می‌شود.

٤- الگوریتم پیشنهادی (CPU-GPUBFS)

بهره‌گیری صرف از پردازنده گرافیکی باعث می‌شود که تنها قادر به پیمایش گراف‌های با میانگین درجه کم یا زیاد باشیم. در نتیجه در این مقاله سعی می‌شود تا با بهره‌گیری از پردازنده مرکزی در کنار پردازنده گرافیکی و توجه به امکانات و شیوه‌های برنامه‌نویسی پردازنده گرافیکی الگوریتم جستجوی اول سطح به گونه‌ای توسعه داده شود که بتواند انواع گراف‌ها با هر میانگین درجه‌ای را با بهره سرعت خوبی پیمایش کند.

با توجه به مطالب ذکرشده برای پیاده‌سازی الگوریتم پیشنهادی یک تابع میزبان (برنامه‌ای که روی پردازنده مرکزی اجرا می‌شود) و سه کرنل طراحی شده و شبه‌کد این الگوریتم در شکل ۲ نشان داده شده است.

همان طورکه قبلاً ذکر شد برنامه‌هایی برای اجراشدن بر روی پردازنده گرافیکی مناسب هستند که در آنها نیاز بسیاری به موازی‌سازی داده وجود دارد. در نتیجه در الگوریتم طراحی‌شده، زمانی که تعداد گره‌های سطح جاری گراف کمتر از تعداد نخ‌های یک بلاک (در سیستم مورد استفاده ۵۱۲ نخ) از پردازنده گرافیکی باشد مانند الگوریتم هانگ و همکارانش از پردازنده مرکزی برای اجرای الگوریتم بهره گرفته شده است. لازم به ذکر است که از روش مبتنی بر صف برای پیاده‌سازی این تابع استفاده شده است زیرا تعداد گره‌هایی که قرار است در این تابع پردازش شوند کم است و بهینه نیست که همه گره‌ها بررسی شوند تا متوجه شویم کدام یک از آنها به سطح جاری تعلق دارند. اما زمانی که تعداد گره‌های سطح جاری گراف بیشتر از ۵۱۲ باشد به جای پردازش آن سطح گراف بر روی پردازنده مرکزی از پردازنده گرافیکی برای پیمایش آن سطح استفاده می‌شود تا بتوان زمان اجرای الگوریتم را نسبت به الگوریتم هانگ کاهش داد. زمانی که تعداد گره‌های سطح جاری بیش از تعداد نخ‌های یک بلاک و کمتر از مقدار (۱) باشد کرنل Quee-bfs-in-mutli-block-kernel فراخوانی می‌شود

