مجله محاسبات نرم
SOFT COMPUTING JOURNAL
تارنمای مجله: scj.kashanu.ac.ir
دانشگاه کاشان
University of Kashan

روشی برای بهبود الگوریتم بهینه‌سازی اجتماع ذرات با استفاده از CUDA بر روی پردازنده‌گرافیکی ✧

محمد پویا اکبرپور[۱]، کارشناسی ارشد، کیهان خام‌فروش[۲*]، استادیار، وفا میهمی[۳]، استادیار
[۱،۲ و ۳] دانشکده مهندسی کامپیوتر، دانشگاه آزاد اسلامی واحد سنندج، سنندج، ایران.

اطلاعات مقاله

تاریخچه مقاله:
دریافت ۰۷ دی ماه ۱۳۹۸
پذیرش ۲۰ مرداد ماه ۱۳۹۹

کلمات کلیدی:
موازی‌سازی الگوریتم
بهینه‌سازی اجتماع ذرات
Fermi
GPU Computing
HPC
CUDA

چکیده

همواره زمان صرف‌شده برای حل مسائل سنگین محاسباتی، یکی از دغدغه‌های برنامه‌نویسان کامپیوتر بوده است. الگوریتم PSO، الگوریتمی فراابتکاری است که به‌دلیل سادگی پیاده‌سازی، برای حل مسائل سنگین محاسباتی استفاده می‌شود ولی با وجود سادگی، این الگوریتم برای حل مسائل سنگین واقعی ناکارآمد است. از طرفی، وجود ویژگی تعاملات محلی ذرات در الگوریتم PSO، این الگوریتم را برای موازی‌سازی مناسب کرده است؛ از طرف دیگر، NVIDIA با اختراع پردازنده‌گرافیکی و معرفی معماری CUDA، تحولات بنیادی را در حل این نوع مسائل، از طریق پیاده‌سازی آن بر روی پردازنده‌گرافیکی ایجاد کرده است. با وجود تمام تحقیقات انجام‌گرفته در زمینهٔ پیاده‌سازی، برخی از جنبه‌های تکنیکی موازی‌سازی به‌منظور پیاده‌سازی الگوریتم به‌صورتی که تسریع و بازدهی مناسب بر روی تمام پردازنده‌های گرافیکی NVIDIA را داشته باشد، رعایت نشده است. در این مقاله سعی شده با انتخاب Geforce GT 525M که پردازنده‌گرافیکی نسبتاً ضعیفی است، جنبهٔ مقیاس‌پذیری روش پیشنهادی رعایت شود؛ به‌طوری که با رسیدن به بیشینهٔ تسریع الگوریتم پیاده‌سازی‌شده بر روی این پردازنده، به بازدهی قابل قبول برای اجرا بر روی سایر پردازنده‌های گرافیکی رسید. برای نیل به این هدف، از مدل چندکرنلی ارائه‌شده استفاده شده است. نتایج حاصل از انجام آزمایش‌ها رسیدن به بیشینهٔ تسریع ۱۵/۹۸ برای حل تابع Rastrigin را نشان می‌دهد.

۱. مقدمه

الگوریتم‌های هوش اجتماعی[۱]، مجموعه‌ای از الگوریتم‌های مبتنی بر جمعیت هستند که از مجموعه رفتارهای اجتماعات موجود در طبیعت الهام گرفته شده‌اند [۱ و ۲]. در الگوریتم‌های هوش اجتماعی، اجتماع از چند عامل ساده تشکیل شده است که این عامل‌ها می‌توانند اطلاعات اکتشافی خود را از طریق تعاملات محلی مستقیم یا غیرمستقیم، با هم معاوضه و به اشتراک بگذارند [۳]. معروف‌ترین الگوریتم این خانواده، الگوریتم بهینه‌سازی اجتماع ذرات (PSO) است که از رفتار اجتماعی دستهٔ پرندگان و ماهی‌ها الهام گرفته شده است [۴ و ۵]. این الگوریتم به‌دلیل سادگی پیاده‌سازی و فهم آسان، جزء روش‌های مؤثر حل مسائل بهینه‌سازی خیلی پیچیده به شمار می‌رود؛ اما به‌رغم تأثیری که در حل مسائل بهینه‌سازی دارد،

✧ نوع مقاله: پژوهشی
* نویسنده مسئول
پست‌های الکترونیک: psdnmpa@gmail.com (اکبرپور)
k.khamforoosh@iausdj.ac.ir (خام‌فروش)
maihami@iausdj.ac.ir (میهمی)

1. Swarm Intelligence

نحوه ارجاع به مقاله: اکبرپور، محمد پویا، خام‌فروش، کیهان، میهمی، وفا، «روشی برای بهبود الگوریتم بهینه‌سازی اجتماع ذرات با استفاده از CUDA بر روی پردازنده‌گرافیکی»، مجله محاسبات نرم، جلد ۸، شماره ۲، ص۲ـ۲۱، پاییز و زمستان ۱۳۹۸.

ممکن است برای حل مسائل پیچیده با مقیاس بزرگ، با مشکلاتی چون زمان صرف‌شده برای حل مسئله مواجه شود و استفاده از آن ناکارآمد و غیرقابل انجام باشد [۶].

از سوی دیگر، انقلاب پردازنده‌های چندین هسته‌ای[1]، جامعهٔ علمی را به‌سمت راه حل ناهمگن[2] سوق داده است و محاسبات ناهمگن[3] وارد عرصهٔ اصلی محاسبات شده‌اند [۷ و ۸]. محاسبات ناهمگن به محاسباتی گفته می‌شود که که بیشتر از یک نوع پردازنده برای انجام محاسبه استفاده شود [۹]. یکی از مهم‌ترین راه حل‌های ناهمگن، انجام محاسبات بر روی پردازنده‌گرافیکی[4] همه‌منظوره است [۱۰]. در دههٔ اخیر، استفاده از این پردازنده برای انجام محاسبات، رشد فراوان و چشمگیری داشته است.

NVIDIA برای پیاده‌سازی برنامه‌ها و انجام عملیات محاسباتی بر روی پردازنده‌گرافیکی، معماری به نام CUDA را برای برنامه‌نویسی پردازنده‌گرافیکی معرفی کرده است. به این روش جدید برنامه‌نویسی پردازنده‌گرافیکی، GPU-Computing می‌گویند. متخصصان و محققان با استفاده از زبان‌های برنامه‌نویسی سطح بالا و با توسعهٔ برنامه‌های مبتنی بر معماری CUDA، دامنهٔ وسیعی از قابلیت‌های پردازنده‌گرافیکی را شناسایی کرده‌اند [۸] و این قابلیت‌ها را در برنامه‌هایی نظیر محاسبات مالی، محاسبات دینامیکی فلوید [۸ و ۱۱]، اجرای محاسبات سنگین [۱۱ و ۱۲]، پردازش زمین‌لرزه [۱۳]، شبیه‌سازی بیوشیمی [۱۴]، مدل‌سازی وضعیت اقلیمی و پیش‌بینی وضع آب‌وهوا [۱۵]، پردازش تصویر [۱۶] و داده‌کاوی [۱۷] مشاهده کنند.

با این پیشرفت، جنبهٔ کاربردی الگوریتم‌های فراابتکاری موازی در حوزه‌های علمی، تجاری و صنعتی بیشتر شده است [۱۸ و ۱۹] و با فراگیر شدن پردازنده‌های‌گرافیکی و پشتیبانی معماری CUDA توسط تعداد کثیری از پردازنده‌های‌گرافیکی [۲۰ و ۲۱]، پیدا کردن بستر مناسب برای اجرای الگوریتم‌های موازی‌شده مبتنی بر پردازنده‌گرافیکی فراهم شده است [۱۲]. در نتیجه با موازی‌سازی الگوریتم‌های هوش اجتماعی با استفاده از این معماری، امکان رسیدن به تسریع قابل قبول ممکن، و در نهایت امکان حل مسائل متعدد محاسباتی سنگین و پیچیده با مقیاس بزرگ میسر شده است.

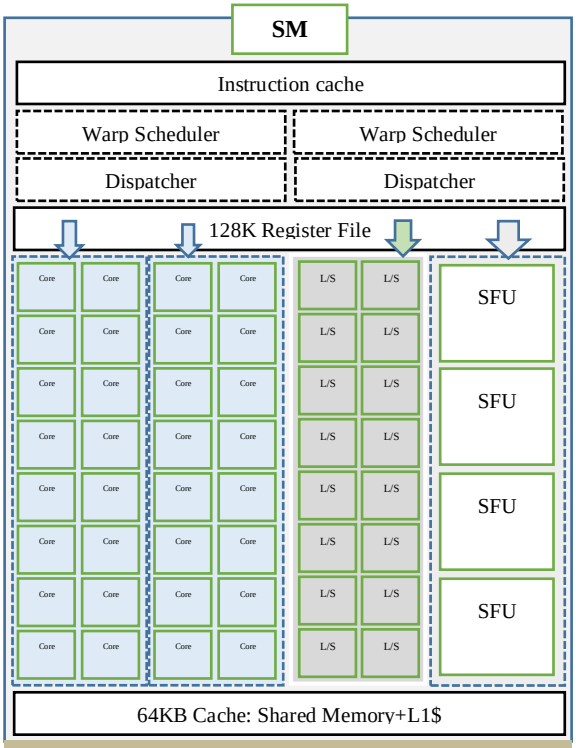
به‌طور کلی، پردازنده‌های گرافیکی محصول NVIDIA بر مبنای کاربرد آن در چهار ردهٔ زیر طبقه‌بندی شده‌اند [۱۰]:

- Tegra: در گوشی و تبلت استفاده شده است.
- GeForce: برای انجام کارهای عمومی و عادی.
- Quadro: برای انجام کارهای حرفه‌ای و انیمیشن‌سازی.
- Tesla: برای انجام کارهای محاسباتی و موازی.

پردازنده‌های‌گرافیکی NVIDIA، در ابتدا با هدف انجام محاسبات طراحی نشدند و تلاش برای بهره‌گیری غیرگرافیکی از این پردازنده‌ها، منجر به معرفی اولین نسل معماری محاسباتی به نام Fermi در سال ۲۰۱۰ میلادی شد که از اجزای کلیدی آن [۸ و ۲۲]، هسته‌های CUDA، حافظهٔ اشتراکی و حافظهٔ نهان سطح ۱، فایلثبات، واحدهای ذخیره/بازیابی[5]، واحدهای تابعی خاص[6] و زمانبند وارپ[7] می‌باشد که در شکل (۱) آورده شده است [۲۲].

به‌طور کلی، در هر پردازنده‌گرافیکی، چند SM وجود دارد [۲۲]. هر SM[8] برای اجرای همروند صدها نخ موازی طراحی شده است. هنگام صدا زدن کرنل[9]، که قطعهٔ کد موازی برنامه است، کرنل به‌صورت بلوک‌های نخ بین SMهای موجود توزیع می‌شوند. یک وارپ، دستهٔ ۳۲تایی از نخ‌هاست. همهٔ نخ‌های یک وارپ، همان دستورات را در واحد زمان اجرا می‌کنند. هرنخ وارپ، شمارندهٔ آدرسی دستور، وضعیت ثبات، کری‌های خروجی و دستورات جاری بر روی داده‌های خودش را دارد [۲۲ و ۲۳]. هر SM دارای دو زمانبند وارپ و دو واحد مجزای ارسال دستورات است که امکان اجرای همروند دو وارپ را فراهم می‌کند [۸ و ۲۲].

1. Many-Core
2. Heterogeneous
3. Heterogeneous Computing
4. GPU
5. Load/Store Units
6. Special Function Units
7. WARP scheduler
8. Streaming Multiprocessor
9. Kernel



شکل (۱): یک SM در معماری Fermi [۲۲ و ۲۳]

زمانبند دوگانهٔ وارپ، دو وارپ را انتخاب می‌کند و یک دستور از هر وارپ، به گروه ۱۶ هسته‌ای از هسته‌ها، ۱۶ واحد ذخیره/بازیابی یا چهار واحد SFU داده می‌شود [۸ و ۲۲]. چون وارپ‌ها به‌صورت مستقل و مجزا اجرا می‌شوند، زمانبند نیازی به بررسی وابستگی‌های داخلی دستورات ندارد. حافظهٔ اشتراکی و ثبات‌ها منابع باارزش SM هستند. نخ‌های یک بلوک، از طریق این منابع می‌توانند با همدیگر ارتباط برقرار کرده و همگام شوند. هر SM، ۱۶ واحد ذخیره/ بازیابی دارد که قابلیت محاسبهٔ آدرس‌های منبع و مقصد را برای ۱۶ نخ در هر کلاک فراهم می‌کند. واحدهای ذخیره و بازیابی، دسترسی به داده‌ها را برای حافظهٔ نهان و رم پشتیبانی می‌کند. واحدهای تابعی خاص، دستورات غیر جبری مانند Sin، Cos، جذر و معکوس کسر را اجرا می‌کنند. در یک کلاک، هر SFU، یک دستور از هر نخ را اجرا می‌کند یک وارپ در مدت ۸ کلاک اجرا می شود.

