Journal of Soft Computing and Information Technology (JSCIT)
Babol Noshirvani University of Technology, Babol, Iran
Journal Homepage: jscit.nit.ac.ir
Volume 12, Number 2, Summer 2023, pp. 48-59
Received: 09/03/2022, Revised: 06/05/2023, Accepted: 08/20/2023

Improving the inter-block synchronization methods in CUDA

Abdorreza Savadi[1*],Mohaddese Salavatizadeh[2] ,and Ali Riahi[3]
1,2,3- Computer Engineering Department, Ferdowsi University of Mashhad, Mashhad, Iran
[1*]savadi@um.ac.ir, [2]salavatizadeh@mail.um.ac.ir, and [3]riahi@mail.um.ac.ir

Corresponding author's address: Abdorreza Savadi, Faculty of Engineering, Ferdowsi University of Mashhad, Mashhad, Iran.

Abstract- The lack of explicit support for inter-block synchronization in the CUDA programming model has weakened performance in some applications. Therefore, in such applications, inter-block synchronization must be implemented in software. Lock-based and lock-free methods have been implemented for this problem. In lock-based synchronization, the execution time increases significantly with the increase in the number of blocks, and in the lock-free methods, there is a limit to the number of blocks. In this paper, two inter-block synchronization methods are proposed. The first method is lock-based, which reduces the impact of increasing the number of blocks on the execution time by grouping the blocks. The second proposed method is lock-free synchronization, which removes the limitation of the number of blocks in synchronization by creating a tree hierarchy of blocks. These methods were used for inter-block synchronization in Smith-Waterman and Bitonic algorithms. Experimental results show that the proposed lock-based method improves the execution time of the synchronization and recorded a speedup of 1.84 in the Smith-Waterman algorithm and 2.24 in the Bitonic sorting algorithm. Also, the results show that in the proposed lock-free method, any number of blocks can be synchronized by correctly choosing the number of levels of the tree hierarchy, and therefore the limitation of the number of blocks has been removed.

Keywords- GPU, CUDA, Inter-block Synchronization, Lock-based and Lock-free Synchronization.

مجله علمی پژوهشی رایانش نرم و فناوری اطلاعات
دانشگاه صنعتی نوشیروانی بابل
صفحه مجله: jscit.nit.ac.ir
جلد ۱۲، شماره ۲، تابستان ۱۴۰۲ ، صفحه ۴۸-۵۹
دریافت: ۱۴۰۱/۰۶/۱۲، بازنگری: ۱۴۰۲/۰۳/۱۵، پذیرش: ۱۴۰۲/۰۵/۲۹

بهبود روش‌های همگام‌سازی بین‌بلاکی در کودا

عبدالرضا سوادی*[1]، محدثه صلواتی‌زاده[2]، علی ریاحی[3]
۳،۲،۱- دانشکده مهندسی، دانشگاه فردوسی مشهد، مشهد، ایران.
[1*]savadi@um.ac.ir, [2]salavatizadeh@mail.um.ac.ir, [3]riahi@mail.um.ac.ir

* نشانی نویسنده مسئول: عبدالرضا سوادی، مشهد، میدان آزادی، دانشگاه فردوسی مشهد، دانشکده مهندسی، گروه مهندسی کامپیوتر.

چکیده- عدم پشتیبانی صریح همگام‌سازی بین بلاکی در مدل برنامه‌نویسی کودا، باعث تضعیف کارآیی در برخی از برنامه‌های کاربردی شده است. بنابراین در چنین برنامه‌هایی، همگام‌سازی بین بلاکی باید به‌صورت نرم‌افزاری پیاده‌سازی شود. روش‌های مبتنی‌بر قفل و بدون قفل برای این مسئله پیاده‌سازی شده‌اند. در همگام‌سازی مبتنی‌بر قفل، زمان اجرا با افزایش تعداد بلاک رشد چشمگیری دارد و در روش همگام‌سازی بدون‌قفل، محدودیت تعداد بلاک‌ها وجود دارد. در این مقاله، دو روش همگام‌سازی بین بلاکی پیشنهاد می‌شوند. اولین روش مبتنی‌بر همگام‌سازی مبتنی‌بر قفل است که با گروه‌بندی مناسب بلاک‌ها، تاثیر افزایش تعداد بلاک بر زمان اجرا را کاهش می‌دهد. دومین روش پیشنهادی همگام‌سازی بدون قفل است که با ایجاد یک سلسله‌مراتبی درختی از بلاک‌ها، محدودیت تعداد بلاک‌ها در این همگام‌سازی را مرتفع می‌کند. این روش‌ها برای همگام‌سازی بین بلاکی در الگوریتم‌های اسمیت واترمن و مرتب‌سازی بایتونیک به کار گرفته شده‌اند. نتایج آزمایش‌ها نشان می‌دهند که روش مبتنی‌بر قفل پیشنهادی، زمان اجرای همگام‌سازی را بهبود می‌بخشد و تسریع ۱/۸۴ در الگوریتم اسمیت واترمن و ۲/۲۴ را در الگوریتم مرتب‌سازی بایتونیک ثبت کرده است. همچنین نتایج نشان می‌دهند که در روش پیشنهادی بدون قفل نیز با انتخاب درست تعداد سطوح سلسله‌مراتب درختی، هر تعداد بلاک می‌توانند همگام شوند و بنابراین محدودیت تعداد بلاک‌ها مرتفع شده است.

واژه‌های کلیدی: واحد پردازنده‌ی گرافیکی، کودا، همگام‌سازی بین بلاکی، همگام‌سازی مبتنی‌بر قفل و بدون قفل.

۱- مقدمه

به دلیل مدل چندنخی گسترده و کارا بودن اجرای برنامه‌ها روی پردازنده‌ی گرافیکی، در سال‌های اخیر بسیاری از برنامه‌های کاربردی روی این پردازنده‌ها منتقل شده‌اند [۱] و این پردازنده بستر مناسبی برای موازی‌سازی داده فراهم کرده است. اما ناهمزمانی[۱] مسئله‌ی مهمی برای برنامه‌ها است. گاهی می‌توان ناهمزمانی را در قسمتی از کد چشم‌پوشی کرد تا عملکرد کلی را بهبود بخشید [۲]، و یا می‌توان نخ‌ها را مستقل از هم طراحی کرد. اما در بسیاری از برنامه‌ها لازم است که نخ‌ها با هم ارتباط داشته باشند، داده‌ی میانی را با یکدیگر تبادل کنند و یا اینکه لازم باشد داده‌ی آن‌ها با هم ترکیب شوند. بنابراین در این موارد لازم است که نخ‌ها با ایجاد سد[۲]، همگام شوند و دستورالعمل‌های همگام‌سازی را فراخوانی کنند و تا زمانی که تمام نخ‌های دیگر به این نقطه برسند، منتظر باشند.

در مدل برنامه‌نویسی کودا، تابعی که باید روی پردازنده‌ی گرافیکی اجرا شود، کرنل نامیده می‌شود. کرنل یک برنامه‌ی ترتیبی است که چندین نخ از آن ایجاد شده و به‌صورت موازی‌سازی داده اجرا می‌شوند. نخ‌ها در دسته‌هایی به نام بلاک‌ها سازماندهی می‌شوند. در کودا امکان همگام‌سازی فقط بین نخ‌های یک بلاک (توسط تابع $syncthreads()$) فراهم شده است و برای همگام‌سازی بین‌بلاکی[۳] (همگام‌سازی سراسری[۴]) تابعی پیشنهاد نشده است. برای کارآیی بهتر پردازنده‌ی گرافیکی در پردازش‌های محاسباتی باید این محدودیت برطرف شود. بنابراین در پژوهش‌های زیادی روش‌هایی ارائه شده‌اند که همگام‌سازی را به‌صورت نرم‌افزاری پیاده‌سازی کرده‌اند.

روش‌های نرم‌افزاری همگام‌سازی بین بلاکی به چهار دسته تقسیم شده‌اند [۳]. در دسته‌ی اول پیشنهاد می‌شود که از همگام‌سازی سراسری پرهیز شود و برای این منظور الگوریتم تغییر یابد [۴-۷]. در این دسته از روش‌ها سعی می‌شود که الگوریتم پیاده‌سازی کرنل به‌گونه‌ای طراحی شود تا بلاک‌ها نسبت به هم مستقل باشند و اساسا نیازی به همگام‌سازی سراسری وجود نداشته باشد. بدیهی است که در بسیار موارد استفاده از این روش امکان‌پذیر نیست. دسته‌ی دوم همگام‌سازی گرید [۸] است که به صورت ضمنی در انتهای اجرای کرنل انجام می‌شود و به همگام‌سازی پردازنده‌ی مرکزی[۵] نیز معروف است. در این روش هر تکراری از کرنل که به همگام‌سازی سراسری نیاز دارد، به‌صورت یک کرنل مستقل فراخوانی می‌شود و در پایان هر کرنل، همگام‌سازی ضمنی انجام شده و سپس کرنل بعدی فراخوانی می‌شود. در پژوهش‌های زیادی ازجمله در [۹-۱۲] از این روش استفاده شده است. دسته‌ی سوم روش‌های مبتنی‌بر قفل[۶] هستند [۱۳-۱۵]. ایده‌ی اصلی این روش‌ها استفاده از یک متغیر سراسری و اتمیک برای شمارش تعداد بلاک‌هایی است که به نقطه همگام‌سازی رسیده‌اند. چالش اصلی در این روش‌ها این است که با افزایش تعداد بلاک‌ها، زمان اجرای فرآیند همگام‌سازی سراسری رشد محسوسی دارد. دسته‌ی چهارم روش‌های بدون قفل[۷] هستند [۱۳،۱۴]. چالش این روش‌ها در محدودیت تعداد بلاک‌هایی است که می‌توانند با هم همگام شوند. رویکردهای سوم و چهارم دو روش کلی برای همگام‌سازی بین بلاکی هستند و در بخش بعدی تشریح خواهند شد. این دو روش موضوع اصلی این مقاله هستند. به دلیل زمان‌بر بودن فرآیند همگام‌سازی سراسری نرم‌افزاری، در برخی موارد سربار تولید شده از اجرای آن بیشتر از محاسبات کرنل بوده و انجام همگام‌سازی مقرون به صرفه نیست [۱۶].