(۱) تعداد چندپردازنده‌های جریانی × تعداد نخ‌های یک بلاک

اما باید به این نکته توجه کرد که عدم استفاده از صف در این کرنل باعث افزایش زمان اجرای الگوریتم می‌شود. زیرا در هر سطح مجبوریم تمام گره‌ها را بررسی کنیم تا متوجه شویم کدام یک از آنها به سطح جاری تعلق دارند. اگر تعداد گره‌های سطوح بعدی کمتر از مقدار (۱) و بیشتر از تعداد نخ‌های یک بلاک باشد کنترل اجرا در این کرنل باقی خواهد ماند. کودا هیچ تابعی برای همگامی بلاک‌های نخ مختلف با یکدیگر ارائه نکرده است. در نتیجه اکثر الگوریتم‌های پیاده‌سازی شده بر روی پردازنده گرافیکی برای همگامی بلاک‌های نخ با یکدیگر از روش همگامی صریح پردازنده مرکزی استفاده می‌کنند. اما در این مقاله برای کاهش زمان همگامی این کرنل از تابع همگامی مبتنی بر قفل شیائو [۱۲] استفاده شده است. این روش بدین صورت است که یک متغیر mutex سراسری تعریف می‌شود که تعداد بلاک‌های نخی که می‌خواهند همگام شوند را می‌شمارد. یک نخ از هر بلاک متغیر mutex را یک واحد افزایش می‌دهد و زمانی که مقدار این متغیر برابر تعداد بلاک‌های کرنل شد همگامی کامل شده و بلاک‌ها می‌توانند اجرای خود را ادامه دهند. از آنجایی که از روش برنامه‌نویسی مشابه به PRAM برای پیاده‌سازی این کرنل استفاده شده است امکان عدم تعادل بار بین نخ‌ها در زمان پردازش گراف‌های نامنظم وجود دارد. برای کاهش این عدم تعادل بار از شیوه شناسایی رأس‌های با درجه بالا استفاده شده است. این شیوه بدین صورت است که زمانی که تعداد همسایه‌های گره‌هایی که باید پردازش شوند بسیار بیشتر از تعداد همسایه‌های گره‌های دیگر باشند، این گره‌ها در یک صف سراسری قرار می‌گیرد تا در پایان اجرای کرنل جاری به طور مجزا پردازش شوند. اما زمانی که میانگین درجه گراف کمتر از ۵ باشد (طبق آزمایشات صورت‌گرفته این میانگین درجه گراف ۵ در نظر گرفته شده است) و تعداد گره‌های سطح جاری از مقدار (۱) بیشتر باشد کرنل Queeu-bfs-out-mutli-block-kernel فراخوانی خواهد شد. از آنجایی که میانگین درجه گراف در هنگام فراخوانی این کرنل کم است می‌توان از سربار پیاده‌سازی صف در این کرنل چشم‌پوشی کرد و از صف برای نگهداری گره‌های مرز استفاده کرد. در صورتی که از صف در این کرنل استفاده نشود از آنجایی که میانگین درجه گراف کم است و ممکن است گراف بسیار بزرگ باشد باید تمام گره‌های گراف بررسی شوند تا متوجه شویم که کدام یک از آنها به سطح جاری تعلق دارند. در نتیجه سربار پیاده‌سازی صف نسبت به زمان بررسی گره‌های متعلق به سطح جاری

```
15: Void Quee-bfs-in-mutli-block-kernel (int*q, int N, int curr,
    int*level, int*nodes, int*edges, bool*finished)
16: {
17:     Int tid=blockIdx.x×Max_threads_per_block+threadIdx.x;
18:     If (tid<N)
19:     {
20:         While (q is not empty & num_nodes(q)
            <(Max_threads_per_block×NUM_SM) & stay)
21:         {
22:             Pid=atomicor((int *)&q[tid],0);
23:             For (int i=nodes[pid]; i<nodes[pid+1]; i++)
24:             {
25:                 Int id=edges[i];
26:                 If(level[id]==INF)
27:                 {
28:                     level[id]= level[pid]+1;
29:                     Push these nodes to local q
30:         }}}
31:         __syncthreads();
32:         Remove visited vertex from q;
33:         If (local q <> null){
34:             Copy local q to q
35:             Stay=true;
36:             Finished=false;
37:         }
38:         Strart_global_barrier();
39: }}
```

شکل ۳: شبه‌کد کرنل Quee-bfs-in-mutli-block-kernel.

```
40: Void Queeu-bfs-out-mutli-block-kernel (int*q, int N, int curr,
    int*level, int*nodes, int*edges, bool*finished)
41: {
42:     Int tid=blockIdx.x×Max_threads_per_block+threadIdx.x;
43:     If (tid<N)
44:     {
45:         While (q is not empty)
46:         {
47:             Pid=atomicor((int *)&q[tid],0);
48:             For (int i=nodes[pid]; i<nodes[pid+1]; i++)
49:             {
50:                 Int id=edges[i];
51:                 If(level[id]==INF)
52:                 {
53:                     level[id]=level[pid]+1;
54:                     Push these nodes to local q
55:         }}}
56:         __syncthreads();
57:         Remove visited vertex from q;
58:         If (local_q<>NULL)
59:         {
60:             Copy local q to q
61:             Finished=false;
62: }}}
```

شکل ۴: شبه‌کد کرنل Queeu-bfs-out-mutli-block-kernel.

بسیار کمتر است. مانند کرنل Quee-bfs-in-mutli-block-kernel از روش برنامه‌نویسی مشابه به PRAM برای پیاده‌سازی این کرنل بهره گرفته شده و برای کاهش عدم تعادل بار بین نخ‌ها از شیوه شناسایی رأس‌های با درجه بالا استفاده شده است.