قابلیت یک پردازنده‌گرافیکی بر اساس دو ویژگی مهم تعداد هسته‌های CUDA[1] و اندازهٔ حافظه[2] ارزیابی می‌شود [۱۰ و ۲۲]. بر همین اساس، دو معیار مختلف برای توصیف کارایی پردازنده‌گرافیکی وجود دارد [۱۰، ۲۲ و ۲۳]:

- اوج کارایی محاسباتی[3] (gflops[4]): معیار اندازه‌گیری توان محاسباتی با واحد تعداد عملیات ممیز شناور در واحد زمان.
- پهنای باند حافظه[5] (GB/s): حجم داده‌های خوانده از حافظه، یا نوشته‌شده به حافظه، نسبت به زمان.

Nvidia برای توصیف قابلیت پردازنده‌گرافیکی، قابلیت محاسباتی[6] را تعریف کرده است [۱۰]. برای مثال در معماری Fermi، عدد قابلیت ۲ است.

در جدول (۱) خلاصه‌ای از ویژگی‌های دو معماری Pascal و Fermi به همراه قابلیت محاسباتی و همچنین ویژگی‌های پردازنده‌گرافیکی GeForce GT 525M مورد استفاده در این مقاله، آورده شده است [۲۴]:

جدول (۱): مقایسهٔ ویژگی‌های پردازنده‌گرافیکی [۲۴]

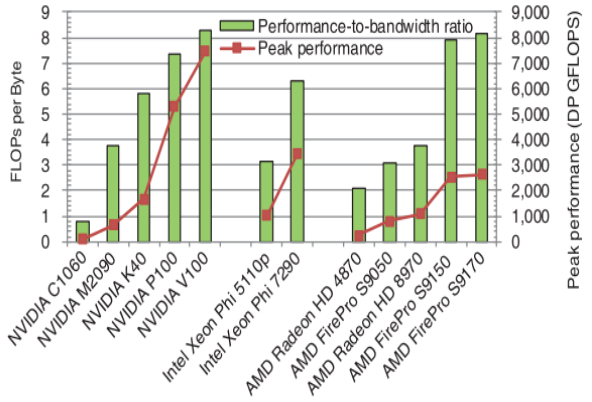
	Geforce GT 525m	Geforce GT 1050Ti
Architecture	Fermi	Pascal
GPU	GF108	GP107
CUDA cores	96	768
Memory	1 GB	4 GB
Peak performance (double)	19.20 Gflops	66.82 Gflops
Bandwith	28.80 GB/s	112.1 GB/s
Compute capability	2.1	6.1
Multiprocessors	2	6
GPU clock	600 MHz	1291 MHz
Memory data rate	1800 MHz	7008 MHz

پهنای باند حافظه، عامل مهمی برای بیشینه‌سازی سرعت GPU نسبت به CPU است. برای مثال، در سیستمی که از حافظهٔ رم DDR3 با باس ۱۶۰۰ استفاده می‌کند، پهنای باند تئوری احتسابی، ۱۲/۸ گیگابایت در ثانیه است و این در حالی است که پهنای باند متناظر با Geforce GT 525M، ۲۸/۸ گیگابایت در ثانیه است. در مقالهٔ [۲۵]، مقایسه‌ای از نسبت کارایی به پهنای باند حافظه و اوج کارایی پردازنده‌گرافیکی

1. Number of CUDA cores

2. Memory size
3. Peak computational performance
4. Billion floating-point operations per second
5. Memory Bandwidth
6. Compute Capability

نسبت به یکدیگر و CPU صورت گرفته که در شکل (۲) نشان داده شده است.



شکل (۲): مقایسهٔ اوج کارایی در پردازنده‌های گرافیکی با CPU [۲۵]

CUDA یک معماری سخت‌افزاری و نرم‌افزاری است. این معماری، امکان برنامه‌ریزی پردازنده‌گرافیکی را برای برنامه‌نویسان با استفاده از زبان برنامه‌نویسی C، C++، Fortran، OpenCL، DirectCompute، فراهم می‌کند.CUDA بستری برای انجام محاسبات ناهمگن است؛ یعنی هنگام انجام محاسبات، بخش ترتیبی کدها بر روی CPU اجرا می‌شود و بخش موازی آن، به‌صورت موازی بر روی GPU اجرا می‌شود. یک برنامهٔ CUDA، از کرنل‌هایی تشکیل شده است که بر روی دستگاه پردازنده‌گرافیکی اجرا می‌شوند [۲۶]. کرنل جزء کلیدی مدل برنامه‌نویسی CUDA است. با صدا زدن کرنل‌های برنامه، کرنل به‌صورت موازی بر روی نخ‌ها اجرا می‌شوند. بلوک نخ، مجموعه‌ای از نخ‌های همروند است که به‌صورت موازی اجرا می‌شوند. نخ‌ها از طریق حافظهٔ اشتراکی و روش همگام‌سازی مانعی با همدیگر همگام می‌شوند. نخ‌های بلوک‌های مختلف، متفاوت از هم بوده و هر نخ، شناسهٔ معینی داشته و این نخ‌ها با هم ارتباط ندارند. نخ‌ها با دو مختصات منحصربه‌فرد، از سایر نخ‌های دیگر متمایز می‌شوند:

- blockIdx: اندیس بلوک داخل گرید
- threadIdx: اندیس نخ داخل بلوک

بر مبنای مختصات می‌توانیم تکه‌هایی از داده‌ها را به نخ‌های مختلف منتصب کنیم. یک گرید، آرایه‌ای از بلوک‌های نخ است که همان کرنل را اجرا می کنند [۲۷]. گرید، همگام‌سازی میان کرنل‌های وابسته را انجام می‌دهد. CUDA یک مدل برنامه‌نویسی مقیاس‌پذیر است؛ به این معنی که برنامه‌ها را بر روی پردازنده‌های‌گرافیکی مختلف، که معماری متفاوتی دارند، اجرا می‌کند. مقیاس‌پذیری، یکی از ویژگی‌های مهم هر برنامهٔ موازی است. مقیاس‌پذیری نشان می‌دهد که با اضافه کردن منابع سخت‌افزاری به یک برنامهٔ موازی سرعت به نسبت مقدار اضافه‌شده افزایش می‌یابد. برای مثال، مدت زمان اجرا یک برنامهٔ CUDA بر روی دو SM نسبت به اجرای آن بر روی یک SM قابل قیاس است.

در بخش دوم این مقاله، الگوریتم بهینه‌سازی اجتماع ذرات و مطالعاتی که در این زمینه انجام گرفته است به‌صورت مختصر تشریح می‌شود. در بخش سوم، تکنیک‌ها و روش پیشنهادی موازی‌سازی الگوریتم بهینه‌سازی اجتماع ذرات بر روی GPU بیان می‌شود. در بخش چهارم، ارزیابی روش پیشنهادی ارائه می‌شود و در بخش پنجم، نتیجه‌گیری ارائه می‌گردد.

۲. مطالعات و پیشینه

الگوریتم بهینه‌سازی اجتماع ذرات، شبیه‌سازی رفتار دستهٔ پرندگان است [۲۸]. در این الگوریتم، هر راه حل یک پرنده در فضای جست‌وجوست که به آن ذره گفته می‌شود. همهٔ ذرات با استفاده از تابع برازندگی مورد ارزیابی قرار می‌گیرند. ذرات در فضای مسئله پرواز کرده و ذرات بهینهٔ فعلی را دنبال می‌کنند.

ذرات در فضای مسئله پخش می‌شوند و هر ذره به‌عنوان یک نقطه در یک فضای d بُعدی رفتار می‌کند. ذرهٔ i ام به‌صورت زیر نشان داده می‌شود:

$$x_i = (x_{i1}, x_{i2}, \dots, x_{id})$$

نرخ تغییر مکان یعنی سرعت پرواز، ذرات را هدایت می‌کنند. سرعت برای ذرهٔ i ام به‌صورت زیر نشان داده می‌شود:

$$v_i = (v_{i1}, v_{i2}, \dots, v_{id})$$

بهترین قبلی ذرهٔ i ام در P_{best} ذخیره شده و موقعیت به‌صورت زیر نمایش داده می‌شود:

$$p_i = (p_{i1}, p_{i2}, \dots, p_{id})$$

بعد از پیدا کردن بهترین ذره، سایر ذرات، سرعت و مکان

خـود را بـا اسـتفاده از تسـاوی‌هـای ذیـل کـه بـه‌ترتیـب در فرمول‌های (۱) و (۲) آمده است، بروزرسانی می‌کنند:

$$v_{id} = w_i v_{id} + C_1 R_l (p_{id} - x_{id}) + C_2 R_g (p_{gd} - x_{id}) \quad (۱)$$

$$x_{id} = x_{id} + v_{id} \quad (۲)$$

که در آن، x_{id} موقعیت ذره، v_{id} سرعت، w_i فاکتور وزن، d بُعد بین ابعاد ($1 \le d \le n$) است. و C_1 و C_2 اعداد ثابت مثبت بوده و فاکتورهای یادگیری هستند که معمولاً برابـر ۲ در نظـر گرفتـه می‌شوند [۲۹]. R_l و R_g اعداد تصادفی حقیقی بین بـازه (۱و۰) هستند. و P_i بهترین موقعیت قبلی ذرهٔ i ام است. اندیس بهترین موقعیت در بین تمـام ذرات جمعیـت بـا g و انـدیس بهتـرین موقعیت در بین همهٔ ذرات همسایه ۳ با l نشان داده شده است [۳۰].

سرعت ذرات در هر بُعـد بـه بیشـینهٔ سـرعت یعنـی V_{max} بستگی دارد [۳۰]. اگر مجموع سرعت‌ها باعث شود که سرعت آن بُعد فراتر از V_{max} رود، با یک پـارامتر مشـخص شـده کـه توسط کاربر از قبل تنظیم شده است سرعت آن بُعـد بـه V_{max} محدود می‌شود [۳۰]. حداکثر تعداد دفعات تکرار برای اجرای الگوریتم بهینه‌سازی اجتمـاع ذرات و حـداقل الزامـات خطـا به‌وسیلهٔ شرط توقف بررسی می‌شود. فاکتور وزن برای بهینه‌تر کردن جست‌وجو معرفی شد و معمولاً وزن در محدوده‌ای بین ۱/۲ و ۰/۸ انتخاب می‌شود تا شانس پیدا کردن بهینهٔ سراسری افزایش یابد و تعداد دفعات شکست کاهش پیدا کند [۳۱]. شبه کد الگوریتم بهینه‌سازی اجتماع ذرات در الگـوریتم (۱) آورده شده است [۳۲].

```
For each particle
   Initialize particle
End For
For i=1 to it_Max
   For each particle x in X
      Fx=Calculate fitness value for X
      If Fx is better than FlBest in history then
        lBest=x //Set current value as the new lBest
      End if
   End For
   Choose the particle with the best fitness value of all the
 particles as the PgBest
gBest= best x in X
   For each particle Xi in X
      V=WV+C1 Rl (lBest -X )+C2 Rg (gBest -X )
      X=X+V
   End For
End For
```

الگوریتم (۱): شبه کد الگوریتم بهینه‌سازی اجتماع ذرات [۳۲]

الگوریتم‌های فراابتکاری برای حل انواع مسائل چندجمله‌ای NP مختلف توسعه داده شده‌اند. تـلاش محققـان بـرای بهبـود عملکرد این الگوریتم‌ها به‌منظور رسیدن به تسـریع بـالا، تـوان عملیاتی بالا، کیفیت راه‌حل، بهره‌برداری مؤثر از منابع موجـود برای حل مسائل محاسباتی سنگین بوده اسـت [۳۳ و ۳۴]. بـر اساس مطالعات و پژوهش‌هایی که در زمینهٔ بهبود اجرای PSO بر روی پردازنده‌گرافیکی انجام گرفته، موازی‌سـازی الگـوریتم بهینه‌سازی اجتماع ذرات بـه دو روش همگـن و یـا نـاهمگن امکان‌پذیر است. روش‌ها و استراتژی‌های مختلفی برای بهبـود پیشنهاد شده است [۳۵ و ۳۶]:

- بیشینه‌سازی سطح موازی به‌منظور بیشینه‌سازی اسـتفاده از منابع پردازشی؛
- کمینه‌سازی مبادلهٔ داده بین میزبان و دستگاه؛
- استفاده از حافظهٔ اشتراکی و حافظه‌های نهان؛
- استفادهٔ بهینه از حافظه برای افزایش سطح موازی؛
- بهینه‌سازی دستورات برای افزایش سطح موازی؛
- تنظیم و مدیریت نخ‌ها؛
- بیشینه‌سازی پهنای باند با استفاده از حافظهٔ اشتراکی؛
- بیشینه‌سازی استفاده از حافظه برای برنامه.

در مقالهٔ [۳۷]، مسئلهٔ تشخیص علائم راهنمایی و رانندگی را با استفاده از الگوریتم بهینه‌سازی اجتماع ذرات پیاده‌سازی‌شده بر روی پردازنده‌گرافیکی حل کرده و تسـریع بیسـت برابـری نسبت به مدل CPU را مشاهده کردند. علت عالی بودن شتاب گزارش‌شده، استفاده از یک پردازنده‌گرافیکی پرقـدرت بـا ۱۴ SM است درحالی‌که در این مقاله با روش پیشنهادی ارائه‌شده و با استفاده از ۲ SM و فقط ۹۶ هستهٔ پردازشی به تسریع عـالی رسیده است.