در این مقاله ما دو روش پیشنهادی برای همگام‌سازی سراسری ارائه می‌کنیم که چالش‌های روش‌های قبلی را برطرف می‌کنند:

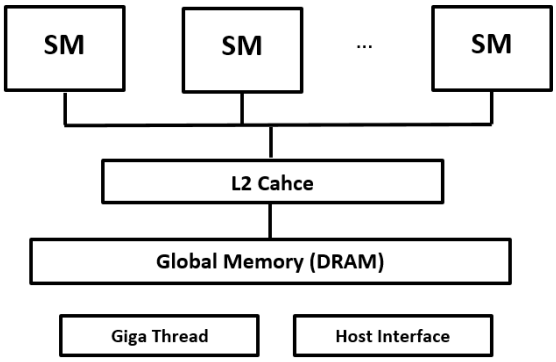
- روش پیشنهادی اول مبتنی‌بر قفل است که در آن چالش افزایش زمان اجرای فرآیند همگام‌سازی سراسری با افزایش تعداد بلاک‌ها مرتفع شده است.
- روش دوم پیشنهادی یک روش بدون قفل است که چالش محدودیت تعداد بلاک‌های شرکت‌کننده در همگام‌سازی سراسری را برطرف می‌کند.

در ادامه‌ی این مقاله، ابتدا در بخش ۲ پیش‌زمینه‌های موردنیاز برای این مقاله تشریح و در بخش ۳ پژوهش‌های قبلی در این حوزه بررسی خواهند شد. سپس در بخش ۴ روش‌های پیشنهادی این مقاله مطرح شده و در بخش ۵ این روش‌ها ارزیابی می‌شوند. در بخش ۶ نتایج به‌دست آمده مورد بحث قرار گرفته و نتیجه‌گیری انجام می‌شود و پیشنهاداتی برای ادامه‌ی پژوهش ارائه خواهند شد.

۲- پیش‌زمینه

۲-۱- معماری پردازنده‌ی گرافیکی

اجزای اصلی یک پردازنده‌ی گرافیکی که در شکل ۱ نشان داده شده است، شامل یک حافظه‌ی سراسری[۸]، حافظه‌ی نهان L2، واسط میزبان، گیگاترد[۹] و واحدهای مهمی به نام چندپردازنده‌های جریانی[۱۰] هستند. حافظه‌ی سراسری تنها حافظه‌ای است که می تواند با حافظه‌ی اصلی پردازنده‌ی مرکزی تبادل داده داشته باشد. واسط میزبان وظیفه‌ی برقراری ارتباط و هماهنگی با پردازنده‌ی مرکزی را بر عهده دارد. گیگاترد زمانبند سطح یک پردازنده‌ی گرافیکی است. هسته‌های پردازشی در داخل چندپردازنده‌های جریانی سازماندهی شده‌اند. چندپردازنده‌های جریانی کاملا با هم مشابه هستند و علاوه‌بر هسته‌های پردازشی، شامل حافظه‌های برتراشه[۱۱] نیز می‌شوند. این حافظه‌ها شامل رجیستر فایل، حافظه نهان سطح یک، حافظه‌های ثابت[۱۲]، بافت[۱۳] و اشتراکی[۱۴] می‌باشند [۱۷]. این حافظه‌ها به‌طور اشتراکی توسط نخ‌های یک بلاک در دسترس هستند و اهمیت فوق‌العاده‌ای در تسریع برنامه‌ها دارند.



شکل ۱- نمای کلی از ساختار پردازنده‌های گرافیکی شرکت

حافظه‌ی نهان سطح دو خارج تراشه[۱۵] است، اما در سطح راه‌انداز دستگاه مدیریت می‌شود و برنامه‌نویس نمی‌تواند روی آن متغیر یا آرایه‌ای تعریف کند. حافظه‌های ثابت و بافت جزئی از حافظه‌ی سراسری هستند. این حافظه‌ها به همراه حافظه‌ی سراسری تنها حافظه‌هایی هستند که می‌توانیم متغیرهایی روی آن‌ها ایجاد کنیم که به‌صورت بین بلاکی قابل استفاده باشند و در همگام‌سازی سراسری از آن‌ها استفاده کنیم. علاوه‌براین حافظه‌ها، حافظه‌ی محلی نیز وجود دارد که با حافظه‌ی سراسری یکپارچه است و توسط سیستم زمان اجرای کودا[۱۶] مدیریت می‌شود و برنامه‌نویس کنترلی روی آن ندارد. این حافظه زمانی استفاده می‌شود که

آرایه‌ای در داخل کرنل تعریف شود و یا اینکه رجیستر کافی برای اجرای یک نخ در دسترس نباشد [۸].

۲-۲- مدل برنامه‌نویسی کودا

برنامه‌های کودا فقط روی پردازنده‌های گرافیکی انویدیا[۱۷] اجرا می‌شوند. این برنامه‌ها از دو بخش میزبان و کرنل تشکیل شده‌اند و توسط مترجم کودای انویدیا[18] از هم جدا می‌شوند. بخش کرنل روی پردازنده‌ی گرافیکی اجرا می‌شود و یک کد ترتیبی است که نمونه‌های زیادی از آن به‌صورت نخ‌ها اجرا می‌شوند. پیکربندی کرنل اهمیت بسیار زیادی در بهبود موازی‌سازی و کارآیی برنامه دارد. در این پیکربندی تعداد بلاک‌ها (اندازه‌ی گرید) و تعداد نخ‌ها در هر بلاک (اندازه‌ی بلاک) تعیین می‌شوند. تمام نخ‌های یک بلاک روی یک چندپردازنده‌ی جریانی اجرا می‌شوند. البته ممکن است چند بلاک روی یک چندپردازنده‌ی جریانی اجرا شوند؛ ولی تمام بلاک‌ها مستقل از هم هستند و نخ‌های آن‌ها جز ازطریق حافظه‌ی سراسری که می‌توانند داده را با هم به اشتراک بگذارند، هیچ ارتباطی با هم ندارند. باتوجه به کند بودن حافظه‌ی سراسری این ارتباطات جریمه‌ی سنگینی به کارآیی تحمیل می‌کند. درعوض، نخ‌های یک بلاک با استفاده از حافظه‌های برتراشه (خصوصا حافظه‌ی اشتراکی) با هم ارتباط تنگاتنگی دارند و با کارآیی بالایی می‌توانند با هم همکاری داشته باشند. این موضوع یکی از دلایلی است که در کودا همگام‌سازی درون بلاکی پیاده‌سازی شده، اما همگام‌سازی بین بلاکی پیاده‌سازی نشده است.

۳- مروری بر کارهای گذشته

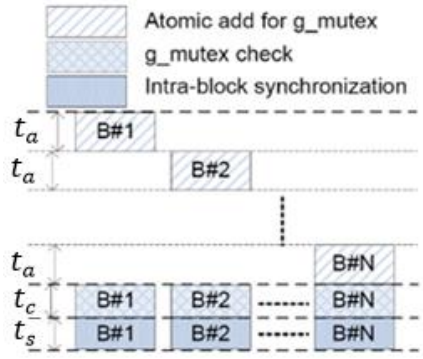
پائولو و همکاران [۱۱] الگوریتم مجارستانی که برای مسئله‌ی تخصیص خطی ارائه شده است را به‌صورت موازی در بستر کودا پیاده‌سازی کردند. آن‌ها برای افزایش کارآیی از چندین بلاک استفاده کردند و بنابراین در هر تکرار الگوریتم نیازمند همگام‌سازی سراسری بودند. برای این منظور، از روش همگام‌سازی گرید استفاده کردند و هر تکرار در الگوریتم موازی را به صورت یک کرنل مستقل فراخوانی کردند. وانگ و همکاران [۱۸] روش همگام‌سازی سراسری مبتنی‌بر قفل را بر روی الگوریتم بهینه‌سازی ذرات[۱۹] اجرا کردند. نسخه‌ی قبلی این الگوریتم همگام‌سازی سراسری را با استفاده از روش گرید و با استفاده از دو کرنل انجام می‌داد. وانگ و همکاران این دو کرنل را ازطریق تکنیک همجوشی کرنل[۲۰] به یک کرنل تبدیل کردند. هر نخ از کرنل دوم پس از خاتمه‌ی نخ کرنل اول آغاز می‌شود. بین اجرای دو کرنل همگام‌سازی مبتنی‌بر قفل را با یک متغیر سراسری اتمیک پیاده‌سازی کردند. آن‌ها با این عمل به تسریع ۲۰۰ دست یافتند. به عبارت دیگر زمان اجرا ۲۰۰ برابر کاهش یافته است. در ادامه دو روش همگام‌سازی مبتنی‌بر قفل و بدون قفل ارائه خواهند شد.

۳-۱- همگام‌سازی مبتنی بر قفل

در روش همگام‌سازی مبتنی‌بر قفل [۱۴] یک متغیر اتمیک و سراسری وجود دارد که کنترل دسترسی به آن، ازحیث انحصار متقابل، به‌صورت ضمنی مدیریت می‌شود. همان‌طور که در الگوریتم ۱ نشان داده شده است، متغیر g_mutex به‌صورت اتمیک و ترتیبی به‌وسیله‌ی نماینده‌ی هر بلاک (نخ با شناسه صفر) یک واحد افزوده می‌شود. پس از اینکه همه‌ی نمایندگان بلاک‌ها افزایش متغیر g_mutex را انجام دادند، شرط حلقه‌ی تکرار به اتمام می‌رسد و همگام‌سازی تمامی نخ‌ها محقق می‌شود. لازم به ذکر است مقدار متغیر $goalVal$ برابر با تعداد بلاک‌ها است.

```
__device__ volatile int g_mutex;
__device__ void LBS (int goalVal){
        int tid = threadIdx.x*blockDim.y + hreadIdx.y;
        if (tid == 0) {
                atomicAdd((int *)&g_mutex, 1);
                while(g_mutex != goalVal) ;
        }
        __syncthreads();
}
```

الگوریتم ۱- همگام‌سازی سراسری مبتنی بر قفل [۱۴].