در صورتی که میانگین درجه گراف بیشتر از ۴ و تعداد گره‌های سطح جاری از مقدار (۱) بیشتر باشد کرنل warp-bfs-kernel فراخوانی خواهد شد. از آنجایی که بیشتر گره‌های گراف به این سطح تعلق دارند در نتیجه از صف برای نگهداری گره‌های مرز استفاده نشده تا بدین ترتیب بتوان زمان اجرای الگوریتم را کاهش داد. برای نگهداری گراف از دو آرایه استفاده شده که آرایه اول گره‌های گراف را و آرایه دوم همسایه‌های هر گره را نگه می‌دارد. اگر تعداد گره‌های سطوح بعدی از آستانه تعریف‌شده کمتر شوند باز هم کنترل اجرا در این کرنل باقی خواهد ماند تا زمانی که کل گراف پیمایش شود. زیرا برای انتقال کنترل اجرا به کرنل‌های دیگر

```
63: Void warp-bfs-kernel (int N, int curr, int*levels, int*nodes,
    int*edges, bool*finished)
64: {
65:     Int w-off=thread_id % w_sz;
66:     Int w-id=thread_id/w_sz;
67:     Int num_warps=num_threads/w_sz;
68:     Extern__shared__warp_mem smem[];
69:     Warp_mem_t*my=smem+(local_thread_id/w_sz);
70:
71:     //copy my work to local
72:     Int v_=w_id×warp_sz;
73:     Memcpy_simd<w_sz>(w_off, warp_sz, my->levels, &level[v]);
74:     Memcpy_simd<w_sz>(w_off, warp_sz+1, my->nodes,
        &nodes[v]);
75:
76:     //iterate over my work
77:     For (int v=0; v<warp_sz; v++){
78:         If (my->levels[v]==curr){
79:             Int num_nbr=my->nodes[v+1]–my->nodes[v];
80:             Int*nbrs= &edges[my->nodes[v]];
81:             Expands_bfs_simd<w_sz>(w_off, num_nbr, nbrs, levels,
            curr, finished);
82: }}}
```

شکل ۵: شبه‌کد کرنل warp-bfs-kernel.

باید تعداد گره‌های سطح بعدی شمارش و در یک صف قرار گیرند که خود عملیات پرهزینه‌ای است. از روش برنامه‌نویسی مبتنی بر کلاف برای پیاده‌سازی این کرنل بهره گرفته شده است. در این روش هر گره به یک کلاف تخصیص می‌یابد و هر نخ کلاف تنها یک همسایه آن گره را به روز رسانی می‌کند. در نتیجه تعادل بار بین نخ‌های کرنل وجود دارد. لازم به ذکر است که ترتیب خاصی برای اجرای نخ‌های یک کلاف در میان هشت هسته پردازنده گرافیکی وجود ندارد و ابتدا همسایه‌های با شماره پایین‌تر به روز رسانی می‌شوند. از آنجایی که در کرنل Queeu-bfs-out-mutli-block-kernel و warp-bfs-kernel تعداد گره‌های سطح جاری بیش از مقدار (۱) است نمی‌توان از روش همگامی مبتنی بر قفل پردازنده گرافیکی برای همگامی بلاک‌های نخ کرنل با یکدیگر استفاده کرد. بنابراین برای این نوع همگامی مجبور به استفاده از روش همگامی صریح پردازنده مرکزی هستیم.

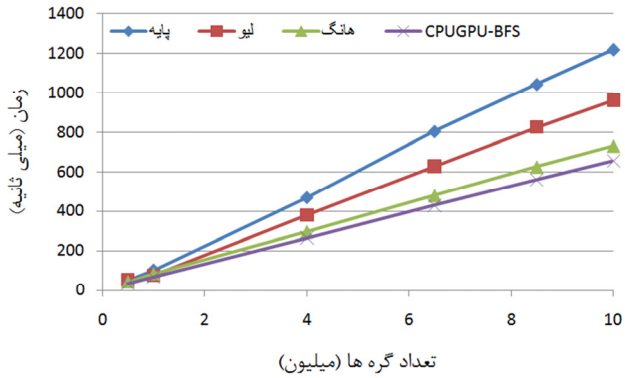
برای درک بیشتر الگوریتم شبه‌کد کرنل‌ها به ترتیب در شکل‌های ۳ تا ۵ نشان داده شده است.