در مقالهٔ [۳۸]، یک روش موازی برای پیاده‌سازی الگوریتم PSO بر روی پردازنده‌گرافیکی ارائه کردند و با ارزیـابی آن بـا سه تابع آزمون مشاهده کردند که الگوریتم پیاده‌سازی‌شـده بـر روی پردازنده‌گرافیکی بیش از ۱۰ برابر سـریع‌تـر از الگـوریتم پیاده‌سازی‌شده بر روی CPU اجرا می‌شود.

در مقالهٔ [۳۹]، الگوریتم بهینه‌سازی اجتماع ذرات با استفاده

از زبان برنامه‌نویسی C، CUDA-C و Matlab پیاده‌سازی شد و آن‌ها را با ۶ تابع آزمون ارزیابی کردند و نتایج نشان داد که الگوریتم پیاده‌سازی‌شده با زبان CUDA-C به‌مراتب سریع‌تر و با عملکرد بهتری نسبت به برنامهٔ پیاده‌سازی‌شده بر روی CPU اجرا می‌شود.

در مقالهٔ [۴۰]، روشی برای انتخاب باند تصاویر طیفی مبتنی بر الگوریتم بهینه‌سازی اجتماع ذرات بر روی پردازنده‌گرافیکی پیاده‌سازی شد و مشاهده کردند که روش ارائه‌شده مبتنی بر PSO به‌مراتب محاسبات را بهبود بخشیده و عملکرد بهتری نسبت به PSA دارد.

در مقالهٔ [۴۱]، الگوریتم جست‌وجوی الگو، مبتنی بر الگوریتم بهینه‌سازی اجتماع ذرات بر روی پردازنده‌گرافیکی پیاده‌سازی شد و مشاهده کردند که روش ارائه‌شده سرعت اجرا و عملکرد بهتری نسبت به مدل CPU دارد.

در مقالهٔ [۴۲]، تأثیر کیفیت اعداد تصادفی را بر الگوریتم PSO پیاده‌سازی‌شده بر روی پردازنده‌گرافیکی بررسی شد و مشاهده کردند که اجرای موازی بدون‌همگام‌سازی بسیار سریع‌تر از اجرای موازی با همگام‌سازی است.

در مقالهٔ [۴۳]، الگوریتم MOPSO را با استفاده از OpenCL و CUDA بر روی پردازنده‌گرافیکی پیاده‌سازی شد و مشاهده کردند زمان محاسبات الگوریتم پیاده‌سازی‌شده با استفاده از CUDA و OpenCL بر روی پردازنده‌گرافیکی به‌طور چشمگیری کاهش می‌یابد.

در مقالهٔ [۴۴]، مسئلهٔ Schwefel را با استفاده از الگوریتم بهینه‌سازی اجتماع ذرات بر روی پردازنده‌گرافیکی، پیاده‌سازی و حل کردند. نتایج حاصل از اجرا، بهبود و رسیدن به عملکرد بسیار بالا در پردازنده‌گرافیکی را نشان می‌دهد.

در مقالهٔ [۴۵]، الگوریتم بهینه‌سازی اجتماع ذرات اقلیدسی بر روی پردازنده‌گرافیکی پیاده‌سازی شد و با استفاده از پنج تابع آزمون آن را مورد ارزیابی قرار دادند و مشاهده کردند که مدل پیاده‌سازی‌شده بر روی پردازنده‌گرافیکی، مسئله را در مدت‌زمان کوتاه‌تری اجرا کرده و تسریع ۱۶/۲۷ برابری را نسبت به برنامهٔ پیاده‌سازی‌شده بر روی CPU ارائه می‌کند.

در مقالهٔ [۴۶]، الگوریتم اجتماع ذرات چند کانالی بر روی پردازنده‌گرافیکی پیاده‌سازی و مدت‌زمان اجرا با استفاده از چهار تابع آزمون ارزیابی شد و مشاهده کردند که با افزایش تدریجی جمعیت ذرات، سرعت افزایش می‌یابد و مدت‌زمان اجرای الگوریتمی که بر روی پردازنده‌گرافیکی پیاده‌سازی شده، به‌مراتب کمتر از نگارش CPU آن است.

در مقالهٔ [۴۷]، الگوریتم دسته‌بندی اسناد مبتنی بر الگوریتم بهینه‌سازی اجتماع ذرات بر روی پردازنده‌گرافیکی پیاده‌سازی شد و پس از مقایسهٔ نتایج حاصل از اجرا بر روی یک CPU، یک GPU و دو GPU مشاهده کرند که سرعت اجرای بر روی دو GPU تقریباً دو برابر سرعت اجرا بر روی یک GPU است.

در مقالهٔ [۴۸]، سطح انتخابی فرکانس را با استفاده از الگوریتم بهینه‌سازی اجتماع ذرات بر روی پردازنده‌گرافیکی پیاده‌سازی شد و مشاهده کردند که با استفاده از GPU، این الگوریتم ۱۰۰ برابر سریع‌تر از مدل CPU اجرا می‌شود.

در مقالهٔ [۴۹]، مدیریت پورتفولیو را با استفاده از الگوریتم بهینه‌سازی اجتماع ذرات بر روی پردازنده‌گرافیکی پیاده‌سازی و معرفی شد و مشاهده گردید که مدیریت پورتفولیو پیاده‌سازی‌شده بر روی پردازنده‌گرافیکی تسریع ۴۰ برابری نسبت به روش پیاده‌سازی‌شده بر روی CPU دارد.

در مقالهٔ [۵۰]، الگوریتم بهینه سازی اجتماع ذرات تطبیقی را برای حل مسائلی با ابعاد بزرگ بر روی پردازنده‌گرافیکی پیاده‌سازی شد. نتایج حاصل از آزمون با چهار تابع نشان داد که پردازنده‌گرافیکی برای مسائلی با ابعاد بزرگ عملکرد بهتری نسبت به CPU دارد و تسریع ۸۵ برابری نسبت به مدل پیاده‌سازی‌شده بر روی CPU دارد.

در مقالهٔ [۵۱] با استفاده از استراتژی shuffle روشی برای پیاده‌سازی الگوریتم بهینه‌سازی اجتماع ذرات بر روی پردازنده‌گرافیکی ارائه شد و با انجام آزمایش‌هایی با استفاده از سه تابع مختلف تسریع ۶۲ برابری را برای تابع Rastringin نسبت به CPU مشاهده کردند.

در مقالهٔ [۵۲]، مروری بر پیاده‌سازی الگوریتم‌های هوش اجتماعی بر روی پردازنده‌گرافیکی انجام گرفت و روشی

جدید برای ارزیابی عملکرد ارائه شد.

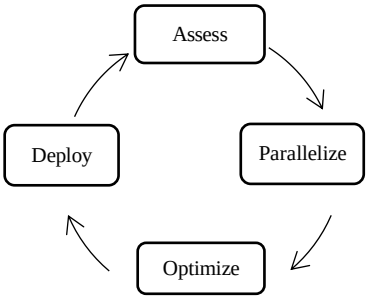
با وجود تمام تلاش‌های محققان در این زمینه، متأسفانه در اکثر مطالعات، از پردازنده‌های گرافیکی پرقدرت محاسباتی استفاده شده است که شرط کافی برای رسیدن به بیشینهٔ سرعت در سایر پردازنده‌های گرفیکی نیست و ممکن است زمان قابل قبولی در سایر پردازنده‌ها مشاهده نشود و در نتیجه بازدهی لازم را نداشته باشد و دلیل دیگر آن سربار تولید عدد تصادفی و مواجه شدن با سرریز حافظه به‌دلیل بزرگ شدن مقیاس مسئله و سنگین شدن آن در محاسبات سنگین است. با توجه به نتیجه و تحلیل انجام‌گرفتهٔ برتری که این روش دارد به‌اختصار به شرح ذیل است:

- پویایی، مقیاس‌پذیری، بازدهی و انعطاف‌پذیری بیشتر کدهای برنامه بر مبنای معماری‌های مختلف و قابلیت‌های محاسباتی متفاوت؛
- کاهش مشغولیت پردازنده‌گرافیکی به‌هنگام تولید اعداد تصادفی؛
- ارائهٔ مدل جدید چندکرنلی تغییریافته؛
- افزایش سطح موازی‌سازی دستورات و نخ‌های سطح بلوک از طریق Reduce؛
- کاهش انشعابات از طریق پیاده‌سازی دستوارت PTX؛
- کاهش سطح وابستگی با متحمل شدن سربار تعریف حافظه‌ای به‌اندازهٔ جمعیت و افزایش سطح موازی‌سازی کرنل ارزیابی؛
- کاهش مبادلهٔ داده بین CPU و GPU از طریق ادغام کرنل‌ها.

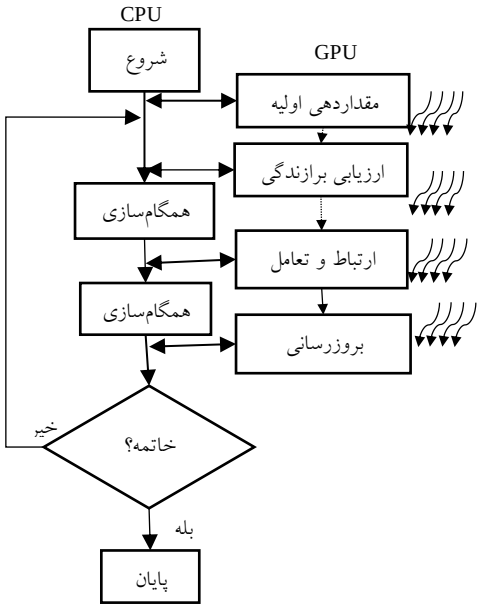
۳. روش پیشنهادی

در این بخش، مدل، نحوهٔ برنامه‌نویسی موازی و پیاده‌سازی برنامهٔ PSO با استفاده از CUDA بر روی پردازنده‌گرافیکی، و استراتژی‌های به کار گرفته‌شده برای بهبود آن می‌پردازد. برای انجام فرایند پیاده‌سازی از شیوهٔ چهارمرحله‌ای APOD بهره گرفته شده است [۵۳].

همان طور که در شکل (۳) دیده می‌شود در APOD، بهینه‌سازی برنامه، یک فرایند تکراری است و لازم نیست که زمان زیادی را صرف تمام استراتژی‌های بهینه‌سازی احتمالی قبل از مشاهدهٔ تسریع مناسب کرد. در عوض، برنامه می‌تواند با استفاده ازاستراتژی‌های اعمالی به‌صورت پیوسته و مرحله به مرحله با استفاده از مراحل ارزیابی، موازی‌سازی، بهینه‌سازی و توسعه تکمیل شود. توسعهٔ نهایی بر مبنای APOD، با در نظر گرفتن مقیاس‌پذیری و قابلیت محاسباتی منجر به ارائهٔ مدل جدید پیاده‌سازی موازی الگوریتم بهینه‌سازی اجتماع ذرات بر مبنای مدل چند کرنلی پایه، مطابق شکل (۴) این مقاله شده است.



شکل (۳): شیوهٔ APOD [۵۳]



شکل (۴): مدل پیاده‌سازی چندکرنلی الگوریتم PSO

در این مدل، ابتدا ذرات به‌صورت موازی مقداردهی اولیه می‌شوند با توجه به اینکه ذرات باید به‌صورت تصادفی در محدودهٔ تعیین‌شده بر روی GPU مقداردهی اولیه شوند، برای تولید اعداد تصادفی در اولین گام از توابع کتابخانه‌های موجود

در CUDA یعنی curand به‌صورت بهینه با استفاده از روش‌های پیشنهادی بهره گرفته شد. سپس به‌دلیل استفاده از روش APOD و بهبود عملکرد نهایی curand بر اساس مقالات [۳۷، ۳۹ و ۵۱] به‌منظور توسعه و بهبود عملکرد برنامه، تغییراتی در روش تولید عدد تصادفی داده شد که در بخش مربوط آورده شده است. عملیات مربوط به ارزیابی به‌صورت موازی انجام می‌گیرد که برای کاهش وابستگی و افزایش سطح موازی دستورات برنامه متحمل سربار حافظه اضافی برای نگهداری نتیجهٔ ارزیابی برای هر ذره و به اندازهٔ تمام ذرات شده است. عملیات مربوط به بروزرسانی سرعت، موقعیت و بهترین موقعیت تمام ذرات به‌صورت موازی اجرا می‌شوند. بهبود این عملیات‌ها با استفاده از Reduce و Unrolling صورت گرفته است. با توجه به اینکه بروزرسانی موقعیت وسرعت وابسته به بهترین ذرهٔ محلی و سراسری است و همهٔ ذرات از آن تأثیر می‌پذیرند، در این مرحله همگام‌سازی کرنل‌های وابسته انجام می‌گیرد. شبه کد عملیات مربوط به بروزرسانی و بیشینهٔ دفعات تکرار به‌عنوان شرط خاتمهٔ فرایند بهینه‌سازی محسوب می‌شود.