شکل ۲- محاسبه‌ی زمان اجرای همگام‌سازی مبتنی‌بر قفل [۱۴].

فرض کنید که در کرنل N بلاک وجود دارد، زمان هر افزودن اتمیک و چک کردن g_mutex به ترتیب، t_a و t_c است، زمان همگام‌سازی در داخل بلاک t_s است. در شکل ۲ فرآیند انجام همگام‌سازی به‌صورت خط زمانی نشان داده شده است. در این شکل تمام بلاک‌ها محاسبات خود را هم‌زمان به پایان می‌رسانند. همچنین براساس این شکل زمان اجرای همگام‌سازی به‌صورت معادله‌ی ۱ محاسبه می‌شود که در آن N تعداد بلاک‌ها است. در

این معادله هزینه‌ی همگام‌سازی مبتنی‌بر قفل متناسب با N به‌صورت خطی افزایش می‌یابد [۱۴].

$$t_{LBS} = N \times t_a + t_c + t_s \quad (۱)$$

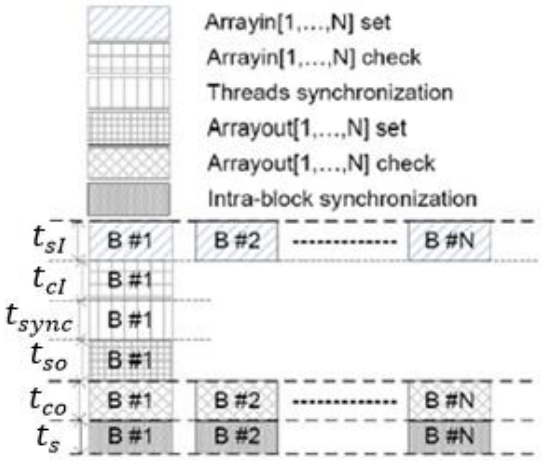
در [۱۹] روش مبتنی‌بر قفل روی الگوریتم جستجوی اول سطح اجرا شده است و در [۲۰] نیز این روش برای شبیه‌سازی مونت کارلو در یک مسئله‌ی دنیای واقعی استفاده شده است. نتایج این پژوهش نشان می‌دهد که برای اندازه‌ی کوچک داده (تا ۵۱۲) به دلیل سربار زیاد همگام‌سازی سراسری، استفاده از پردازنده‌ی گرافیکی مقرون به صرفه نیست. اما برای داده‌ی بزرگتر، با افزایش حجم محاسبات، عملکرد پردازنده‌ی گرافیکی بهتر از پردازنده‌ی مرکزی است.

۳-۲- روش همگام‌سازی بدون قفل

روش‌های مبتنی‌بر قفل، به دلیل دسترسی‌های ترتیبی به متغیرهای اتمیک، زمان‌بر هستند [۲۱]. در روش همگام‌سازی بدون قفل از عملیات اتمیک اجتناب می‌شود. ایده‌ی اصلی این روش اختصاص یک متغیر همگام‌سازی برای هر بلاک است؛ به‌طوری‌که هر بلاک می‌تواند وضعیت همگام‌سازی خود را بدون رقابت با سایر نخ‌ها ثبت کند. همانطورکه در الگوریتم ۲ نشان داده شده است، روش همگام‌سازی بدون قفل از دو آرایه $Arrayin$ و $Arrayout$ برای همگام‌سازی بلاک‌های متفاوت استفاده می‌کند. فرض می‌کنیم که پارامتر t_{SI} زمان نوشتن یک سلول از $Arrayin$، t_{CI} زمان چک کردن یک سلول از $Arrayin$ و t_S زمان همگام‌سازی درون بلاک است. زمان نوشتن و چک کردن یک سلول از $Arrayout$ نیز به ترتیب t_{SO} و t_{CO} است [۱۴]. در این دو آرایه، هر سلول متعلق به یک بلاک است. باتوجه به الگوریتم ۲، با شروع همگام‌سازی سراسری، نخ شماره‌ی صفر هر بلاک، سلول متناظر در $Arrayin$ را به $goalVal$ مقداردهی می‌کند. زمان اجرای این عمل t_{SI} است. باید توجه شود باوجودیکه N بلاک این آرایه را مقداردهی می‌کنند، ولی در عمل فقط زمان نوشتن آخرین بلاک حس می‌شود و مابقی توسط محاسبات سایر بلاک‌ها پنهان[۲۱] می‌شوند. در ادامه بلاک شماره‌ی یک مقادیر این آرایه‌ی $Arrayin$ را چک کرده و درصورتیکه همه‌ی بلاک‌ها سلول متناظر خودشان را مقداردهی کرده باشند، آرایه‌ی $Arrayout$ را مقداردهی می‌کند. زمان این عملیات چک کردن t_{CI} و زمان نوشتن در $Arrayout$ نیز برابر t_{SO} است؛ چراکه نخ‌های بلاک یک همگی به‌صورت موازی این اعمال را انجام می‌دهند و به همین دلیل است که حداکثر تعداد بلاک‌های شرکت‌کننده در همگام‌سازی سراسری محدود به اندازه‌ی بلاک است. درحین فعالیت بلاک یک، سایر بلاک‌ها توسط نخ شماره‌ی صفر مشغول چک کردن سلول متناظر از آرایه‌ی $Arrayout$ هستند و زمان صرف شده برای این منظور t_{CO} است. توجه شود که بلاک‌ها همگی به‌صورت موازی این عمل را انجام می‌دهند. در اکثر اجراها آخرین بلاکی که از چک کردن این آرایه عبور کرده است، بلاک شماره‌ی یک بوده است. در شکل ۳ توالی اجرای این روش نشان داده شده است و زمان اجرای آن ازطریق معادله‌ی ۲ محاسبه می‌شود.

```
__device__ void LFS (int goalVal,
        volatile int *Arrayin, volatile int *Arrayout){
    int tid = threadIdx.x * blockDim.y+ threadIdx.y;
    int nBlockNum = gridDim.x * gridDim.y;
    int bid = blockIdx.x * gridDim.y + blockIdx.y;
    if (tid == 0)
        Arrayin[bid] = goalVal;
    if (bid == 1) {
        if (tid < nBlockNum)
        while (Arrayin[tid] != goalVal);
        __syncthreads();
        if (tid < nBlockNum)
            Arrayout[tid] = goalVal;
    }
    if (tid == 0)
        while (Arrayout[bid] != goalVal) ;
    __syncthreads();
}
```

الگوریتم ۲- همگام‌سازی سراسری بدون قفل [۱۴].



شکل ۳- محاسبه‌ی زمان اجرای همگام‌سازی بدون قفل [۱۴].

$$t_{LFS} = t_{SI} + t_{CI} + t_s + t_{SO} + t_{CO} \quad (۲)$$

در همگام‌سازی مبتنی‌بر قفل، بین زمان اجرای همگام‌سازی با تعداد بلاک‌های کرنل یک رابطه‌ی خطی وجود دارد. برای همگام‌سازی بدون قفل، ازآنجاکه هیچ عملیات اتمیک وجود ندارد،

تمام عملیات را می‌توان به‌صورت موازی اجرا کرد. همچنین زمان همگام‌سازی به تعداد بلاک‌های یک کرنل وابسته نیست و بنابراین زمان همگام‌سازی تقریبا مقداری ثابت است. البته در ارزیابی‌ها نشان داده خواهد شد که این زمان کاملا ثابت نیست. در همگام‌سازی بدون قفل، تعداد بلاک‌ها باید از اندازه‌ی بلاک کمتر باشد؛ چون برای همگام‌سازی بین بلاکی لازم است که به‌ازای هر بلاک، یک نخ نماینده در بلاک ناظر (بلاک شماره‌ی یک) وجود داشته باشد. اگر تعداد بلاک‌ها از تعداد نخ‌های بلاک ناظر بیشتر باشد، بلاک ناظر روی آن بلاک‌های اضافه نظارتی نداشته و درنتیجه همگام‌سازی به‌درستی صورت نمی‌گیرد و این یک محدودیت بسیار مهم در همگام‌سازی بدون قفل است.

۴- روش‌های پیشنهادی

۴-۱- روش پیشنهادی مبتنی‌بر قفل بهبودیافته

در بخش قبل نشان دادیم که با افزایش تعداد بلاک‌ها زمان اجرای همگام‌سازی مبتنی‌بر قفل به‌صورت خطی رشد می‌کند. این رشد در تعداد بلاک زیاد، چشمگیر است. در این بخش اولین روش پیشنهادی را برای کاهش زمان اجرا و تسریع مناسب این روش ارائه می‌کنیم.