۵- نتایج آزمایشگاهی

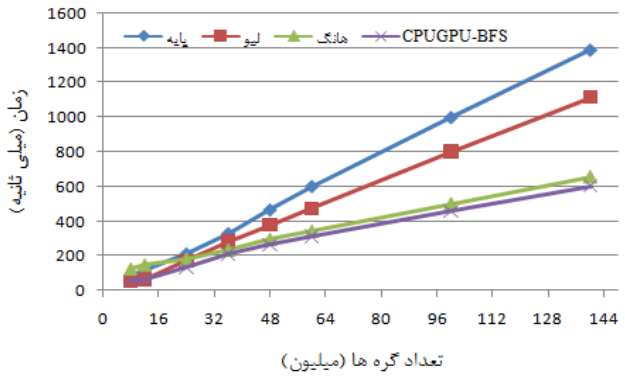
در این بخش نتایج حاصل از ارزیابی الگوریتم ارائه‌شده شرح داده می‌شود اما در ابتدا مشخصات ماشین و نرم‌افزارهای استفاده‌شده در آزمایش‌ها معرفی می‌شود. سپس انواع گرافی که آزمایش‌ها بر روی آنها انجام گرفته‌اند معرفی می‌شود. پس از آن زمان اجرای الگوریتم پیشنهادی با الگوریتم‌های هانگ، پایه و لیو مقایسه می‌شود.

در آزمایش‌ها از یک پردازنده اصلی Intel core i۵ با چهار گیگابایت حافظه استفاده شده است. برای پیاده‌سازی تابع الگوریتم، بر روی پردازنده مرکزی از openmp به همراه چهار نخ استفاده شده است. کارت گرافیکی استفاده شده NVIDIA GeForce ۳۱۰ M سازگار با کودا است. این پردازنده دارای دو چندپردازنده جریانی است که هر کدام هشت هسته دارند. پیاده‌سازی‌ها با استفاده از کودا ۳٫۱ [۱۳] و C++ بر روی ویندوز ۷ صورت گرفته‌اند.

الگوریتم‌های پیاده‌سازی شده با گراف‌های مصنوعی و جهان حقیقی (گراف‌های DIMACS [۱۴] و یک کلکسیون مجموعه داده بزرگ عمومی [۱۵]) آزمایش می‌شوند. قابل ذکر است که از کتابخانه گراف



شکل ۶: گراف‌های تصادفی یکنواخت (میانگین درجه گراف ثابت).



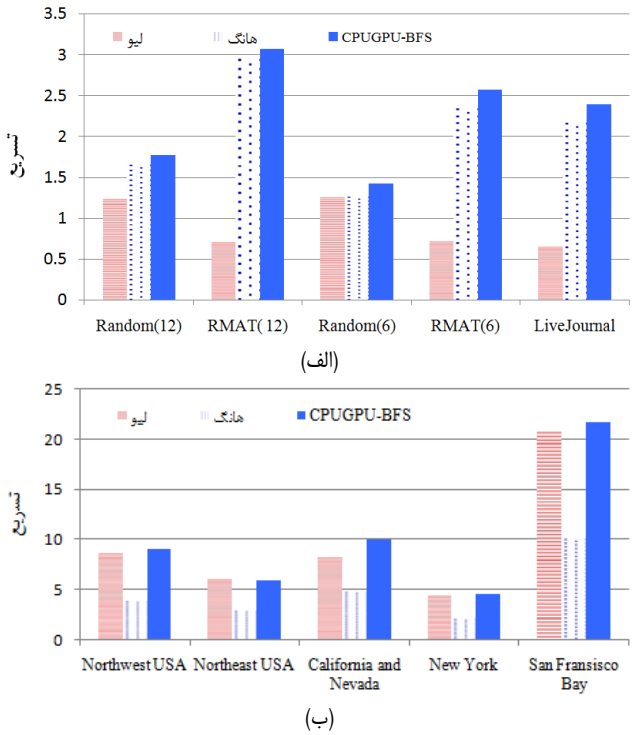
شکل ۷: گراف‌های تصادفی یکنواخت (میانگین درجه گراف متغیر).