۳. ۱. نگاشت و پیاده‌سازی

بهبودهای انجام‌گرفته در پیاده‌سازی هر بخش، توسط نویسندگان مقاله به‌صورت منحصربه‌فردی انجام شده است. متغیرهای تعریف‌شده بر روی پردازنده‌گرافیکی به‌صورت زیر تعریف شده‌اند:

N_X تعداد ذرات و N_Y تعداد ابعاد مسئله و دامنهٔ تابع برازندگی در محدودهٔ [Min, Max] است. آرایه‌هایی که اطلاعات ذرات را نگهداری می‌کنند عبارت‌اند از:

- gd_xx [$N_X \times N_Y$]
- gd_vx [$N_X \times N_Y$]
- gd_pbestx [$N_X \times N_Y$]
- gd_pbest [N_X]
- gd_minval [N_X]

علت استفاده از آرایه‌های یک‌بُعدی، ساختار حافظهٔ سراسری GPU است که فقط آرایه‌های یک‌بعدی را می‌پذیرد. بنابراین برای نگاشت همهٔ نخ‌ها به $N_X \times N_Y$ دادهٔ موجود، باید فرمول‌های (۳) و (۴) برقرار باشد:

$$S_1 \times S_2 \times T_1 \times T_2 = N_X \times N_Y \quad (۳)$$

که در آن، $S_1 \times S_2$ اندازهٔ بلوک و $T_1 \times T_2$ اندازهٔ گرید است. بنابراین فرمول (۳) مجموع تعداد نخ‌های گرید را محاسبه می‌کند؛ به این معنی که تضمین می‌کند ذرات ما به‌صورت موازی بارگذاری و پردازش شوند.

$$I = (B_Y \times T_2 + B_X)\, S_1 \times S_2 + T_Y \times S_2 + T_X \quad (۴)$$

که در آن، I اندیس عنصر آرایه، (T_X, T_Y) اندیس نخ‌های داخل بلوک و (B_X, B_Y) اندیس بلوک‌های گرید است. بنابراین نگاشت I امین عنصر آرایه از طریق فرمول (۴) صورت می‌گیرد.

به این ترتیب، تمام نخ‌های یک کرنل، به‌صورت یک به یک به N داده، نگاشت می‌شود. بنابراین اعمال یک عمل بر روی یک نخ، باعث می‌شود که تمامی N نخ دقیقاً همان عمل را به‌صورت همزمان انجام دهند.

۳. ۲. کاهش انشعابات

انشعابات داخل برنامه باعث کاهش سطح موازی‌سازی برنامه و کُند شدن سرعت اجرای آن بر روی پردازنده‌گرافیکی می‌شوند [۵۴ و ۵۵]. با تعریف دستورات پیش‌پردازه‌ای به‌صورت الگوریتم (۲) بر روی پردازنده‌گرافیکی انشعابات مربوط به سطح دستورات تا حد ممکن کاهش پیدا کند.

```
//Define Macro for Calculate Minimum
1.  #define MIN(x,y) ((x < y) ? x : y)
//Define Macro for Invert Random
2.  #define CheckRandom(x,y,z) ((x > y) ? -z : z)
//Define Macro for Calculat Best in Minimum
3.  #define MinumBest(x,y,z,m) ((x < y) ? z : m)
//Define Macro for Calculated Minimum is true?
4.  #define MINBOOL(x,y) ((x < y) ? true : false)
//Define Macro for Calculate Maximum
5.  #define MAX(x,y) ((x > y) ? x : y)
// Define Macro for Calculate Minimum Thread
6.  #define MIN_IDX(x,y, idx_x, idx_y) ((x < y) ? idx_x : idx_y)
//Define Macro for Calculate Maximum Thread
7.  #define MAX_IDX(x,y, idx_x, idx_y) ((x > y) ? idx_x :
//Define Macro for Calculate Minimum Thread Value
8.  #define IFASSIGN(x,y, val_x, val_v) ((x > y) ? val_x : val_v)
```

الگوریتم (۲): دستورات پیش‌پردازه‌ای مربوط به کاهش انشعابات

۳. ۳. تکنیک Unrolling Loops

با استفاده از تکنیک Unrolling Loops حلقه‌ها بهینه شده است [۵۵]. با استفاده از تکرار دستورات تعداد گام‌ها و انشعابات مرتبط با حلقه‌ها کاهش یافته است. این تکنیک باعث می‌شود که تعداد دستورات مستقل با استفاده از تکرار آن‌ها در بدنهٔ حلقه افزایش یافته و از طریق کاهش سطح وابستگی، باعث افزایش همزمانی عملیات حافظه مربوط به خواندن و نوشتن می‌شود. با افزایش سطح موازی‌سازی دستورات و تعداد عملیات موازی، پهنای باند حافظه بالا می‌رود. در نتیجه زمانبند وارپ، تعداد وارپ‌های بیشتری را زمانبندی کرده و به پنهان‌سازی تأخیر حافظه کمک می‌کند. همان طور که در الگوریتم (۳) دیده می‌شود، تعداد گام‌ها و انشعابات تصمیم‌گیری مربوط به حلقه for داخل کرنل ارزیابی با استفاده از تکرار دستورات بدنهٔ حلقه کاهش داده شده است. برای نگاشت بُعد i ام از فرمول (۵) استفاده شده است:

$$D_i = (T_i + (N_x \times i)) \qquad (۵)$$

که در آن، N_x تعداد ذرات و T_i نخ جاری و i اندیس جاری است.

Unrolling LOOPS
Input: *ix,nx, gd_xx, gd_pbestx*
//ix get thread id
//nx get number of particle
// gd_xx(X_i)(i=1,2,..,D) to get position value
//gd_pbestx(X_i)(i=1,2,..,D) to get best position value
//Find Data in Block via *ix+(i*nx)* that *ix* is thread ID
//Set the position value via *ix+((i+1)*nx)* that *ix* is thread ID
1. Set *i*=0 //set Default dimesion
2. **for** each dimension *i* **do** by *2* Step in each dimension
   a. *C= ix+(i*nx)//current dimension*
   b. *N= ix+((i+1)*nx)// Next dimension*
   c. *gd_pbestx[C]=gd_xx[C]*
   d. *gd_pbestx[N]=gd_xx[N]*
3. **end for**

Output: *gd_pbestx*

الگوریتم (۳): شبه کد کاهش انشعابات حلقه

۳. ۴. تکنیک Reducing with Unrolling

یکی از روش‌های بخش‌بندی داده‌ها، بخش‌بندی چرخه‌ای است [۱۰ و ۲۲]. به این معنی که تعداد داده‌های بیشتری با استفاده از یک نخ پردازش می‌شود. با استفاده از تکنیک Reducing with Unrolling می‌توان این کار را انجام داد؛ یعنی بیشتر از یک داده با استفاده از یک نخ پردازش شود.

در الگوریتم (۴) بخش ابتدایی کرنل پیدا کردن بهترین، که با این استراتژی بهینه‌سازی شده، آورده شده است:

Find Best Postion in parallel
Input: *ix,nx, gd_xx, gd_pbestx*
// unrolling warp and warp synchoronous
1. **if** *tid* <**32 then** //if *tid* smaller than warp
   a. **if** *blockSize* >= **64 then**
//Compare *wsMINIdx[tid]*,with Best *index of* Minimum value found
      I. *LastBestindex*=MIN_IDX(*wsMIN[tid+32], LastBestMin,wsMINIdx[tid+32], LastBestindex*)
//Compare *wsMIN[tid]* with Best Minimum Value found
      II. *LastBestMin*=MIN(*wsMIN[tid+32], LastBestMin*)
   b. **end if**
   c. **if** *blockSize* >= **32 then**
      III. *LastBestindex*=MIN_IDX(*wsMIN[tid+16], LastBestMin,wsMINIdx[tid+16], LastBestindex*)
      IV. *LastBestMin*=MIN(*wsMIN[tid+16], LastBestMin*)
   d. **end if**
   e. **…..**
2. **end if**
   …
   //write result for this block to global mem
Output: *BBlock, BestIndex, BestGlobalIndex*

الگوریتم (۴): شبه کد بهینه‌سازی Reducing with Unrolling

۳. ۵. تکنیک Reducing with Unrolled Warps

در معماری فرمی، هر وارپ، به دستهٔ ۳۲تایی از نخ‌ها گفته می‌شود. پس با اجرای هر دستور وارپ، همگام‌سازی به‌طور خودکار انجام می‌شود. بنابراین زمانی که تعداد نخ‌ها کمتر از تعداد نخ‌های وارپ یعنی ۳۲ است، نیازی به همگام‌سازی از طریق فراخوانی Syncthread نیست و می‌توان حلقه‌ها و انشعابات را با استفاده از این تکنیک کاهش داد. در الگوریتم (۵) با استفاده از این تکنیک، پیدا کردن بهترین مقدار بهینه، پیاده‌سازی شده است. هنگام برگشت نتایج ممکن است ترتیب اجرای دستورات تغییر کند و نتایج نامعتبری مشاهده شود. پس نوع حافظه به‌صورت volatile تعریف شده است. بنابراین هنگامی که متغیری از این نوع تعریف می‌شود به این معنا است که در هر زمان توسط سایر نخ‌ها این متغیر می‌تواند مقدارش تغییر کند و برای دستیابی مستقیم به حافظه است.

Find Best Postion in parallel
Initialize, set the "*block size*" and "*grid size*", with number of threads in a grid equaling to Swarm Size (*N*).
Input: *ix,nx, g_idata, g_idxs*
Transfer All Particle from global to shared memory
//*gridDim* is number of active threan in block
// set thread ID
1. *i = blockIdx.x * blockSize * 2 + threadIdx.x*
2. *gridSize = blockSize * gridDim.x * 2*
//*g_idata*(X_i)(i=1,2,..,*N*) to get temporary position value
// *g_idxs*(X_i)(i=1,2,..,*N*) to get temporary index of position value
//in-place reduction in global memory and unrolling 2 data blocks
3. **for** each block *i* **do** by *gridsize* **Step**
   a. **Assign** *index+blockSize* to *tid*
   b. **if ((*N* is Power of (2)) OR (*tid*)< *N*) then//boundary chek**
   //Compare *g_idxs[tid]*,with Best *index of* Minimum value found
   I. MIN_IDX(*g_idata[tid]*,*LastBestMin*,*g_idxs[tid]*,*LastBestiindex*)
   //Compare *g_idata [tid]*with Best Minimum Value found
   II. MIN(*g_idata [tid]*, *LastBestMin*)
   c. **end if**
4. **end for**
Output: *BBlock, BestIndex, BestGlobalIndex*

الگوریتم (۵): شبه کد بهینه‌سازی پیدا کردن بهترین با اعمال Reducing with Unrolled Warps

۳.۶. تکنیک Unrolling Kernel

این تکنیک، برای کاهش تعداد کرنل‌های الگوریتم و بهینه‌سازی و بهبود عملکرد الگوریتم پیاده‌سازی شده است. با استفاده از این تکنیک، کرنل مربوط به بروزرسانی موقعیت بر اساس بهترین محلی و بهترین سراسری در یک کرنل پیاده‌سازی می‌شود. الگوریتم (۶) بهبود کدهای کرنل مربوط به بروزرسانی موقعیت با استفاده از این تکنیک را نشان می‌دهد. برای نگاشت بهترین اندیس بلوک همسایه و بهترین اندیس سراسری از فرمول (۶) استفاده شده است:

$$BBlock = (T_i \bmod N_x) / (N_x / N_b) \qquad (۶)$$

که در آن، T_i شناسه نخ جاری و N_x تعداد ذرات و N_b تعداد بلوک‌ها گروه‌بندی شده است. بنابراین BBlock اندیس بلوک بهترین همسایه است.

۷.۶. استراتژی تولید اعداد تصادفی

برای بهبود عملکرد و افزایش سطح موازی‌سازی و کیفیت تولید اعداد تصادفی، از دو تابع تولید اعداد تصادفی متفاوت استفاده می‌شود. الگوریتم (۷) شبه کدهای پیاده‌سازی‌شده مربوط به یکی از توابع تولید اعداد تصادفی را نشان می‌دهد.

همان طور که پیداست، تولید اعداد تصادفی بر اساس شناسهٔ نخ (*tid*) داخل بلاک که مرتبط با دادهٔ متناظر و عدد قبلی تولید شده است می‌تواند منجر به تولید اعداد تصادفی مطابق با فرمول (۷) شود. این در حالی است که در کارهای مشابه اعداد شبه‌تصادفی با استفاده از توابع کتابخانه‌ای CUDA به‌صورت موازی تولید شده و در حافظهٔ دستگاه ذخیره می‌شود که خود سربار برای دستگاه ایجاد می‌کند و باعث کاهش کارایی می‌شود.

$$Rand_i = (T_i + GRnd + LRnd) \qquad (۷)$$

که در آن، T_i شناسهٔ نخ جاری، LRnd عدد تصادفی محلی در هر فراخوانی و GRnd عدد تصادفی سراسری در هر تکرار است. بنابراین $Rand_i$ عدد تصادفی برای نخ جاری است.