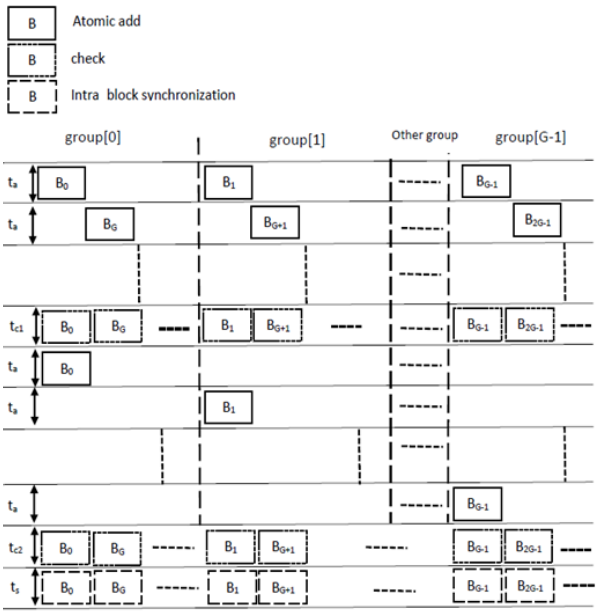
پیشنهاد می‌شود بلاک‌ها به گروه‌های تقریبا برابر تقسیم شوند و هر گروه متغر مخصوص به خود ($G\_mutex[i]$) را داشته باشد که به‌صورت اتمیک توسط بلاک‌های موجود در گروه به میزان یک واحد اضافه می‌شود. زمانی‌که متغیر گروه برابر تعداد اعضای گروه ($group[i]$) شد، لازم است متغیر سراسری g_mutex توسط بلاک نماینده‌ی گروه که شناسه آن با شماره گروه برابر است (نماینده‌ی گروه صفر، بلاک با شناسه صفر است) یک واحد افزوده شود. کد همگام‌سازی برای گروه i ام به‌صورت الگوریتم ۳ است. این الگوریتم را مبتنی‌بر قفل بهبودیافته[22] می‌نامیم. هنگامی‌که مقدار متغیر g_mutex برابر تعداد گروه‌ها شود، بدان معنی است که تمامی گروه‌ها جمع اتمیک را انجام دادند. در این حالت تابع $syncthreads()\_\_$ فراخوانی می‌شود و همگام‌سازی درون بلاکی صورت می‌گیرد.

```
if (tid_in_block == 0) {
        atomicAdd ((int *)&G_mutex[i] , 1);
        while ( G_mutex[i] != group[i] ) ;
        if (blockID == i)
            atomicAdd((int *)&g_mutex , 1);
}
```

الگوریتم ۳- کد مربوط به گروه i ام برای همگام‌سازی مبتنی‌بر قفل بهبودیافته.

تحلیل زمانی این روش:

فرض می‌کنیم که در کرنل N بلاک وجود داشته باشد. همچنین فرض می‌کنیم که t_a زمان عملیات جمع اتمیک، t_{c1} و t_{c2} زمان چک کردن متغیرها و t_s نیز زمان همگام‌سازی درون بلاکی باشند. معادله‌ی ۳ زمان اجرای نسخه‌ی پیشنهادی همگام‌سازی مبتنی‌بر قفل در شکل ۴ را نشان می‌دهد. پارامترهای معادله‌ی ۳ در جدول ۱ تشریح شده‌اند. هدف از عملیات جمع اتمیک در هر گروه، حصول اطمینان از این است که تمام بلاک‌ها آمادگی جهت همگام‌سازی را دارند و به نقطه موردنظر رسیده‌اند. هدف از جمع اتمیک توسط نماینده‌ی بلاک‌ها این است که اطمینان حاصل شود که تمام گروه‌ها به نقطه همگام‌سازی رسیده‌اند. درصورتیکه این دو شرط برقرار باشند، آنگاه همگام‌سازی سراسری اتفاق افتاده است و در انتها تابع $syncthreads()\_\_$ فراخوانی می‌شود.



شکل ۴- محاسبه‌ی زمان اجرای روش پیشنهادی مبتنی‌بر قفل.

با افزایش تعداد بلاک‌ها زمان اجرای روش مبتنی‌بر قفل بهبودیافته مانند روش مبتنی‌بر قفل رشد خطی دارد. بنابراین تعداد گروه‌ها برای دسته‌بندی بلاک‌ها اهمیت فوق العاده‌ای دارد. کمترین زمان اجرا وقتی حاصل می‌شود که ضریب t_a در معادله‌ی ۳ کمینه شود. این ضریب در معادله‌ی ۴ به‌صورت تابعی از متغیر n_group با مقدار ثابت N تعریف شده است. ریشه‌ی مشتق این تابع مقدار بهینه را برای متغیر n_group ارائه می‌کند. بنابراین مقدار بهینه برای تعداد گروه‌ها به‌صورت $n\_group = \sqrt{N}$ است و این بدان معنی است که روش مبتنی‌بر قفل بهبودیافته زمانی بهترین کارایی را ارائه می‌کند که تعداد گروه‌ها برابر جذر تعداد بلاک‌ها باشد.

$$t_{ILBS} = \left\lceil \frac{N}{n\_group} \right\rceil \times t_a + n\_group.t_a + t_{c1} + t_{c2} + t_s$$
$$= \left(\left\lceil \frac{N}{n\_group} \right\rceil + n\_group\right) \times t_a + t_{c1} + t_{c2} + t_s \quad (۳)$$

$$f(n\_group) = \left\lceil \frac{N}{n\_group} \right\rceil + n\_group \quad (۴)$$

جدول ۱- تشریح پارامترهای معادله‌ی ۳.

پارامتر	توضیحات
t_{ILBS}	زمان اجرای روش مبتنی‌بر قفل بهبودیافته
N	تعداد بلاک‌ها
n_group	تعداد گروه‌ها (عددی بین ۱ تا N)
t_a	زمان هر افزودن اتمیک
t_{c1}	زمان بررسی $G\_mutex[]$
t_{c2}	زمان بررسی $g\_mutex[]$
t_s	زمان همگام‌سازی درون بلاکی

۴-۲- روش پیشنهادی بدون قفل نامحدود

نشان دادیم که در روش همگام‌سازی بدون قفل تعداد بلاک‌های شرکت‌کننده در همگام‌سازی حداکثر باید به تعداد اندازه‌ی بلاک باشد. در این بخش روشی پیشنهاد می‌کنیم که این محدودیت را مرتفع می‌کند و آن را همگام‌سازی بدون قفل نامحدود[23] می‌نامیم. این روش را می‌توان برای تعداد بلاک‌های موردنظر توسعه داد. در این روش، سلسله‌مراتبی از بلاک‌ها در سطوح مختلف درنظرگرفته می‌شوند. این سلسله‌مراتب براساس نیاز، قابل گسترش است و می‌توان با داشتن m سطح، $m \times Blocksize - 1$ بلاک را همگام کرد. شماره‌گذاری سطوح از 0 تا $m\text{-}1$ است. اگر N بلاک وجود داشته باشد، تعداد سطوح موردنیاز از معادله‌ی ۵ به‌دست می‌آید.

$$m = \lceil log_{Blocksize} N \rceil + 1 \quad (۵)$$

پایین‌ترین سطح (سطح m-1) شامل تمامی بلاک‌هایی هست که نیاز به همگام‌سازی دارند و بالاترین سطح (سطح صفر) شامل یک بلاک هست. بلاک با شناسه یک را در بالاترین سطح و سطوح میانی را براساس چینشی از کل بلاک‌ها در نظر می‌گیریم. قابل ذکر است برای ایفای نقش بلاک‌های سطوح میانی، بلاک جداگانه‌ای در نظر گرفته نمی‌شود و از همان بلاک‌های موردنظر برای همگام‌سازی استفاده می‌کنیم که در سطوح مختلف این سلسله‌مراتب، وظایف گوناگونی را انجام می‌دهند تا درنهایت به همگام‌سازی سراسری برسیم. در این سلسله مراتب، مراحل همگام‌سازی از طریق مقداردهی آرایه‌ها از سطحی به سطح دیگر گزارش داده می‌شود، تا متناسب با وضعیت آرایه‌ها، مراحل همگام‌سازی دنبال شود. هر سطح i دو آرایه مختص به خود دارد. برای درک بهتر، نام آرایه‌ها براساس شماره سطح نام‌گذاری شده است. آرایه $Arrayin_i$ و آرایه $Arrayout_i$ آرایه‌های مختص به سطح i هستند.

روال الگوریتم پیشنهادی به این صورت است که بلاک‌های سطح i به‌واسطه‌ی نماینده‌شان (نخ با شناسه صفر) آرایه $Arrayin_i$ مختص به سطح خود را مقداردهی کرده، سپس سطح i-1 توسط تمامی نخ‌هایش آرایه $Arrayin_i$ را بررسی می‌کند و تا زمانی‌که هریک از بلاک‌های سطح i، آرایه $Arrayin_i$ را مقداردهی نکنند، نخ‌های سطح i-1 در حالت انتظار می‌مانند. بلاک‌های سطح i-1 نیز آرایه $Arrayin_{i-1}$ را توسط نماینده‌ها مقداردهی می‌کنند و این سلسله مراتب تا رسیدن به بالاترین سطح (سطح صفر با فقط یک بلاک) ادامه می‌یابد. بالاترین سطح مانند سایر سطح‌ها موظف به بررسی سطح پایینی خودش (سطح ۱) است. آرایه $Arrayin_1$ توسط نخ‌های سطح صفر مورد بررسی قرار می‌گیرد. اگر مقداردهی آرایه $Arrayin_1$ صورت گرفته باشد، تابع $\_\_syncthreads()$ فراخوانی می‌شود و همگام‌سازی در سطح صفر انجام می‌گیرد. این کار بدین معناست که بلاک‌های سطح ۱ همگام شده‌اند. آرایه $Arrayout_1$ توسط نخ‌های سطح صفر مقداردهی شده تا همگام شدن نمایندگان سطح ۱ را به اطلاع بلاک‌های سطح ۱ برساند.