SNAP برای تولید گراف‌های مصنوعی مدل تصادفی یکنواخت و RMAT استفاده شده است [۱۶].

در اولین آزمایش زمان اجرای الگوریتم ارائه‌شده برای گراف‌های تصادفی یکنواخت با میانگین درجه ثابت محاسبه و با زمان اجرا سه کار پیشین مقایسه شده است. در شکل ۶ نتایج این آزمایش نشان داده شده و میانگین درجه گراف در این آزمایش ۱۲ فرض شده است. محور افقی این شکل تعداد گره‌های گراف بر حسب میلیون و محور عمودی زمان اجرای الگوریتم بر حسب میلی‌ثانیه را نشان می‌دهد.

در آزمایش دوم زمان اجرا این الگوریتم برای گراف‌های تصادفی یکنواخت با میانگین درجه متغیر محاسبه شده و نتایج آن در شکل ۷ به تصویر کشیده شده است. تعداد گره‌های گراف ثابت (چهار میلیون) در نظر گرفته شده است. محور افقی تعداد لبه‌های گراف و محور عمودی زمان اجرای الگوریتم را بر حسب میلی‌ثانیه نشان می‌دهد. توجه داشته باشید که زمانی که تعداد لبه‌های گراف بیشتر از ۱۴۴ میلیون باشد گراف روی حافظه پردازنده گرافیکی جا نمی‌شود.

همان طور که مشاهده می‌کنید زمان اجرای الگوریتم ارائه‌شده از تمام الگوریتم‌های پیشین کمتر است و با زیادشدن تعداد گره‌ها این فاصله بیشتر هم می‌شود زیرا با بهره‌گیری از پردازنده مرکزی به جای پردازنده گرافیکی برای سطوحی که تعداد گره‌های آن کم است توانستیم زمان اجرای الگوریتم را برای آن سطوح کاهش دهیم. همچنین با استفاده از روش برنامه‌نویسی مبتنی بر کلاف و شیوه شناسایی رئوس با درجه بالا در پردازنده گرافیکی توانستیم تا حد ممکن از واگرایی مسیر اجرای الگوریتم جلوگیری کنیم و به دلیل استفاده از صف و شیوه برنامه‌نویسی مبتنی بر کلاف الحاق حافظه نیز تضمین می‌شود. به طور جزئی‌تر، به دلیل استفاده از صف در کرنل‌های Quee-bfs-in-mutli-block-kernel و Quee-bfs-out-mutli-block-kernel زمان محاسباتی الگوریتم کاهش

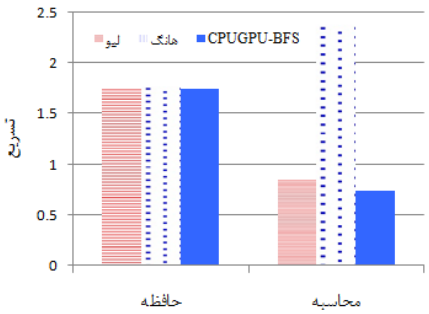


شکل ۸: بهره‌وری سرعت الگوریتم‌ها نسبت به الگوریتم پایه، (الف) گراف‌های مصنوعی و SNAP و (ب) گراف‌های DIMACS.

می‌یابد زیرا تنها به تعداد گره‌های سطح جاری از نخ‌های پردازنده گرافیکی استفاده می‌شود و لازم نیست در هر سطح همه گره‌ها بررسی شوند تا متوجه شویم کدام گره به سطح جاری تعلق دارد. از آنجا که تعداد گره‌های سطح جاری گراف در کرنل Quee-bfs-in-mutli-block-kernel کم است و در کرنل Quee-bfs-out-mutli-block-kernel نیز میانگین درجه گراف پایین است در نتیجه سربار پیاده‌سازی صف قابل چشم‌پوشی است. همچنین با طراحی سلسله‌مراتبی از کرنل‌ها و استفاده از شیوه‌های مناسب همگامی، زمان همگامی الگوریتم را نیز توانستیم نسبت به الگوریتم‌های پیشین کاهش دهیم.