Unrolling Kernel
Initialize, set the "*block size*" and "*grid size*", with number of threads in a grid equaling to Swarm Size (*N*).
Input:*N,nx,ny,nb,gd_xx,gd_vx,gdweight_up,gd_pbestx.GRnd*
// *nx,nb* for Size of *Particle* and *block*
// *gdweight_up* for weight factor
// *gd_xx* to global memory
// *gd_vx* to global memory
// *localbestindex for Copmuted best index of Fitness*
// set thread ID
1. *ix = blockIdx.x * blockSize + threadIdx.x*
//Best Local *index* in Block
2. *BBlock*= (*ix* MOD *nx*) / (*nx* / *nb*)
//*BestIndex* in global
3. *BestIndex=globalbestindex[0]*
4. *BestGlobalIndex= localbestindex[BestIndex+((ix/nx)*nx)*]
//MyRNG is Function For Generate Random
Random number in device
//rnd is random number in global memory
5. R_1 *= MyRNG(ix+GRnd+LRnd)*
//MyRNG2 is Function For Generate Random number in device
6. R_2 *= MyRNG(ix+GRnd+LRnd)*
7. a_1*=gdweight_up*gd_vx[ix]*
8. a_3 *= gd_pbestx[ix]-gd_xx[ix]*
9. a_5 *= gd_pbestx*[*localbestindex*[*BBlock*]]-gd_xx[ix]
// Particle Velocity and Position Update
10. *gd_vx[ix] =* a_1*+(*C_1**(*R_1**a*$_3$*)) + (*C_2**(*R_2**a*$_5$*))*
11. *gd_vx[ix]= MIN(gd_vx[ix],* V_{MAX}*)*
12. *gd_vx[ix]= MAX(gd_vx[ix],-*V_{MAX}*)*
13. *gd_xx[ix]=gd_xx[ix]+gd_vx[ix]*
//Unroll kernel
14. R_1*=MyRNG(ix+rnd)*
15. R_2*=MyRNG2(ix+rnd2)*
16. a_1*=gdweight_up*gd_vx[ix]*
17. a_3*=(gd_pbestx[ix] - gd_xx[(ix)])*
18. a_5*=(gd_pbestx[BestGlobalIndex]- gd_xx[ix])*
19. *gd_vx[ix]=*a_1*+(*C_1**(*R_1**a*$_3$*))+ (*C_2**(*R_2**a*$_5$*))*
20. *gd_vx[ix]=MIN(gd_vx[ix],* V_{MAX}*)*
21. *gd_vx[ix]=MAX(gd_vx[ix],-*V_{MAX}*)*
22. *gd_xx[ix]=gd_xx[ix]+gd_vx[ix]*
Output: *gd_xx, gd_vx*

الگوریتم (۶): شبه کد بهینه‌سازی بروزرسانی موقعیت و سرعت با استفاده از تکنیک Unrolling kernel

Random Number Generator Function on GPU
Initialize, set the "*block size*" and "*grid size*", with number of threads in a grid equaling to Swarm Size (*N*).
//tid is thread ID
Input: ***tid,next***
1. Set *result* to 0 by default
2. Set *Prime Number* to 1103515245 by default
   //mul24 is function for multiply long number
3. *next* = mul24(*next*, 32)
4. *next* = mul24(*next*, *Prime Number*)
5. *next* = *next*+ 12345
   //convert long number to float number
6. *result* = (*next* / 65536) Mod 32768

Output: *RandomNumber in (0,1)*

الگوریتم (۷): شبه کد تولید عدد تصادفی بر اساس شناسهٔ نخ

۴. ارزیابی

در این مقاله، برنامهٔ اصلی PSO بر روی CPU پیاده‌سازی شده به‌اختصار *C* نام‌گذاری شده است [۵۷]. همین برنامه بر روی بستر سخت‌افزاری که در جدول (۳) توضیح داده شده، با استفاده از روش پیشنهادی بر روی پردازنده‌گرافیکی NVIDIA GeForce GT 525M به مشخصات جدول‌های (۱) و (۳) پیاده‌سازی و اجرا شده و به‌اختصار *GO* نام‌گذاری شده است. برنامهٔ PSO موازی‌سازی‌شده با استفاده از روش پیشنهادی در مقاله [۳۹]، بر روی پردازنده‌گرافیکی NVIDIA GeForce GT 525M به مشخصات جدول‌های (۱) و (۳) پیاده‌سازی و اجرا شده و *GS* نام‌گذاری گردیده است. در این پژوهش، تسریع، معیار اندازه‌گیری است؛ یعنی مدت‌زمانی که *GO* سریع‌تر از *C* اجرا می‌شود. برای بررسی مقیاس‌پذیری و بازدهی مدل ارائه‌شده، از پردازنده‌گرافیکی NVIDIA GeForce 1050Ti به مشخصات جدول‌های (۱) و (۳) استفاده شده است.

۴. ۱. توابع آزمون

به‌منظور نتیجه‌گیری دقیق‌تر و ارزیابی عملکرد برنامه‌های *C*، *GS* و *GO*، از سه تابع آزمون با پیچیدگی محاسباتی متفاوت استفاده شده است. صفر بودن مقدار بهینه، ویژگی مشترک این توابع است [۵۶]. متغیرهای توابع f_1 و f_2 مستقل هستند و متغیرهای تابع f_3 وابسته هستند. به این صورت که i+1 وابسته به i است. دامنه‌های تابع f_1 و تابع f_2 یکسان هستند اما تابع f_2 نسبت به تابع f_1 پیچیدگی محاسباتی بیشتری دارد؛ زیرا تابع f_1 فقط از جملات مربع تشکیل شده است درحالی‌که تابع f_2 علاوه بر جملات مربع و توان دوم، جملات کسینوسی نیز دارد. در جدول (۲) توابع مورد استفاده با عنوان، نام، ضابطه‌ها و دامنه‌های مربوط به آن‌ها آورده شده است:

جدول (۲): توابع آزمون

عنوان	نام تابع	ضابطه و دامنهٔ تابع
f_1	De Jong	$\sum_{i=1}^{n} x_i^2$
	دامنه	$(-5.12,5.12)^D$
f_2	Rastrigin	$10+\sum_{i=1}^{n}[x_i^2-10\times\cos(2\pi x_i)]$
	دامنه	$(-5.12,5.12)^D$
f_3	Rosenbrock	$\sum_{i=1}^{n}[100\times(x_i-x_{i-1}^2)^2+(x_i-1)^2]$
	دامنه	$(-1,1)^D$

۴. ۲. بستر آزمون

برای انجام آزمایش، مشخصات سیستم میزبان به شرح جدول (۳) است و برنامه به زبان C++ نوشته شده است.

جدول (۳): مشخصات سیستم میزبان بستر آزمایش

CPU	Model	Intel® Core™ i3-2330M CPU
	Frequency	2.20 GHz
	Bandwidth	21.3 GB/S
	Average MIPS	35000 MIPS
	FP64 (double)	24.085 GFLOPS
GPU	GPU_1	GeForce GT 525M CUDA™
	GPU_2	GeForce GT 1050Ti CUDA™
RAM	4.00 GB	
OS	Windows 10 pro 64Bit	
Tools	SDK7.5&MicroSoft Visual C++ 2013	

همان طور که در جدول (۱) مشاهده می‌شود پردازنده گرافیکی NVIDIA GeForce GT 525M مورد استفاده در این پژوهش نسبت به سایر پردازنده‌های گرافیکی، پردازندهٔ ضعیفی است [۲۱ و ۵۸]. با بیشینه‌سازی استفاده از منابع و رسیدن به تسریع قابل قبول همانند این مقاله، می‌توان انتظار رسیدن به کدهای مقیاس‌پذیر را داشت و با تسریع و عملکرد بهتر در سایر پردازنده‌های گرافیکی با قابلیت محاسباتی بالاتر، انتظار فراگیری الگوریتم پیاده‌سازی‌شده برای حل مسائل

سنگین در توسعه‌های آتی محققان میسر است. چه بسا مقایسه‌ای بین تسریع دو GPU موجود در جدول (۴) درستی این مطلب و مقیاس‌پذیری را نمایان می‌کند.

۴. ۳. معیار اندازه‌گیری

تسریع و بازدهی جزء متداول‌ترین معیارهای اندازه‌گیری کارایی هستند. تسریع را می‌توان به‌صورت نسبت زمان صرف‌شده برای اجرای برنامهٔ ترتیبی به زمان صرف‌شده برای اجرای برنامهٔ موازی محاسبه کرد [۵۴]. فرمول (۸)، فرمول کلی محاسبهٔ تسریع را بیان می‌کند.

$$Speedup = \frac{T_{CPU}}{T_{GPU}} \quad (۸)$$

که در آن، T_{CPU} میانگین زمان اجرای برنامه در CPU و T_{GPU} میانگین زمان اجرای برنامه در GPU است. بر مبنای همین فرمول، فرمول (۹) تسریع محاسبه‌شده برای برنامهٔ GS نسبت به برنامهٔ C است:

$$S_{GS} = \frac{T_C}{T_{GS}} \quad (۹)$$

که در آن، T_c میانگین زمان اجرای برنامهٔ C بر روی CPU است و T_{GS} میانگین زمان اجرای برنامهٔ GS بر روی GPU است و S_{GS} تسریع محاسبه‌شده برای برنامهٔ GS نسبت به برنامهٔ C است. بر مبنای همین فرمول، فرمول (۱۰) تسریع محاسبه‌شده برای برنامهٔ GO نسبت به C است:

$$S_{GO} = \frac{T_C}{T_{GO}} \quad (۱۰)$$

که در آن، T_c میانگین زمان اجرای برنامهٔ C بر روی CPU است و T_{GO} میانگین زمان اجرای برنامهٔ GO بر روی GPU است و S_{GO} تسریع محاسبه‌شده برای برنامهٔ GO نسبت به برنامهٔ C است.

برای ارزیابی عملکرد موازی و محاسبهٔ بازدهی از روش ارائه‌شده در مقالهٔ [۵۲] که در فرمول (۱۱) و فرمول (۱۲) آورده شده، استفاده شده است:

$$R = \frac{P_{GPU}}{P_{CPU}} \quad (۱۱)$$

که در آن، P_{GPU} اوج کارایی محاسباتی پردازنده‌گرافیکی استفاده‌شده برحسب $FOPS$ و P_{CPU} اوج کارایی محاسباتی تئوری CPU استفاده شده است. پس R نسبت اوج کارایی پردازنده‌گرافیکی نسبت به اوج کارایی CPU خواهد بود.

$$RE = \frac{S}{R} \quad (۱۲)$$

که در آن، S تسریع محاسبه‌شده و R نسبت اوج کارایی پردازنده‌گرافیکی نسبت به اوج کارایی CPU است. پس RE بازده محاسباتی اصلاح شده است [۵۲].

۴. ۴. آزمون

نویسندگان برای ارزیابی و تست عملکرد الگوریتم، سه برنامهٔ C، GO و GS را برای حل توابع f_1، f_2 و f_3 به تعداد ۲۰ بار مستقل اجرا کردند و دفعات تکرار را به‌صورت ثابت برابر ۱۰۰۰ در نظر گرفتند و دو آزمایش را با تعاریف زیر انجام دادند:

- آزمایش اول: بُعد به‌صورت ثابت (D=10) و جمعیت ذرات به‌صورت متغیر از ۲۰۰۰ تا ۱۰۰۰۰ با تغییر ۱۰۰۰ ذره در هر گام.
- آزمایش دوم: جمعیت به‌صورت ثابت (N=2000) ذره و ابعاد به‌صورت متغیر از ۱۰ تا ۱۰۰ بُعد با تغییر ۱۰ بُعد در هر گام.

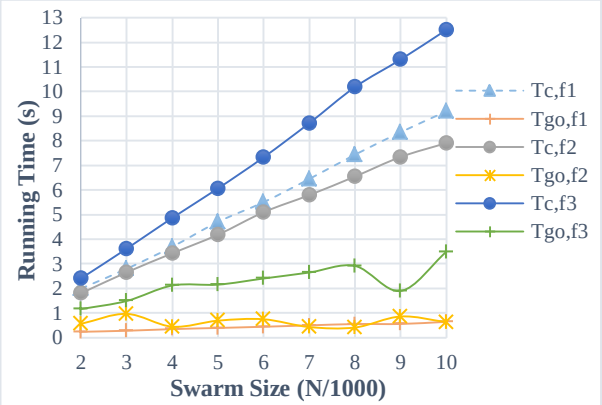
۴. ۴. ۱. یافته‌های آزمایش اول

در انجام این آزمایش، دفعات تکرار برنامه به‌صورت ثابت برابر ۱۰۰۰ در نظر گرفته شده است. بُعد مسئله (D) برای تمام توابع به‌صورت ثابت، ۱۰ در نظر گرفته شده است. N، متغیر جمعیت ذرات است. T_c، T_{GS} و T_{GO} زمان‌های اجرای برنامهٔ C، GS و GO هستند. RE_{GS} و RE_{GO}، بازده محاسبه‌شده برای دو برنامهٔ GS و GO هستند و S_{GS} و S_{GO}، تسریع محاسبه‌شده برای دو برنامهٔ GS و GO می‌باشند. یافته‌های اجرای آزمایش اول بر روی پردازنده‌های گرافیکی GPU_1 و GPU_2 در جدول (۴) آورده شده است.