برای انتشار همگام‌سازی از بالاترین به پایین‌ترین سطح سلسه‌مراتبی نیاز است. هر نخ نماینده در سطح i بررسی می‌کند که آرایه $Arrayout_i$ توسط سطح i-1 مقداردهی شده یا نه؟ اگر مقداردهی شده باشد، $\_\_syncthreads()$ فراخوانی شده و همگام‌سازی در آن سطح انجام می‌گیرد و آرایه $Arrayout_{i+1}$ برای سطح i+1 را مقداردهی می‌کند. این مقداردهی باعث می‌شود که بلاک‌های سطح i+1 از همگام‌شدن نمایندگان خود مطلع شوند. به این ترتیب همگام‌سازی بین سطوح انتشار می‌یابد. در آخرین سطح (سطح m-1)، پس از بررسی $Arrayout_{m-1}$ با فراخوانی تابع $\_\_syncthreads()$ باعث همگام‌سازی تمامی بلاک‌ها می‌شوند. اساس کار این است که از پایین‌ترین سطح سلسله‌مراتب به سمت بالاترین سطح رفته و با مقداردهی آرایه‌های in اعلام آمادگی جهت همگام‌سازی انجام می‌شود. سپس اولین همگام‌سازی در بالاترین سطح به کمک $Arrayout$ صورت می‌گیرد و به سمت سطوح پایینی انتشار می‌یابد. کد این روال در الگوریتم ۴ نشان داده شده است.

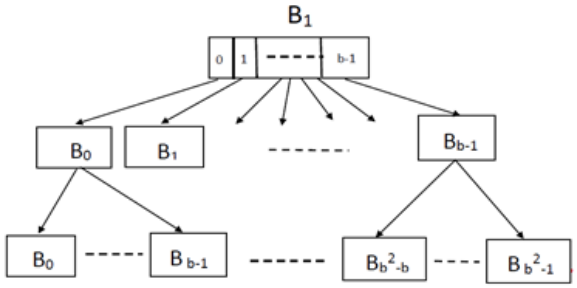
روش همگام‌سازی بدون قفل نامحدود جهت رفع محدودیت تعداد بلاک در روش همگام‌سازی بدون قفل ارائه شده است. طبیعتا برای رفع این محدودیت هزینه‌هایی باید متحمل شد. در ادامه به تحلیل این روش پرداخته می‌شود. شکل ۵ سلسله مراتب سه سطحی از

بلاک‌ها را برای اجرای روش پیشنهادی نشان می‌دهد. در این شکل پارامتر b برابر مقدار $Blocksize$ و B_i نیز بلاک i ام است. با در نظرگرفتن این سلسله‌مراتب سه سطحی می‌توان حداکثر برای $Blocksize^2$ تا بلاک همگام‌سازی سراسری را انجام داد. اگر این درخت در m سطح تشکیل گردد، آنگاه الگوریتم می‌تواند تا $Blocksize^{m-1}$ بلاک را همگام کند. بنابراین با افزایش تعداد بلاک‌ها، فقط کافی است تعداد سطوح این درخت افزایش یابد و درنتیجه محدودیتی در تعداد بلاک‌های شرکت‌کننده در فرآیند همگام‌سازی سراسری وجود ندارد. شکل ۶ وظایف هرسطح از سلسله مراتب شکل ۵ را در طول زمان همگام‌سازی نشان می‌دهد.

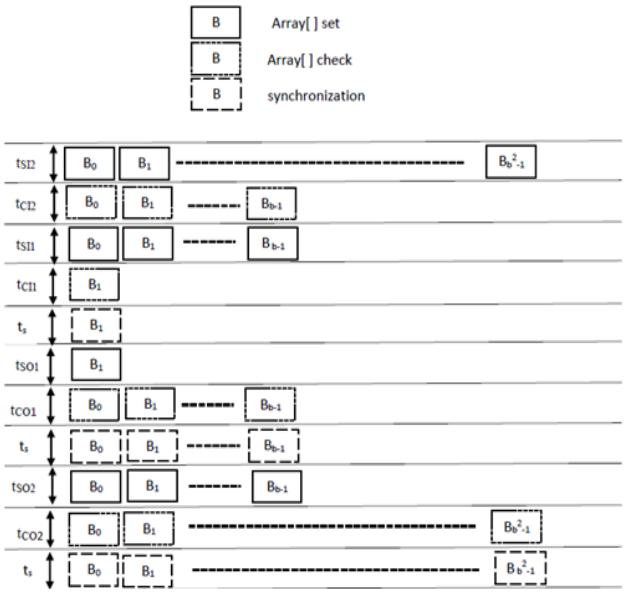
```
__device__ volatile int *Arrayin, *Arrayout;
__device__ volatile int  *Arraycheckin, *Arraycheckout;
__device__ void ULFS (int goalVal ) {
    int tid = threadIdx.x * blockDim.y + threadIdx.y;
    int bid = blockIdx.x * gridDim.y + blockIdx.y;
    int tidingrid = bid*BlockSize+ tid;
    int nBlockNum = gridDim.x * gridDim.y;
    if (tid == 0)  Arrayin[bid] = goalVal;
    x= nBlockNum / BlockSize ;
    if (nBlockNum % BlockSize != 0)   x=x+1;
    if (x <= 1) { LFS(…);  }
    else {
        if ( bid >= 0 and bid <= x-1 ){
            if (tidingrid < nBlockNum) {
            while (Arrayin[tidingrid] != goalVal);
            if (tid == 0) Arraycheckin[bid] = goalVal;
        }
        if (bid == 1) {
            if (tid < x)
                while (Arraycheckin[tid] != goalVal);
            __syncthreads();  // level 0
            synchronization
            if (tid < x)  Arraycheckout[tid] = goalVal;
        }
        if ( bid >= 0 and bid <= x-1 ){
            if (tid == 0)
                while (Arraycheckout[bid]!= goalVal);
            __syncthreads();  // level 1
            synchronization
            if (tidingrid < nBlockNum)
                Arrayout[tid] = goalVal;
        }
        if (tid == 0) while (Arrayout[bid] != goalVal);
        __syncthreads();
    }
}
```

الگوریتم ۴- کد روال همگام‌سازی سراسری بدون قفل نامحدود.

زمان همگام‌سازی محاسبه شده در معادله‌ی ۶ ارائه شده است. این زمان برای تعداد بلاک بین $Blocksize + 1$ تا $Blocksize^2$ تقریبا ثابت بوده و تابعی از تعداد بلاک‌ها نیست. پارامترهای استفاده شده در معادله‌ی ۶ و شکل ۶ در جدول ۲ معرفی شده‌اند. به میزان $t_{SI1} + t_{CI1} + t_S + t_{SO1} + t_{CO1}$ افزایش زمان نسبت به روش پایه همگام‌سازی بدون قفل داریم.



شکل ۵- سلسله مراتب بلاک‌ها در روش ULFS.



شکل ۶- محاسبه‌ی زمان اجرای روش بدون قفل نامحدود

$$t_{ULFS} = t_{SI2} + t_{CI2} + t_{SI1} + t_{CI1} + 3t_S + t_{SO1} + t_{CO1} + t_{SO2} + t_{CO2} \qquad (۶)$$

در این قسمت روش‌های همگام‌سازی پیشنهادی و پیشین با دو الگوریتم اسمیت واترمن[۲۴] [۲۴] و مرتب‌سازی بایتونیک[۲۵] [۲۲] مورد ارزیابی قرار می‌گیرند و از نظر زمان اجرا مقایسه می‌شوند. دلیل انتخاب این دو الگوریتم این است که اولا در [۱۴] برای ارزیابی از این دو الگوریتم استفاده شده است و دوما کدهای پیاده‌سازی شده‌ی این دو الگوریتم نیز دردسترس هستند. مرتب‌سازی بایتونیک یک الگوریتم موازی مرتب‌سازی است که براساس توالی‌های بایتونیک پیاده‌سازی می‌شود. یک توالی بایتونیک دنباله‌ای از اعداد است که حاوی یک بخش صعودی و یک بخش نزولی است. روال مرتب‌سازی به‌صورت بازگشتی

پیاده‌سازی می‌شود که البته در نسخه‌ی موازی به‌ازای هر فراخوانی بازگشتی یک نخ ایجاد می‌شود. این نخ‌ها باید با هم همگام شوند تا مرحله‌ی بعدی در الگوریتم قابل فراخوانی باشد. در [۲۳] الگوریتم بایتونیک در بستر کودا و تنها با یک بلاک پیاده‌سازی شده است و برای همگام‌سازی نیز از تابع $\_\_syncthreads()$ استفاده شده است. در [۱۴] این الگوریتم به‌صورت چندبلاکی پیاده‌سازی شده است که ما این الگوریتم را با دو روش پیشنهادی بهبود دادیم.

جدول ۲- تشریح پارامترهای معادله‌ی ۶ و شکل ۶

پارامتر	توضیحات
t_{ULFS}	زمان اجرای روش بدون قفل نامحدود
t_{SI2}	زمان مقداردهی $Arrayin$ در سطح دو توسط نخ‌های با شناسه‌ی صفر در هر بلاک
t_{CI2}	زمان بررسی $Arrayin$ در سطح یک توسط همه‌ی نخ‌های بلاک
t_{SI1}	زمان مقداردهی $Arrayin$ در سطح یک توسط نخ‌های با شناسه‌ی صفر در هر بلاک
t_{CI1}	زمان بررسی $Arrayin$ در سطح صفر توسط همه‌ی نخ‌های بلاک
t_s	زمان همگام‌سازی درون بلاکی
t_{SO1}	زمان مقداردهی $Arraycheckout$ در سطح صفر توسط تمام نخ‌های بلاک
t_{CO1}	زمان بررسی $Arraycheckout$ در سطح یک توسط تمام نخ‌های با شناسه‌ی صفر در هر بلاک
t_{SO2}	زمان مقداردهی $Arrayout$ در سطح یک توسط تمام نخ‌های بلاک
t_{CO2}	زمان بررسی $Arrayout$ در سطح دو توسط نخ‌ها با شناسه‌ی صفر در هر بلاک

الگوریتم اسمیت واترمن در بایوانفورماتیک برای یافتن یک هم‌ترازی در توالی‌های اسید نوکلئیک و یا پروتئین استفاده می‌شود. درواقع این الگوریتم محدوده‌های مشابهت توالی‌ها را تشخیص می‌دهد. این الگوریتم از نوع برنامه‌نویسی پویا است و به دلیل استفاده از ماتریس‌ها، مناسب برای پیاده‌سازی روی پردازنده‌ی گرافیکی است. مشابه بسیاری از الگوریتم‌های برنامه‌نویسی پویا، الگوریتم اسمیت واترمن نیز از چند مرحله تشکیل شده است که در پایان هر مرحله نخ‌ها باید با هم همگام شوند. این الگوریتم در [۲۵] در بستر کودا پیاده‌سازی شده است و در [۱۴] به‌صورت چندبلاکی پیاده‌سازی شده و همگام‌سازی سراسری روی آن بکار گرفته شده است. در این بخش با استفاده از دو روش پیشنهادی عملکرد این الگوریتم را بهبود داده‌ایم.