در سومین آزمایش میزان بهره‌وری سرعت الگوریتم پیشنهادی و سایر الگوریتم‌ها نسبت به الگوریتم پایه محاسبه و در شکل ۸ نشان داده شده است. در گراف‌های Random و RMAT تعداد گره‌ها چهار میلیون و میانگین درجه گراف در کنار اسم هر گراف ذکر شده است. به عنوان مثال در Random(۱۲) میانگین درجه گراف ۱۲ در نظر گرفته شده است. همان طور که مشاهده می‌شود الگوریتم هانگ برای بیشتر گراف‌های DIMACS تسریع کمتری نسبت به سایر الگوریتم‌ها دارد زیرا این الگوریتم برای پیمایش گراف‌های متراکم مناسب است و میانگین درجه در این گراف‌ها تقریباً دو است. از آنجایی که الگوریتم لیو برای پیمایش گراف‌های خلوت و منظم مناسب است لذا این الگوریتم برای پیمایش گراف‌های RMAT، Random(۱۲) و LiveJournal بهره‌وری سرعت کمی دارد زیرا گراف‌های RMAT و LiveJournal نامنظم هستند و میانگین درجه در گراف Random(۱۲) زیاد است. در جدول ۱ تعداد گره‌ها و لبه‌های نمونه گراف‌های DIMACS و LiveJournal نشان داده شده است.

نهایتاً در آخرین آزمایش زمان محاسبات و انتقال داده‌ها از حافظه پردازنده مرکزی به پردازنده گرافیکی و بازگشت آن برای الگوریتم ارائه‌شده با سایر الگوریتم‌ها مقایسه شده که تعداد گره‌ها در این آزمایش ۳۰۰۰ و میانگین درجه گراف هشت در نظر گرفته شده است (شکل ۹).



شکل ۹: زمان محاسبات و انتقال داده‌ها از حافظه پردازنده مرکزی به پردازنده گرافیکی.

جدول ۱: ویژگی نمونه گراف‌های استفاده‌شده در آزمایش‌ها.

نام	تعداد گره‌ها	تعداد لبه‌ها
LiveJournal	۴۳۰۸۴۵۱	۶۸۹۹۳۷۷۳
Northwest USA	۱۲۰۷۹۴۵	۲۸۴۰۲۰۸
Northeast USA	۱۵۲۴۴۵۳	۳۸۹۷۶۳۶
California and Nevada	۱۸۹۰۸۱۵	۴۶۵۷۷۴۲
New York	۲۶۴۳۴۶	۷۳۳۸۴۶
San Francisco Bay	۳۲۱۲۷۰	۸۰۰۱۷۲

٦- نتیجه‌گیری

در این مقاله یک روش ترکیبی ارائه شد که به طور پویا در هر مرحله بهینه‌ترین نسخه از الگوریتم‌های پیاده‌سازی شده بر روی پردازنده مرکزی و پردازنده گرافیکی را انتخاب می‌کند. بدین ترتیب می‌توان هر نوع گراف با هر میانگین درجه‌ای را با بهره سرعت خوبی پیمایش کرد. در این کار برای کاهش زمان اجرای الگوریتم پیاده‌سازی شده بر روی پردازنده گرافیکی نسبت به سایر الگوریتم‌های پیشین سعی شد که با توجه به تعداد گره‌های هر سطح گراف از روش‌های برنامه‌نویسی و شیوه‌های همگامی مناسب آن سطح استفاده شود. از آنجا که زمانی که از شیوه برنامه‌نویسی مشابه به PRAM برای پیاده‌سازی کرنل‌ها استفاده می‌شود امکان عدم تعادل بار بین نخ‌ها وجود دارد برای کاهش عدم تعادل بار در این کرنل‌ها از شیوه شناسایی رأس‌های با درجه بالا استفاده گردیده است. با این شیوه از واگرایی مسیر اجرای الگوریتم جلوگیری شد. به دلیل استفاده از صف و شیوه برنامه‌نویسی مبتنی بر کلاف الحاق حافظه الگوریتم نیز تضمین شد. بهره‌وری سرعت الگوریتم ارائه‌شده نسبت به الگوریتم‌های پایه، لیو و هانگ به ترتیب ۱/۳۲x تا ۲۰/۸x، ۱x تا ۳/۸x و ۱/۰۹x تا ۱۰/۷۱x است.

مراجع

[1] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56-69, Mar./Apr. 2010.