۴. ۴. ۲. تحلیل یافته‌های آزمایش اول

در شکل (۵)، میانگین زمان اجرای توابع آزمون نسبت به میانگین زمان اجرای روش پیشنهادی آورده شده است.

با توجه به شکل (۵) مشاهده می‌شود که با رشد جمعیت، زمان اجرای تابع f_2 در برنامهٔ C بر روی CPU، به‌صورت یکنواخت و صعودی رشد می‌کند ولی اجرای همین تابع در برنامهٔ GO بر روی پردازنده‌گرافیکی به‌طور چشمگیری کاهش پیدا کرده است و این تابع با وجود رشد جمعیت ذرات با تأخیر کمتر و نزدیک به زمان اجرای تابع f_1 حل می‌شود. با وجود اینکه دامنه‌های توابع f_1 و f_2 یکسان است و تابع f_2 نسبت به تابع f_1، پیچیدگی بیشتری دارد ولی مشاهده می‌شود که برنامهٔ GO تأخیر کمتری نسبت به برنامهٔ C دارد به‌گونه‌ای که زمان اجرای این دو تابع بر روی هم منطبق است.



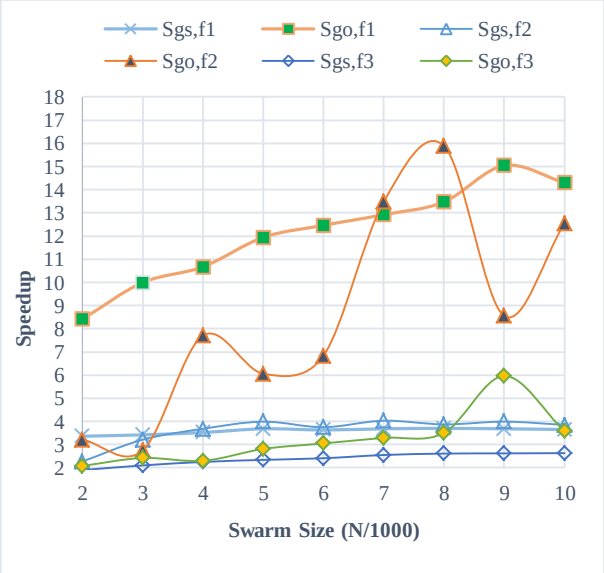
شکل (۵): مقایسهٔ میانگین زمان اجرای سه تابع بر روی GPU_1

جدول (۴): یافته‌های اجرای آزمایش اول بر روی GPU_1 و GPU_2

F(X)	N	T_C	GPU_1						GPU_2					
			T_{GS}	T_{GO}	S_{GS}	S_{GO}	RE_{GS}	RE_{GO}	T_{GS}	T_{GO}	S_{GS}	S_{GO}	RE_{GS}	RE_{GO}
f_1	2000	1.99	0.59	0.24	3.37	8.29	4.23	10.4	0.36	0.19	5.53	10.47	1.98	3.78
	3000	2.8	0.82	0.28	3.41	10	4.28	12.54	0.41	0.19	6.83	14.74	2.45	5.3
	4000	3.69	1.05	0.35	3.51	10.54	4.4	13.22	0.5	0.21	7.38	17.57	2.67	6.34
	5000	4.7	1.28	0.39	3.67	12.05	4.6	15.12	0.55	0.22	8.55	21.36	3.1	7.71
	6000	5.51	1.52	0.44	3.63	12.52	4.55	15.71	0.64	0.22	8.61	25.05	3.1	9.01
	7000	6.45	1.75	0.5	3.69	12.9	4.63	16.18	0.74	0.23	8.72	28.04	3.14	10.09
	8000	7.44	2.01	0.55	3.7	13.53	4.64	16.97	0.81	0.26	9.19	28.62	3.32	10.31
	9000	8.35	2.27	0.56	3.68	14.91	4.62	18.7	0.89	0.24	9.38	34.79	3.39	12.54
	10000	9.2	2.52	0.64	3.65	14.38	4.58	18.04	0.94	0.26	9.79	35.38	3.53	12.76
f_2	2000	1.8	0.8	0.56	2.25	3.21	2.82	4.03	0.41	0.34	4.39	5.29	1.59	1.91
	3000	2.65	0.83	0.97	3.19	2.73	4	3.42	0.68	0.53	3.9	5	1.41	1.8
	4000	3.43	0.93	0.45	3.69	7.62	4.63	9.56	0.26	0.23	13.19	14.91	4.76	5.37
	5000	4.18	1.05	0.69	3.98	6.06	4.99	7.6	0.35	0.18	11.94	23.22	4.33	8.36
	6000	5.1	1.36	0.75	3.75	6.8	4.7	8.53	0.37	0.23	13.78	22.17	4.97	8
	7000	5.79	1.44	0.43	4.02	13.47	5.04	16.9	0.25	0.17	23.16	34.06	8.36	12.29
	8000	6.55	1.69	0.41	3.88	15.98	4.87	20.05	0.21	0.18	31.19	36.39	11.25	13.12
	9000	7.34	1.84	0.86	3.99	8.53	5.01	10.7	0.63	0.61	11.65	12.03	4.22	4.33
	10000	7.91	2.05	0.63	3.86	12.56	4.84	15.76	0.21	0.18	37.67	43.94	13.55	15.82
f_3	2000	2.41	1.27	1.16	1.9	2.08	2.38	2.61	0.93	0.77	2.59	3.13	0.94	1.12
	3000	3.61	1.72	1.49	2.1	2.42	2.63	3.04	1.05	0.79	3.44	4.57	1.23	1.66
	4000	4.87	2.17	2.12	2.24	2.3	2.81	2.89	1.17	0.89	4.16	5.47	1.51	1.98
	5000	6.06	2.59	2.15	2.34	2.82	2.94	3.54	1.25	0.93	4.85	6.52	1.77	2.34
	6000	7.33	3.04	2.4	2.41	3.05	3.02	3.83	1.32	1	5.55	7.33	2.02	2.63
	7000	8.72	3.41	2.65	2.56	3.29	3.21	4.13	1.48	1.06	5.89	8.23	2.13	2.96
	8000	10.18	3.9	2.91	2.61	3.5	3.27	4.39	1.6	1.15	6.36	8.85	2.31	3.21
	9000	11.3	4.31	1.9	2.62	5.95	3.29	7.46	1.72	0.61	6.57	18.52	2.38	6.67
	10000	12.51	4.75	3.49	2.63	3.58	3.3	4.49	1.83	1.27	6.84	9.85	2.45	3.53

تأخیر تابع‌های f_2 و f_3 در برنامهٔ C بر روی CPU قابل توجه است و مدت زمان زیادی صرف حل مسئله می‌شود؛ که بیانگر ارائه تأخیر کمتر و مدت زمان اجرای قابل قبول‌تر است. همچنین زمان اجرای تابع f_3 در برنامهٔ GO بر روی پردازنده‌گرافیکی به‌طور چشمگیری کاهش پیدا کرده و این تابع با وجود رشد جمعیت ذرات با تأخیر کمتر و نزدیک به زمان اجرای تابع f_2 حل شده است. بنابراین می‌توان نتیجه گرفت که برنامهٔ GO پیاده‌سازی‌شده بر روی پردازنده‌گرافیکی برای توابع شدیداً محاسباتی و وابسته نیز مناسب است.

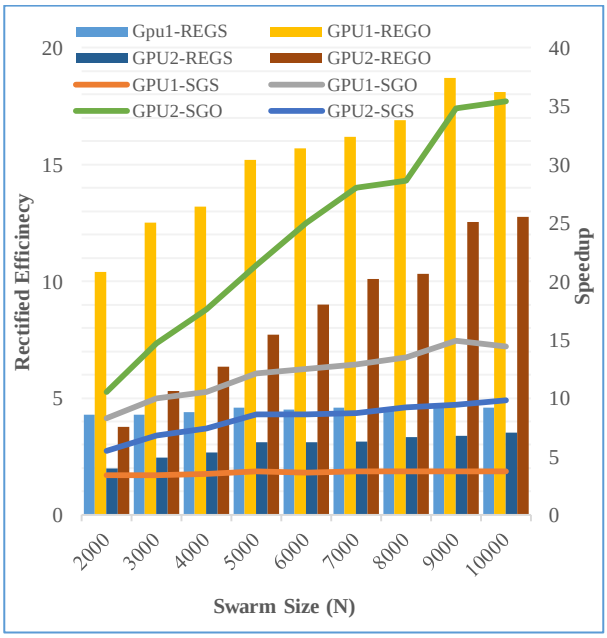
در شکل (۶) تسریع محاسبه‌شده برای سه تابع f_1، f_2 و f_3 در دو برنامهٔ GS و GO که بر روی GPU_1 انجام گرفته شده، با هم مقایسه شده است.



شکل (۶): مقایسهٔ تسریع تابع‌ها در GS و GO بر روی GPU_1

دیده می‌شود که برنامهٔ GO پیاده‌سازی‌شده عملکرد به‌مراتب بهتری از برنامهٔ GS دارد و این عملکرد برای تابع f_3 که متغیرهای آن وابسته هستند نیز صدق می‌کند. با وجود پیچیدگی بالای تابع f_2، تابع f_2 بیشترین مقدار تسریع را داشته و بعد از آن تابع f_1 با تسریع یکنواخت‌تری نسبت به تابع f_2 رشد می‌کند. علت موجی شکل بودن تسریع تابع f_2 تأثیر روش پیشنهادی و نحوهٔ بلوک‌بندی مناسب برای جمعیت ذرات بزرگ است؛ یعنی با بزرگ شدن مقیاس مسئله، برنامهٔ GO عملکرد به‌مراتب بهتری از GS و C دارد. با توجه به شکل (۶) تسریع تابع f_2 در برنامهٔ GO پیشنهادی به‌صورت چشمگیری رشد داشته و به مقدار تقریبی ۱۵/۹۸ برابری برنامهٔ C، و ۴ برابری برنامهٔ GS رسیده است. دلیل کاهش تسریع در برنامهٔ GS مدت‌زمانی است که صرف تولید عدد تصادفی می‌شود. در نهایت نتیجه می‌گیریم که روش پیشنهادی که در برنامهٔ GO اعمال شده است، عملکرد به مراتب بهتری از برنامه‌های GS و C دارد و برای حل مسائل سنگین با مقیاس بزرگ مناسب است. همچنین برنامهٔ GO می‌تواند مدت‌زمان قابل قبول‌تری را برای تابع‌های بهینه‌ساز با پیچیدگی‌های متفاوت ارائه کند و با استفاده از آن می‌توان با سرعت بیشتری این توابع را بر روی پردازنده‌گرافیکی نسبت به C و GS اجرا کرد.

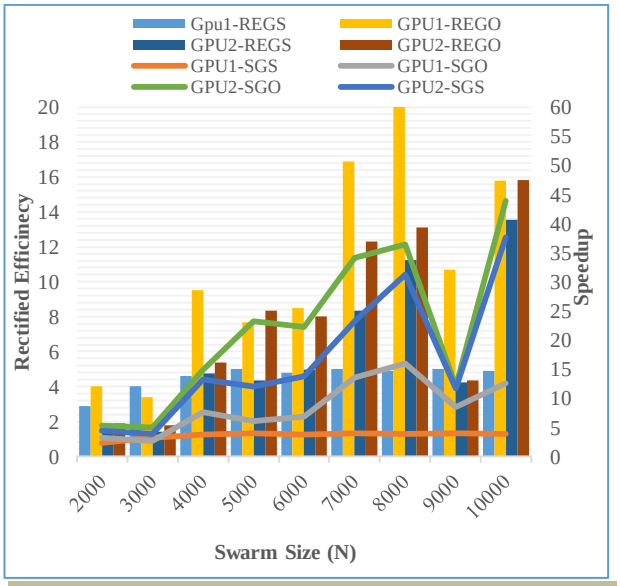
بررسی بازدهی و مقیاس‌پذیری روش پیشنهادی بر روی برنامهٔ GO برای تابع اول در شکل (۷) آورده شده است.



شکل (۷): مقایسهٔ تسریع و بازدهی تابع اول بر روی GPU_1 و GPU_2

دیده می‌شود که بازده محاسبه شده(RE) برای تابع اول با رشد جمعیت افزایش می‌یابد و برنامهٔ GO نسبت به برنامهٔ GS مقیاس‌پذیری به‌مراتب بهتر و بازده بالاتری دارد.