دو روال همگام‌سازی پیشنهادی روی دو الگوریتم اسمیت واترمن و مرتب‌سازی بایتونیک که نیازمند همگام‌سازی بین بلاکی می‌باشند پیاده‌سازی شده‌اند. اندازه‌ی بلاک‌ها ۳۲ درنظرگرفته شده و آزمایش‌ها برای ۷ تا ۶۰ بلاک انجام شده‌اند. برای تعداد بلاک کمتر از ۷ زمان همگام‌سازی بیشتر از زمان اجرای محاسبات است. به همین دلیل این میزان داده مناسب اجرا روی پردازنده‌ی گرافیکی نمی‌باشد. در جدول ۳ سیستم استفاده شده در آزمایشات معرفی شده است. معیار ارزیابی برای مقایسه‌ی الگوریتم‌های مبتنی‌بر قفل و مبتنی‌بر قفل بهبودیافته زمان اجرای برنامه برحسب ثانیه است. در آزمایشات هدف بهبود (کاهش) زمان اجرای برنامه‌هاست. معیار ارزیابی برای مقایسه‌ی الگوریتم‌های بدون قفل و بدون قفل نامحدود حذف محدودیت تعداد بلاک‌ها در همگام‌سازی سراسری است.

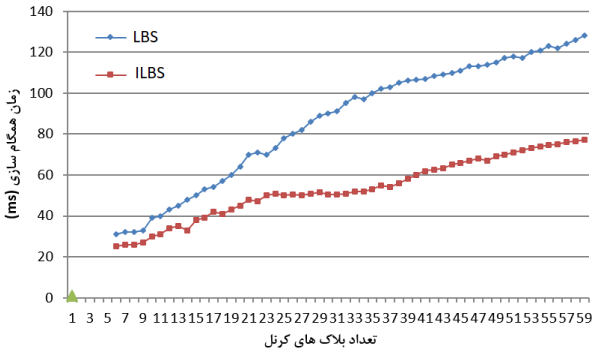
جدول ۳- مشخصات دستگاه استفاده شده در آزمایشات.

کارت گرافیکی	GTX ۱۰۷۰
معماری پردازنده‌ی گرافیکی (قابلیت محاسباتی)	پاسکال (۶/۱)
تعداد هسته‌های یک چندپردازنده جریانی × تعداد چندپردازنده‌های جریانی	۱۲۸ × ۱۵
تعداد ثبات‌ها در هر چندپردازنده جریانی	۶۵۵۳۶
حداکثر اندازه‌ی حافظه‌ی اشتراکی	۴۸ کیلوبایت
حداکثر اندازه‌ی حافظه‌ی نهان سطح یک	۴۸ کیلوبایت
اندازه‌ی حافظه‌ی نهان سطح دو	۲ مگابایت
اندازه‌ی حافظه‌ی سراسری	۸ گیگابایت
حداکثر تعداد نخ‌ها در یک بلاک	۱۰۲۴
حداکثر تعداد نخ‌ها در یک چندپردازنده جریانی	۲۰۴۸
نسخه‌ی کودا	۹/۲
سیستم عامل	اوبونتو ۱۸/۰۴

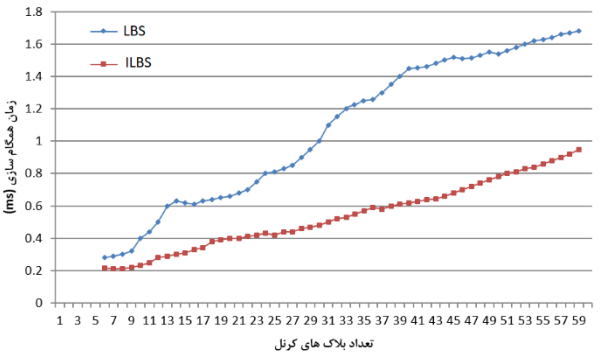
۴-۳- ارزیابی روش همگام‌سازی مبتنی‌بر قفل بهبودیافته

در این بخش روش‌های همگام‌سازی مبتنی‌بر قفل و مبتنی‌بر قفل بهبودیافته با استفاده از دو الگوریتم اسمیت واترمن و مرتب‌سازی بایتونیک مقایسه می‌شوند. متریک ارزیابی این دو روش تسریع در زمان اجرای عملیات همگام‌سازی است. برای الگوریتم اسمیت واترمن از مجموعه داده‌ی پروسایت استفاده شده و برای مرتب‌سازی بایتونیک داده‌ی تصادفی تولید شده است. سایز بلاک‌ها ۳۲ درنظرگرفته شده و برای ۷ تا ۶۰ بلاک آزمایش انجام شده است. برای تعداد بلاک کمتر از ۷ مشاهده شده است که زمان همگام‌سازی بیشتر از زمان اجرای محاسبات برنامه‌ها روی پردازنده‌ی گرافیکی است و بنابراین مقرون‌به‌صرفه نیست. برای دسته‌بندی بلاک‌ها در روش پیشنهادی، ۶ گروه در نظر گرفته شده است. شکل ۷ زمان اجرای دو روش همگام‌سازی مبتنی‌بر قفل و مبتنی‌بر قفل بهبودیافته را روی الگوریتم اسمیت واترمن، و شکل ۸ نیز زمان اجرای این دو روش را روی الگوریتم مرتب‌سازی

بایتونیک نشان می‌دهد. مقادیر تسریع این دو روش در جدول‌های ۴ و ۵ ارائه شده‌اند. نکته‌ی بسیار مهم این است که بیشترین تسریع وقتی حاصل می‌شود که جذر تعداد بلاک‌ها برابر تعداد گروه‌ها (یعنی عدد ۶) است. در بخش قبلی با اثبات ریاضی دلیل آن را شرح دادیم و در اینجا با شبیه‌سازی نیز اعتبارسنجی شده است.



شکل ۷- مقایسه‌ی زمان اجرا روش‌های همگام‌سازی مبتنی‌بر قفل و مبتنی‌بر قفل بهبودیافته با اجرای الگوریتم اسمیت واترمن



شکل ۸- مقایسه‌ی زمان اجرا روش‌های همگام‌سازی مبتنی‌بر قفل و مبتنی‌بر قفل بهبودیافته با اجرای الگوریتم مرتب‌سازی بایتونیک

محاسبه‌ی تسریع در جدول‌های ۴ و ۵ از طریق معادله‌ی ۷ است. بلاک‌ها در این دو جدول به دسته‌هایی با طول برابر تقسیم شده‌اند. برای ثبت زمان اجرا، چندین مرحله برنامه‌ها را اجرا کرده و با انجام پروفایلینگ، زمان اجراها اندازه‌گیری شده است. سپس از میانگین زمان‌های اجرا ثبت شده است. ضمن اینکه از چند اجرای اولیه نیز صرفنظر شده است.

$$speed\ up = \frac{t_{LBS}}{t_{ILBS}} \qquad (۷)$$

۴-۴- ارزیابی روش همگام‌سازی بدون قفل نامحدود

در این بخش روش‌های همگام‌سازی بدون قفل و بدون قفل نامحدود با استفاده از دو الگوریتم اسمیت واترمن و مرتب‌سازی بایتونیک ارزیابی و مقایسه شده‌اند. اندازه‌ی بلاک‌ها ۳۲ درنظرگرفته شده و برای ۷ تا ۶۰ بلاک آزمایش انجام شده است.

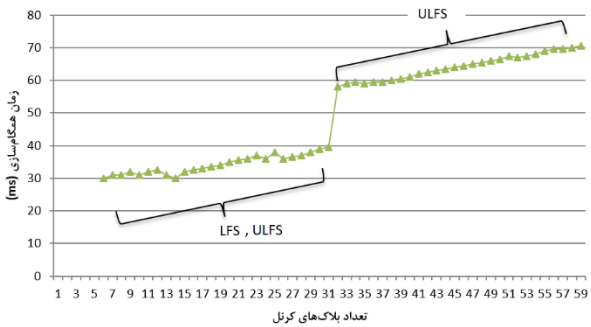
جدول ۴- تسریع روش مبتنی‌بر قفل بهبودیافته روی الگوریتم اسمیت واترمن

هابندی بلاک‌دسته	t_{LBS}	t_{ILBS}	تسریع
۷-۱۵	۳۸/۱۱	۲۹/۶۶	۱/۲۸
۱۶-۲۴	۶۱	۴۳/۶۵	۱/۳۹
۲۵-۳۳	۸۴/۸۸	۵۰/۶۷	۱/۶۷
۳۴-۴۲	۱۰۲/۷۲	۵۰/۷۷	**۱/۸۴**
۴۳-۵۱	۱۱۱/۷۲	۶۶/۴۳	۱/۶۸
۵۲-۶۰	۱۳۷/۳۷	۹۱/۶۵	۱/۴۹

جدول ۵- تسریع همگام‌سازی مبتنی‌بر قفل بهبودیافته روی الگوریتم بایتونیک.