[2] T. Coffman, S. Greenblatt, and S. Marcus, "Graph-based technologies for intelligence analysis," *Communications of the ACM*, vol. 47, no. 3, pp. 45-47, Mar. 2004.

[3] R. Sim and N. Roy, "Global a-optimal robot exploration in slam," in *Proc. IEEE Int. Conf. on, Robotics and Automation, ICRA'05,* vol. pp. 661-666, Barcelona, Spain, Apr. 2005.

[4] S. Heryrani Nobari, *Scalable Data-Parallel Graph Algorithms from Generation to Management*, Ph.D. Thesis, University of Singapore, 2012.

[5] U. Brandes, "A faster algorithm for betweenness centrality," *the J. of Mathematical Sociology*, vol. 25, no. 2, pp. 163-177, 2001.

[6] A. Lumsdaine, D. Gregor, B. Hendrickson, J. Berry, and J. W. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5-20, 2007.

[7] S. Skiena, *The Algorithm Design Manual*, Springer, pp. 166-168, 1998.

[8] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physiscal Review E*, vol. 69, no. 2, 16 pp., Feb. 2004.

[9] L. Luo, M. Wong, and W. M. Hwu, "An effective GPU implementation of breadth-first search," in *Proc. of the 47th Design Automation Conf.*, pp. 52-55, 2010.

[10] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Proc. of the 2011 Inte. Conf. on Parallel Architectures and Compilation Techniques*, pp. 78-88, 2011.

[11] https://developer.nvidia.com/about-cuda

[12] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *HiPC*, vol. 4873, Springer, Dec. 2007.

[13] S. Xiao and W. C. Feng, "Inter-block GPU communication via fast barrier synchronization," in *Proc. IEEE Int. Symp. on Parallel & Distributed Processing, IPDPS'10*, 12 pp., 2010.

[14] NVidia, *C Best Practices Guide*, NVIDIA Corp., Santa Clara, CA, US, 2012.

[15] http://www.dis.uniroma1.it/~challenge9/

[16] *Stanford Large Network Dataset Collection*, Feb. 2013, http://snap.stanford.edu/data

پریسا کشاورزی تحصیلات خود را در مقطع کاردانی کامپیوتر در سال ۱۳۸۵ از آموزشکده فنی و حرفه ای دختران شیراز و کارشناسی مهندسی کامپیوتر را در سال ۱۳۸۸ از موسسه آموزش عالی پاسارگاد شیراز و کارشناسی ارشد را در سال ۱۳۹۲ از دانشگاه آزاد اسلامی مشهد به پایان رسانده است و هم اکنون دبیر هنرستان شیراز می باشد. زمینه های تحقیقاتی مورد علاقه ایشان عبارتند از: الگوریتم‌های موازی، پردازش موازی، برنامه نویسی روی کارت گرافیک.

حسین دلداری در سال ۱۳۵۱ مدرک کارشناسی فیزیک خود را از دانشگاه فردوسی و کارشناسی ارشد را در سال ۱۳۵۷ از دانشگاه اورگان و دکتری در سال ۱۳۷۶ از دانشگاه لیبز دریافت نموده و هم اکنون دانشیار دانشکده فنی مهندسی دانشگاه آزاد اسلامی مشهد می باشد. نامبرده قبل از پیوستن به دانشگاه آزاد اسلامی واحد مشهد استادیار دانشگاه فردوسی بوده است. زمینه‌های تحقیقاتی مورد علاقه ایشان عبارتند از: الگوریتم‌های موازی، پردازش موازی، محاسبات نرم و کاربردهای آن، برنامه نویسی روی کارت گرافیک، شبکه های کامپیوتری و مهندسی نرم افزار.

سعید ابریشمی مدرک کارشناسی، کارشناسی ارشد و دکتری مهندسی کامپیوتر خود را به ترتیب در سال های ۱۳۷۶، ۱۳۸۰ و ۱۳۸۴ از دانشگاه فردوسی دریافت نمود. هم اکنون استادیار و مدیر گروه دانشکده فنی مهندسی دانشگاه فردوسی می باشد. زمینه‌های تحقیقاتی مورد علاقه ایشان عبارتند از: الگوریتم‌های موازی، پردازش موازی و محاسبات ابری.