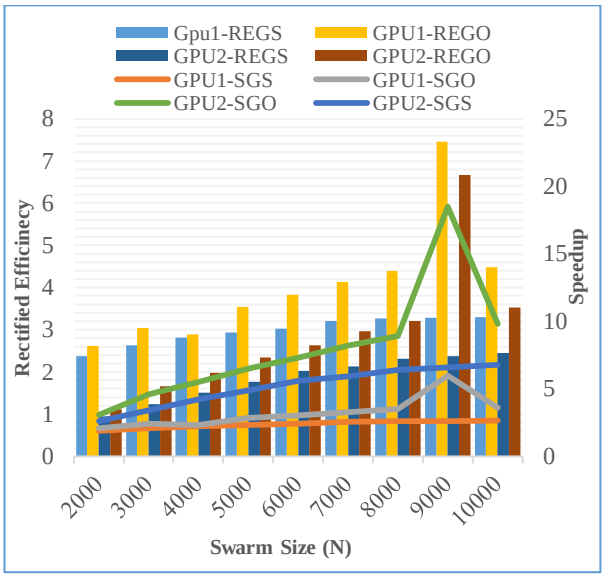
بررسی مقیاس‌پذیری اعمال روش پیشنهادی بر روی برنامه GO برای تابع دوم در شکل (۸) آورده شده است.



شکل (۸): مقایسهٔ تسریع و بازدهی تابع دوم بر روی GPU_1 و GPU_2

دیده می‌شود که بازده محاسبه‌شده (RE) برای تابع دوم با رشد جمعیت افزایش می‌یابد و برنامهٔ GO نسبت به برنامهٔ GS مقیاس‌پذیری و بازدهی به‌مراتب بهتری دارد.

بررسی مقیاس‌پذیری اعمال روش پیشنهادی بر روی برنامهٔ GO برای تابع سوم در شکل (۹) آورده شده است.



شکل (۹): مقایسهٔ تسریع و بازدهی تابع سوم بر روی GPU_1 و GPU_2

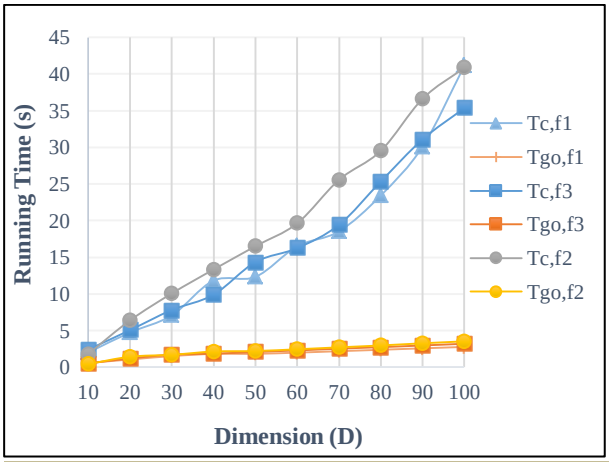
دیده می‌شود که بازده محاسبه‌شده (RE) برای تابع سوم که تابعی وابسته است، با افزایش جمعیت رشد می‌کند و برنامهٔ GO نسبت به برنامهٔ GS مقیاس‌پذیری به‌مراتب بهتری داشته و بازده بالاتری دارد.

۳.۴.۴. یافته‌های آزمایش دوم

در انجام این آزمایش، دفعات تکرار برنامه به‌صورت ثابت برابر ۱۰۰۰ در نظر گرفته شده است. جمعیت ذرات (N) برای تمام توابع به‌صورت ثابت ۲۰۰۰ در نظر گرفته شده است. D متغیر بُعد مسئله است. T_c، T_{GS} و T_{GO} زمان‌های اجرای برنامهٔ C، GS و GO هستند. RE_{GS} و RE_{GO}، بازده محاسبه‌شده برای دو برنامهٔ GS و GO هستند و S_{GS} و S_{GO}، تسریع محاسبه‌شده برای دو برنامهٔ GS و GO می‌باشند. یافته‌های اجرای آزمایش دوم بر روی پردازنده‌های گرافیکی GPU_1 و GPU_2 در جدول (۵) آورده شده است.

۴.۴.۴. تحلیل یافته‌های آزمایش دوم

در شکل (۱۰)، زمان اجرای توابع آزمون نسبت به زمان اجرای روش پیشنهادی آورده شده است.



شکل (۱۰): مقایسهٔ میانگین زمان اجرای سه تابع بر روی GPU_1

با توجه به شکل (۱۰) مشاهده می‌شود که با بزرگ شدن ابعاد مسئلهٔ زمان صرف‌شده برای اجرای تابع f_2 در برنامهٔ C بر روی CPU به‌صورت صعودی رشد می‌کند و مدت‌زمان زیادی صرف حل مسئله می‌شود ولی اجرای تابع f_2 در برنامهٔ GO بر روی پردازنده‌گرافیکی به طور چشمگیری کاهش پیدا کرده است و با توجه به یکسان بودن دامنه‌های تابع‌های f_1 و f_2 و پیچیدگی بیشتر تابع f_2 نسبت به تابع f_1 مشاهده شد که برنامهٔ GO تأخیر کمتری نسبت به برنامهٔ C دارد و با وجود رشد ابعاد مسئله و پیچیدگی محاسباتی بالا، با تأخیر کمتر و

نزدیک به زمان اجرای تابع f_1 حل می‌شود؛ به‌طوری که به نظر می‌رسد که مدت‌زمان اجرای این توابع بـر روی هـم افتـاده و منطبق هستند. و این بیانگر ارائهٔ تـأخیر و مـدت‌زمـان اجـرای قابل قبول برنامهٔ *GO* برای تابع‌های سنگین محاسباتی با وجود شدن بزرگ شدن ابعاد مسئله است. تأخیر تابع‌هـای f_1 ، f_2 و f_3 در برنامهٔ *C* بر روی CPU نزدیک به هم است؛ یعنـی CPU پیچیدگی محاسباتی این نوع مسائل را یکسان در نظر می‌گیـرد و مدت تأخیر حل مسئله به‌صورت صعودی رشد می‌کند ولـی با اجرای تابع f_3 در برنامـهٔ *GO* بـر روی پردازنـده‌گرافیکـی تأخیر به‌طور چشمگیری کاهش پیدا کـرده و بـا شـیب خیلـی ملایمی رشد می‌کند؛ در نتیجه، این تابع بـا وجـود رشـد بُعـد مسئله با تأخیر کمتر و نزدیک به زمان اجرای سایر تابع‌ها حل شده است. بنابراین می‌توان نتیجه گرفت که برنامهٔ *GO* نسـبت به برنامهٔ *C* برای حل تابع وابسته نیز مناسب‌تر است و عملکرد به مراتب بهتری از نظر تأخیر نسبت به برنامهٔ *C* دارد.

جدول (۵): یافته‌های اجرای آزمایش دوم بر روی GPU_1 و GPU_2

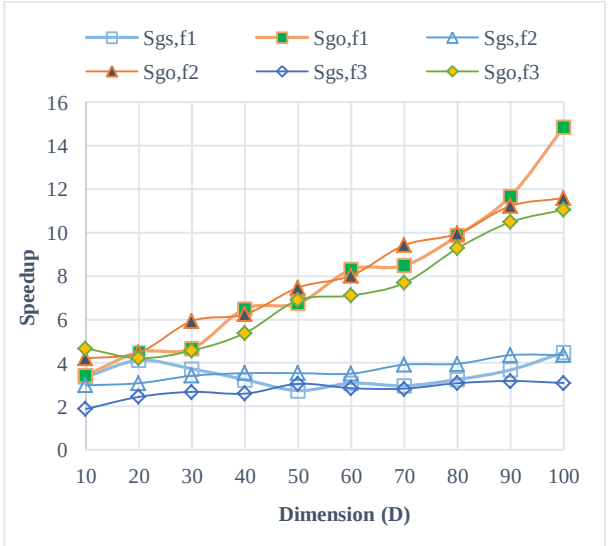
F(X)	D	T_C	GPU_1						GPU_2					
			T_{GS}	T_{GO}	S_{GS}	S_{GO}	RE_{GS}	RE_{GO}	T_{GS}	T_{GO}	S_{GS}	S_{GO}	RE_{GS}	RE_{GO}
f_1	10	1.99	0.59	0.58	3.36	3.43	4.21	4.30	0.36	0.19	5.59	10.72	7.01	13.45
	20	4.74	1.15	1.06	4.12	4.49	5.17	5.63	0.53	0.20	8.93	23.94	11.20	30.04
	30	7.08	1.90	1.52	3.73	4.66	4.68	5.85	0.78	0.34	9.05	20.64	11.36	25.89
	40	11.82	3.66	1.82	3.23	6.50	4.05	8.15	1.42	1.19	8.33	9.98	10.45	12.51
	50	12.33	4.52	1.83	2.73	6.74	3.42	8.45	1.63	1.32	7.55	9.32	9.48	11.70
	60	16.50	5.38	1.99	3.07	8.30	3.85	10.41	1.85	1.47	8.94	11.22	11.22	14.07
	70	18.54	6.29	2.19	2.95	8.48	3.70	10.64	2.20	1.63	8.43	11.34	10.57	14.23
	80	23.44	7.23	2.38	3.24	9.86	4.06	12.37	2.51	1.82	9.33	12.89	11.70	16.17
	90	30.11	8.18	2.58	3.68	11.68	4.62	14.65	3.76	1.98	8.01	15.23	10.05	19.11
	100	41.15	9.16	2.77	4.49	14.85	5.63	18.63	4.02	2.07	10.24	19.93	12.84	25.00
f_2	10	1.80	0.61	0.43	2.97	4.21	3.73	5.28	0.34	0.41	5.27	4.40	6.61	5.51
	20	6.39	2.08	1.43	3.07	4.47	3.85	5.61	1.13	1.03	5.66	6.20	7.10	7.78
	30	10.07	2.95	1.70	3.41	5.92	4.28	7.43	1.47	1.13	6.85	8.92	8.60	11.19
	40	13.34	3.78	2.14	3.53	6.23	4.43	7.82	1.65	1.26	8.09	10.62	10.15	13.32
	50	16.51	4.68	2.21	3.53	7.46	4.43	9.36	1.88	1.40	8.78	11.78	11.01	14.78
	60	19.64	5.59	2.45	3.51	8.01	4.40	10.05	2.11	1.55	9.29	12.67	11.66	15.89
	70	25.57	6.52	2.71	3.92	9.42	4.92	11.82	2.41	1.78	10.60	14.40	13.30	18.06
	80	29.53	7.46	2.97	3.96	9.95	4.97	12.48	2.69	1.93	10.99	15.34	13.79	19.24
	90	36.63	8.40	3.27	4.36	11.21	5.47	14.06	2.96	2.06	12.38	17.76	15.53	22.27
	100	40.87	9.38	3.53	4.36	11.57	5.47	14.51	3.28	2.24	12.48	18.22	15.66	22.86
f_3	10	2.41	1.28	0.52	1.88	4.67	2.36	5.86	0.93	0.77	2.59	3.15	3.25	3.95
	20	5.06	2.07	1.20	2.44	4.22	3.06	5.29	1.13	0.86	4.47	5.92	5.60	7.43
	30	7.81	2.93	1.71	2.67	4.57	3.35	5.73	1.41	1.01	5.56	7.70	6.97	9.66
	40	9.90	3.82	1.84	2.59	5.38	3.25	6.75	1.61	1.17	6.14	8.50	7.70	10.66
	50	14.34	4.72	2.08	3.04	6.88	3.81	8.63	1.88	1.26	7.63	11.35	9.57	14.23
	60	16.25	5.72	2.29	2.84	7.10	3.56	8.91	2.15	1.41	7.57	11.56	9.49	14.50
	70	19.43	6.90	2.53	2.82	7.68	3.54	9.63	2.44	1.62	7.97	11.99	10.00	15.04
	80	25.34	8.26	2.73	3.07	9.26	3.85	11.62	2.69	1.74	9.41	14.54	11.80	18.24
	90	31.09	9.80	2.97	3.17	10.47	3.98	13.13	2.98	1.92	10.45	16.22	13.11	20.35
	100	35.37	11.50	3.20	3.08	11.05	3.86	13.86	3.27	2.03	10.83	17.40	13.58	21.82

در شکل (۱۱) تسریع محاسبه‌شده برای تابع‌های f_1، f_2 و f_3 در دو برنامهٔ GS و GO با هم مقایسه شده است و دیده می‌شود که برنامهٔ GO پیاده‌سازی‌شده بر مبنای روش پیشنهادی عملکرد به‌مراتب بهتری نسبت به برنامه GS دارد و این عملکرد برای تابع f_3 نیز که متغیرهای آن وابسته‌اند صدق می‌کند. تابع f_1 بیشترین مقدار تسریع را داشته و بعد از آن توابع f_2 و f_3 با تسریع صعودی رشد می‌کنند؛ یعنی برنامهٔ GO با استفاده از روش پیشنهادی با بزرگ شدن ابعاد مسئله، عملکرد به‌مراتب بهتری نسبت به GS و C دارد. در شکل (۱۱) تسریع تابع f_1 در برنامهٔ GO پیشنهادی به‌صورت چشمگیری رشد داشته و به مقدار ۱۴/۸۵ برابری برنامهٔ C، و ۳ برابری برنامهٔ GS رسیده است.