هابندی بلاک‌دسته	t_{LBS}	t_{ILBS}	تسریع
۷-۱۵	۰/۴۱	۰/۲۴۵	۱/۶۷
۱۶-۲۴	۰/۶۶	۰/۳۳	۲
۲۵-۳۳	۰/۸	۰/۴۶	۱/۷
۳۴-۴۲	۱/۳	۰/۵۸	**۲/۲۴**
۴۳-۵۱	۱/۵۱	۰/۷	۲/۱۵
۵۲-۶۰	۱/۸۳	۰/۹۷	۱/۹

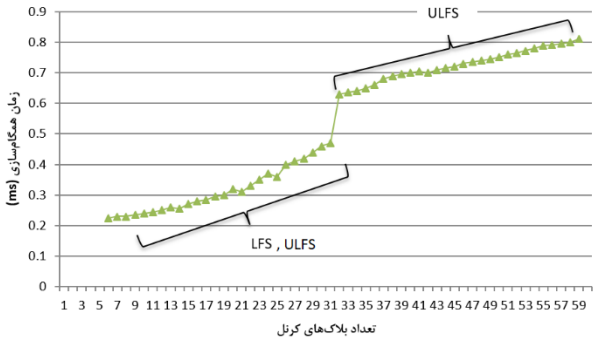
باتوجه به اینکه سایز بلاک ۳۲ است، روش همگام‌سازی بدون قفل حداکثر ۳۲ بلاک را می‌تواند همگام کند. برای تعداد بلاک بیشتر از ۳۲، روش پیشنهادی همگام‌سازی بدون قفل نامحدود استفاده می‌شود. در پیاده‌سازی مربوط به الگوریتم اسمیت واترمن (شکل ۹) و پیاده‌سازی مربوط به مرتب‌سازی بایتونیک (شکل ۱۰) مشاهده می‌شود. برای تعداد بلاک بیشتر از ۳۲ زمان اجرا افزایش چشمگیری نسبت به بلاک‌های قبلی دارد.



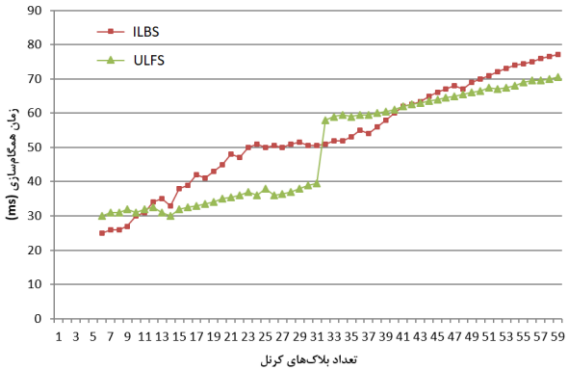
شکل ۹- زمان اجرای روش‌های همگام‌سازی بدون قفل و بدون قفل نامحدود با استفاده از الگوریتم اسمیت واترمن

در روش پیشنهادی به علت گسترش سلسله‌مراتب همگام‌سازی، زمان اجرا افزایش می‌یابد. اگر تعداد بلاک‌ها ۳۳ باشد در الگوریتم اسمیت واترمن ۴۶/۸ درصد و در مرتب‌سازی بایتونیک ۳۴ درصد افزایش زمان اجرا خواهیم داشت. این مقادیر میزان افزایش زمان اجرا در روش همگام‌سازی بدون قفل نامحدود را نشان می‌دهند.

در شکل‌های ۱۱ و ۱۲ دو روش پیشنهادی مبتنی‌بر قفل بهبودیافته و بدون قفل نامحدود با یکدیگر مقایسه شده‌اند. در شکل ۱۱ هردو روش روی الگوریتم اسمیت واترمن و در شکل ۱۲ هردو روش روی الگوریتم بایتونیک اجرا شده‌اند.



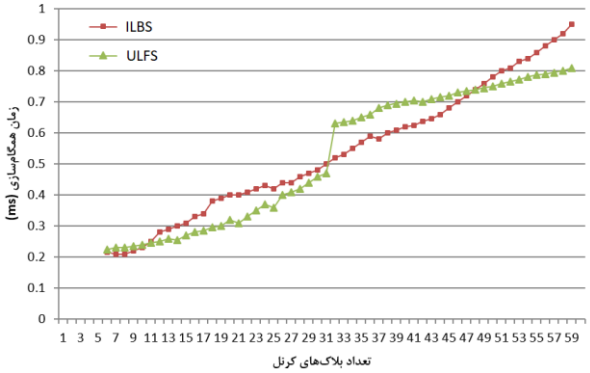
شکل ۱۰- زمان اجرای روش‌های همگام‌سازی بدون قفل و بدون قفل نامحدود با استفاده از الگوریتم مرتب‌سازی بایتونیک



شکل ۱۱- مقایسه‌ی دو روش همگام‌سازی مبتنی‌بر قفل بهبودیافته و بدون قفل نامحدود با استفاده از الگوریتم اسمیت واترمن

۵- نتیجه‌گیری و کارهای آینده

در این پژوهش به مسئله‌ی همگام‌سازی سراسری (بین بلاکی) در برنامه‌های کودا پرداخته شده است و دو روش همگام‌سازی سراسری مبتنی‌بر قفل بهبودیافته و بدون قفل نامحدود پیشنهاد شده است. این روش‌ها با روش‌های همگام‌سازی سراسری مبتنی‌بر قفل و بدون قفل که در سایر پژوهش‌ها ارائه شده‌اند، مقایسه شده‌اند.



شکل ۱۲- مقایسه‌ی دو روش همگام‌سازی مبتنی بر قفل بهبودیافته و بدون قفل نامحدود با استفاده از الگوریتم مرتب‌سازی بایتونیک

چالش اساسی در روش مبتنی‌بر قفل این بود که زمان اجرای این روش با افزایش تعداد بلاک‌ها به‌طور چشمگیری افزایش می‌یابد. نتایج ارزیابی‌ها نشان داده‌اند که روش پیشنهادی مبتنی‌بر قفل بهبودیافته با ایجاد گروه‌بندی روی بلاک‌ها، در مقایسه با روش مبتنی‌بر قفل سرعت اجرای بهتری دارد و به تسریع ۱/۸۴ در الگوریتم اسمیت واترمن و ۲/۲۴ در الگوریتم بایتونیک دست یافته است. روش پیشنهادی مبتنی‌بر قفل بهبودیافته براساس گروه‌بندی بلاک‌های شرکت‌کننده در همگام‌سازی سراسری عمل می‌کند. نخ نماینده‌ی هر گروه به متغیرهای اتمیک دسترسی خواهد داشت و بدین‌ترتیب دسترسی همزمان و رقابت برای در اختیار گرفتن متغیرهای اتمیک کاهش می‌یابد. نمودار مقایسه‌ای دو روش مبتنی‌بر قفل و مبتنی‌بر قفل بهبودیافته در شکل‌های ۷ و ۸ که به‌ترتیب برای الگوریتم‌های اسمیت واترمن و بایتونیک ارائه شده‌اند، نشان از یک الگوی تقریبا مشابه در تسریع زمان اجرا دارد. نقطه‌ی بسیار مثبت روش مبتنی‌بر قفل بهبودیافته این است که به لحاظ تئوریک نقطه‌ی بهینه برای تعداد گروه‌ها را ارائه کرده است و نیازی به جستجو یا آزمون و خطا برای یافتن آن وجود ندارد. مقدار بهینه‌ی تعداد گروه‌ها برابر جذر تعداد بلاک‌های شرکت‌کننده در همگام‌سازی سراسری است و با این تعداد گروه بهترین تسریع حاصل خواهد شد. ما به دو روش این مسئله را نشان دادیم. ابتدا با استفاده از معادله‌ی ۴ و یافتن ریشه مشتق آن به لحاظ تئوریک این نکته را نشان دادیم و سپس با ارائه‌ی نتایج شبیه‌سازی در جدول‌های ۴ و ۵ آن را اعتبارسنجی کردیم. در نتیجه این روش توانسته است افزایش بلاک بر زمان اجرای همگام‌سازی را کاهش دهد و چالش آن تاحدودی مرتفع شده است.

چالش بسیار مهم در روش بدون قفل این است که این روش فقط زمانی می‌تواند مورد استفاده قرار گیرد که تعداد بلاک‌های

شرکت‌کننده در همگام‌سازی سراسری، حداکثر برابر اندازه‌ی بلاک باشد. ما با ارائه‌ی روش بدون قفل نامحدود این محدودیت را مرتفع کردیم. در این روش سلسله‌مراتب درختی از گروه‌ها تشکیل دادیم که با داشتن m سطح حداکثر $Blocksize^{m-1}$ تا بلاک را می‌تواند همگام کند. بنابراین تعداد بلاک‌ها هرچه باشد، با انتخاب درست تعداد سطوح درخت، قابل همگام‌سازی هستند. نکته‌ی مهم این است که همانطور که در الگوریتم ۴ نشان داده شده است، اگر تعداد بلاک‌ها از $Blocksize$ کوچک‌تر باشد، آنگاه روال بدون قفل نامحدود نیازی به ایجاد سلسله‌مراتب درختی در بلاک‌ها ندارد و همان روال بدون قفل را فراخوانی می‌کند. بنابراین در این حالت به لحاظ زمان اجرا هردو مشابه هستند و بنابراین روش پیشنهادی سرباری برای همگام‌سازی بدون قفل ندارد. ازطرفی زمان اجرای روش بدون قفل نامحدود به‌ازای افزایش تعداد بلاک‌ها، افزایش چشمگیری ندارد. دلیل این مسئله این است که فرآیند همگام‌سازی با جداسازی داده‌ی مشترک برای بلاک‌ها (برخلاف روش مبتنی‌بر قفل) امکان اجرای موازی را فراهم کرده است. مگر اینکه تعداد بلاک‌ها به قدری افزایش یابد که باعث افزایش تعداد سطوح در سلسله‌مراتب درختی بلاک‌ها شود. در این صورت در زمان اجرای همگام‌سازی یک جهش خواهیم داشت. شکل‌های ۹ و ۱۰ به خوبی این مسئله را نشان داده‌اند.

در معادله‌ی ۲ زمان اجرای همگام‌سازی سراسری بدون قفل را ارائه کردیم. این معادله نشان می‌دهد که زمان اجرای فرآیند همگام‌سازی سراسری به تعداد بلاک‌های مشارکت‌کننده در آن بستگی ندارد. اما در شکل ۱۰ نشان داده شده است که درعمل با افزایش تعداد بلاک‌ها، زمان همگام‌سازی نیز اندکی افزایش دارد. دلیل این مسئله این است که با افزایش تعداد بلاک‌ها، زمان‌های سربار تعویض متن بین بلاک‌های روی یک چندپردازنده‌ی جریانی افزایش می‌یابد. نقطه ضعف معادلات ارائه شده در این مقاله و مقاله‌های قبلی این است که به‌طور کامل از این زمان‌های سربار صرفنظر شده است.

در شکل‌های ۱۱ و ۱۲ زمان اجرای روش‌های پیشنهادی مبتنی‌بر قفل بهبودیافته و بدون قفل نامحدود روی دو الگوریتم اسمیت واترمن و بایتونیک مقایسه شده‌اند. بررسی این دو شکل نشان می‌دهد که رفتار دو روش مشابه است. در اکثر موارد روش بدون قفل نامحدود بهتر عمل می‌کند، به جز زمانی که تعداد بلاک‌ها کمی بیشتر از مضربی از اندازه‌ی بلاک باشد. در این موارد در زمان اجرای این روش یک جهش ایجاد می‌شود. هردو کرنل را با تعداد بلاک‌های بسیار زیاد فراخوانی کردیم و نتایج نشان می‌دهند که روش بدون قفل نامحدود عملکرد بهتری نسبت به روش مبتنی‌بر قفل بهبودیافته دارد.

در بخش ارزیابی، زمان اجرای چهار روش (دو روش قبلی و دو روش پیشنهادی) باهم مقایسه شده‌اند. این ارزیابی‌ها براساس اندازه‌گیری‌های حاصل از شبیه‌سازی انجام شده است. باوجودیکه برای این چهار روش معادلات ۱، ۲، ۳ و ۶ ارائه شده‌اند، اما در ارزیابی از این معادلات استفاده نشده است. دلیل بسیار مهمی وجود دارد که در اینجا بیان می‌شود. در این معادلات فقط زمان اجرای فعالیت‌های مرتبط با همگام‌سازی سراسری ذکر شده‌اند و سربارهای مرتبط با زمانبندی‌های سطح یک و دو در پردازنده‌ی گرافیکی درنظر گرفته نشده‌اند. سربار زمانبندی سطح یک مربوط به زمان تعویض متن بلاک‌ها روی یک پردازنده‌ی جریانی و سربار زمانبندی سطح دو نیز زمان تعویض متن بین وارپ‌های درون یک بلاک است. در زمان انتشار این مقاله، روش‌های مشخصی برای تخمین دقیق این زمان‌ها دردسترس نیست. علاوه‌براین، این معادلات براساس پارامترهایی ارائه شده‌اند که تخمین آن‌ها نیازمند داشتن یک مدل کارآیی و انجام عملیات ریزمحک‌زنی[۲۶] است که از محدوده‌ی این تحقیق خارج است و در پژوهش‌های بعدی دنبال خواهد شد. درعوض، ارائه‌ی این معادلات به تشریح بهتر مراحل همگام‌سازی سراسری کمک کرده است.

به‌عنوان کارهای آینده پیشنهاد می‌شود که:

- روش‌های پیشنهادی روی مسائل بزرگتر و پیچیده‌تر اعمال شوند تا کارآیی آن‌ها افزایش یابد.
- همچنین پیشنهاد می‌شود که برای همگام‌سازی سراسری از ساختارهای داده در سطوح حافظه مثل حافظه‌های بافت و ثابت استفاده شود و عملکرد آن‌ها مورد ارزیابی قرار گیرد.
- یکی از دلایل اجتناب از همگام‌سازی سراسری، احتمال ایجاد بن‌بست است. بنابراین پیشنهاد می‌شود، یک مدل ایستا برای پیش‌بینی بن‌بست ایجاد شود. این مدل با استفاده از ریزمحک‌زنی منابع موجود در پردازنده‌ی گرافیکی را اندازه‌گیری می‌کند. همچنین با تحلیل ایستای کدهای کودا و کدهای سطح پایین شبه‌اسمبلی[۲۷] منابع موردنیاز هر بلاک و کرنل را نیز تخمین می‌زند. سپس پیش‌بینی می‌کند که آیا احتمال وقوع بن‌بست وجود دارد یا خیر؟ بنابراین می‌تواند تصمیم‌گیری کند که همگام‌سازی سراسری انجام شود یا خیر؟ و یا اینکه از کدام روش همگام‌سازی استفاده شود؟

مراجع

[1] Gao, Lan, Jing Wang, and Weigong Zhang. "Adaptive contention management for fine-grained synchronization on commodity GPUs." ACM Transactions on Architecture and Code Optimization (TACO), Vol.19, No.4, pp 1-21, 2022.

[2] Xi, Rong-Ping, et al. "An Asynchronous Parallel Implementation of Multilevel Fast Multipole Algorithm on GPU Cluster for 3D Electromagnetic Scattering Problems." In 2021 International Applied Computational Electromagnetics Society (ACES-China) Symposium. IEEE, 2021.

[3] Al-Mouhamed, Mayez A., Ayaz H. Khan, and Nazeeruddin Mohammad. "A review of CUDA optimization techniques and tools for structured grid computing." Computing, Vol.102, No.4, pp 977-1003, 2020

[4] Yan, Shengen, Guoping Long, and Yunquan Zhang. "StreamScan: fast scan algorithms for GPUs without global barrier synchronization.", In Proceedmorphings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, 2013.

[5] Dufrechou, Ernesto, Pablo Ezzatti, and Gabriel Usera. "Avoiding synchronization to accelerate a CFD solver in GPU." In 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE, 2019.

[6] Jørgensen, Jakob Rødsgaard, et al. "GPU-FAST-PROCLUS: A Fast GPU-parallelized Approach to Projected Clustering." Advances in Database Technology-EDBT, 2022.

[7] Li, Ruipeng, and Chaoyu Zhang. "Efficient parallel implementations of sparse triangular solves for GPU architectures." In Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing. Society for Industrial and Applied Mathematics, 2020.

[8] NVIDIA, "CUDA C best practices guide", 2019

[9] Peng, Yuanfeng, Vinod Grover, and Joseph Devietti. "CURD: a dynamic CUDA race detector." ACM SIGPLAN Notices, Vol.53, No.4, pp 390-403, 2018.

[10] Bikov, Dusan, and Ilija Bouyukliev. "Parallel fast Walsh transform algorithm and its implementation with CUDA on GPUs." Cybernetics and Information Technologies, Vol.18, No.5, pp 21-43, 2018.

[11] Petrovič, Filip, et al. "A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit." Future Generation Computer Systems, Vol.108, pp 161-177, 2020.

[12] Lopes, Paulo AC, et al. "Fast block distributed CUDA implementation of the Hungarian algorithm." Journal of Parallel and Distributed Computing, Vol.130, pp 50-62, 2019.

[13] Xiao, Shucai, Ashwin M. Aji, and Wu-chun Feng. "On the robust mapping of dynamic programming onto a graphics processing unit." In 15th International Conference on Parallel and Distributed Systems. IEEE, 2009.

[14] Xiao, Shucai, and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization." In IEEE International Symposium on Parallel & Distributed Processing (IPDPS). IEEE, 2010.

[15] Hagedorn, Christopher, et al. "GPU Acceleration for Information-theoretic Constraint-based Causal Discovery." In the KDD'22 Workshop on Causal Discovery. PMLR, 2022.

[16] Feng, Wu-chun, and Shucai Xiao. "To GPU synchronize or not GPU synchronize?." In IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2010.

[17] NVIDIA, "NVIDIA Turing GPU Architecture: Graphics reinvented.", 2018.

[18] Wang, Chuan-Chi, et al. "cuPSO: GPU parallelization for particle swarm optimization algorithms." In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. 2022.

[19] Luo, Lijuan, Martin Wong, and Wen-mei Hwu. "An effective GPU implementation of breadth-first search." In Design Automation Conference. IEEE, 2010.

[20] Komura, Yukihiro, and Yutaka Okabe. "GPU-based single-cluster algorithm for the simulation of the Ising model." Journal of Computational Physics Vol.231, No.4, pp 1209-1215, 2012.

[21] Nasre, Rupesh, Martin Burtscher, and Keshav Pingali. "Atomic-free irregular computations on GPUs." In Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units. 2013.

[22] Batcher, Kenneth E. "Sorting networks and their applications." In Proceedings of the April 30--May 2, 1968, spring joint computer conference. 1968.

[23] Greb, Alexander, and Gabriel Zachmann. "GPU-ABiSort: Optimal parallel sorting on stream architectures." *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006.

[24] Smith, Temple F., and Michael S. Waterman. "Identification of common molecular subsequences." *Journal of molecular biology*, Vol.147, No.1, pp 195-197, 1981.

[25] Manavski, Svetlin A., and Giorgio Valle. "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment." BMC bioinformatics Vol.9, pp 1-9, 2008.

پاورقی‌ها:

[1] Asynchronous

[2] Barrier

[3] Inter-block Synchronization

[4] Global Synchronization

[5] CPU Synchronization

[6] Lock-based Synchronization (LBS)

[7] Lock-free Synchronization (LFS)

[8] Global Memory

[9] GigaThread

[10] Streaming Multiprocessor (SM)

[11] On-chip

[12] Constant

[13] Texture

[14] Shared Memory

[15] Off-chip

[16] CUDA Runtime System

[17] Nvidia

[18] Nvidia CUDA Compiler (NVCC)

[19] PSO

[20] Kernel Fusion

[21] Hide

[22] Improved Lock-based Synchronization

[23] Unlimited Lock-free Synchronization (ULFS)

[24] SWat

[25] Bitonic Sorting

[26] Micro-bechmarking

[27] PTX Code