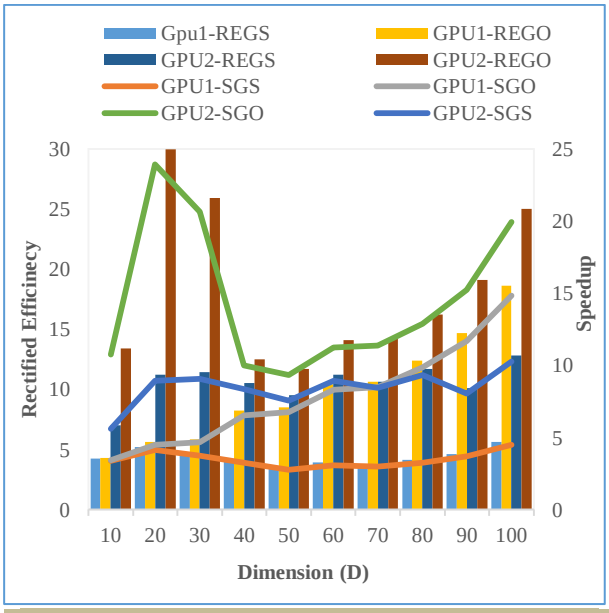
بررسی مقیاس‌پذیری اعمال روش پیشنهادی بر روی برنامهٔ GO برای تابع اول در شکل (۱۲) آورده شده است.

دیده می‌شود که بازده محاسبه‌شده (RE) برای تابع اول با بزرگ شدن ابعاد مسئله افزایش می‌یابد و همچنین برای ابعاد کوچک نیز این روش به‌خوبی عمل کرده و نتیجهٔ قابل توجهی داشته است و برنامهٔ GO نسبت به برنامهٔ GS مقیاس‌پذیری به‌مراتب بهتر و بازدهی بالاتری دارد.

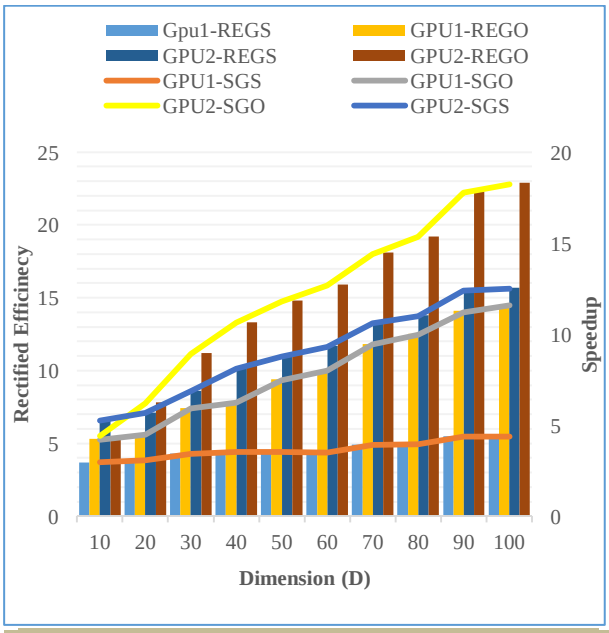
بررسی مقیاس‌پذیری اعمال روش پیشنهادی بر روی برنامهٔ GO برای تابع دوم در شکل (۱۳) آورده شده است.



شکل (۱۱): مقایسهٔ تسریع تابع‌ها در GS و GO



شکل (۱۲): مقایسه تسریع و بازدهی تابع اول بر روی GPU_1 و GPU_2

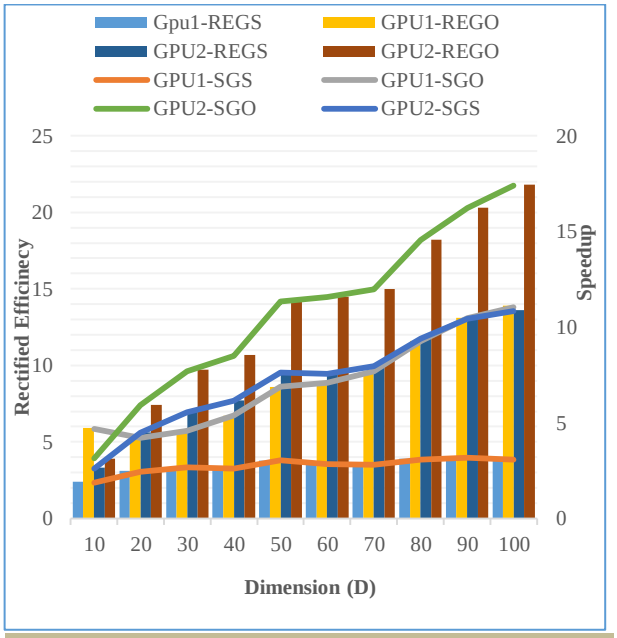


شکل (۱۳): مقایسهٔ تسریع و بازدهی تابع دوم بر روی GPU_1 و GPU_2

دیده می‌شود که بازده محاسبه‌شده (RE) برای تابع دوم با بزرگ شدن ابعاد مسئله، افزایش می‌یابد و برنامهٔ GO نسبت به برنامهٔ GS مقیاس‌پذیری به‌مراتب بهتر و بازدهی بالاتری دارد.

بررسی مقیاس‌پذیری اعمال روش پیشنهادی بر روی برنامهٔ GO برای تابع سوم در شکل (۱۴) آورده شده است. دیده می‌شود که بازده محاسبه‌شده (RE) برای تابع سوم با بزرگ

شدن ابعاد مسئله، افزایش می‌یابد و برنامهٔ *GO* نسبت به برنامهٔ *GS* مقیاس‌پذیری به‌مراتب بهتر و بازدهی بالاتری دارد؛ پس می‌توان نتیجه گرفت که دلیل کاهش تسریع در برنامهٔ *GS* و نزولی شدن نمودار آن مدت‌زمانی است که صرف محاسبات غیرموازی و تولید عدد تصادفی می‌شود. پس روش پیشنهادی که در برنامهٔ *GO* اعمال شده است، عملکرد مناسب‌تری نسبت به برنامه‌های *GS* و *C* دارد و برای حل مسائل سنگین با ابعاد بزرگ بسیار بهتر عمل می‌کند. همچنین برنامهٔ *GO* می‌تواند مدت‌زمان قابل قبول‌تری را برای تابع‌های بهینه‌ساز با پیچیدگی‌های متفاوت ارائه کند و سرانجام با سرعت بیشتری می‌توان این توابع را بر روی پردازنده‌گرافیکی نسبت به *C* و *GS* اجرا کرد.



شکل (۱۴): مقایسهٔ تسریع و بازدهی تابع سوم بر روی GPU_1 و GPU_2

۵. نتیجه‌گیری

در این مقاله، پیاده‌سازی و اجرای الگوریتم بهینه‌سازی اجتماع ذرات بر روی پردازنده‌گرافیکی Geforce GT 525m انجام گرفت. این پردازنده در مقایسه با پردازندهٔ Intel Core i3، پردازندهٔ ضعیفی است؛ زیرا فقط ۹۸ هستهٔ پردازشی دارد که در دو واحد چندپردازشی گنجانده شده است. نویسندگان با اجرای آزمایش‌هایی که بر روی الگوریتم پیاده‌سازی‌شده انجام دادند، نتیجه گرفتند که اجرای الگوریتم موازی بهبودیافته با استفاده از روش و استراتژی‌های پیشنهادی بر روی پردازنده‌گرافیکی می‌تواند به‌سرعت قابل توجه، و تأخیر به‌مراتب کمتری نسبت به CPU برسد و با استفاده از آن، مسائل را با سرعت بسیار بیشتر و تأخیر کمتری نسبت به برنامهٔ پیاده‌سازی‌شده بر روی CPU حل کرد. همچنین عملکرد برنامهٔ پیشنهادی با استفاده از تابع‌هایی با پیچیدگی محاسباتی متفاوت محک زده شد و نتایج نشان داد که با وجود اینکه نخ‌های پردازنده‌گرافیکی بسیار سبک‌وزن‌تر از نخ‌های CPU هستند، با استفاده از روش پیشنهادی و افزایش سطح موازی‌سازی دستورات، می‌توان تأخیر ناشی از این نوع محاسبات را پنهان کرد و برای حل این‌گونه مسائل پیچیده، انتظار نتایج و عملکرد به‌مراتب بهتر پردازنده‌گرافیکی را نسبت به CPU را داشت. همچنین بررسی مقیاس‌پذیری و بازدهی روش پیشنهادی بر روی پردازنده‌گرافیکی Geforce 1050Ti نشان داد که روش پیشنهادی مقیاس‌پذیری بهتر و بازده محاسباتی بالاتری دارد و این به‌دلیل افزایش سطح موازی‌سازی دستورات و کاهش سربار ناشی از تولید اعداد تصادفی می‌باشد.

مراجع

[1] Engelbrecht A. P., *Fundamentals of Computational Swarm Intelligence*, John Wiley & Sons, 2005.

[2] Eberhart R. C., Shi Y. and Kennedy J., *Swarm Intelligence (Morgan Kaufmann series in evolutionary computation)*, Morgan Kaufmann publisher, 2001.

[3] Yang X. S., "*Swarm intelligence based algorithms:a critical analysis.Evolutionary intelligence*", pp. 17-28, 2014.

[4] Kennedy J. and Eberhart R., "*Particle swarm optimization*", in IEEE international conference on neural networks, Perth, Australia, 1995.

[5] Bratton D. and Kennedy J., "*Defining a standard for particle swarm optimization*", IEEE swarm intelligence symposium, pp. 120-127, 2007.

[6] Tan Y. and Zhu Y., "*Fireworks algorithm for optimization*", in Advances in Swarm Intelligence (LNCS 6145). Berlin, Germany: Springer, 2010, pp.

355–364.

[7] Ross P., "*Why CPU frequency stalled*", IEEE Spectr., vol. 45, no. 4, p. 72, Apr. 2008

[8] NVIDIA'S Next Generation CUDA Compute Architecture: Fermi, Nvidia, 2010.

[9] Zahran M., *Heterogeneous computing*, Communications of the ACM, 60(3), pp.42-45, 2017.

[10] Cheng J., *Professional Cuda C Programming*, Indianapolis, IN: John Wiley and Sons, Inc, 2014.

[11] Essaid M., Idoumghar L., Lepagnot J. and Brévilliers M., *GPU parallelization strategies for metaheuristics: a survey*, International Journal of Parallel, Emergent and Distributed Systems, 34(5), pp.497-522, 2018.

[12] Ujaldon M., *HPC Accelerators with 3D Memory*, 2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES), 2016.

[13] Mu D., Lee E. and Chen P., *Rapid earthquake detection through GPU-Based template matching*, Computers & Geosciences, 109, pp.305-314, 2017.

[14] Lim S. and Kang P., *Implementing Scientific Simulations on GPU-accelerated Edge Devices*, International Conference on Information Networking (ICOIN), 2020.

[15] Barajas C., Gobbert M. and Wang J., *Performance Benchmarking of Data Augmentation and Deep Learning for Tornado Prediction*, IEEE International Conference on Big Data (Big Data), 2019.

[16] Reddy B., *Performance Analysis of GPU V/S CPU for Image Processing Applications*, International Journal for Research in Applied Science and Engineering Technology, V(II), pp.437-443, 2017.

[17] Jurczuk K., Czajkowski M. and Kretowski M., *Multi-GPU approach for big data mining*, Proceedings of the Genetic and Evolutionary Computation Conference Companion, 2019.

[18] Mittal S., and Vetter J. S., "*A survey of CPU-GPU heterogeneous computing techniques*", ACM Computing Surveys (CSUR), vol. 47, no. 4, 2015.

[19] Domanski L., Bednarz T., Gureyev T. E., Murray L., Huang E., and Taylor J. A., "*Applications of heterogeneous computing in computational and simulation science*", Fourth IEEE International Conference on Utility and Cloud Computing, IEEE, pp. 382–389, 2011.

[20] Fei X., Li K., Yang W. and Li K., *Analysis of energy efficiency of a parallel AES algorithm for CPU-GPU heterogeneous platforms*, Parallel Computing, 94-95, p.102621, 2020.

[21] Top500.org. November 2019 | TOP500 Supercomputer Sites. [online] Available at: <https://www.top500.org/lists/2019/11/> [Accessed 27 May 2020].

[22] Soyata T., *GPU Parallel Program Development Using CUDA*, 2019.

[23] Nvidia T., *CUDA C BEST PRACTICES GUIDE*, NVIDIA Corporation, 2015.

[24] TechPowerUp, Techpowerup. [online] Available at: <https://www.techpowerup.com/gpu-specs/> [Accessed 27 May 2020].

[25] Ma S., Liu Z., Chen S., Huang L., Guo Y., Wang Z. and Zhang M., *Coordinated DMA: Improving the DRAM Access Efficiency for Matrix Multiplication*, IEEE Transactions on Parallel and Distributed Systems, 30(10), pp.2148-2164, 2019.

[26] Ghorpade J., Parande J., Kulkarni M. and Bawaskar A., "*GPGPU PROCESSING IN CUDA ARCHITECTURE*", Advanced Computing: An International Journal (ACIJ), vol. 3, pp. 105-120, January 2012.

[27] Cook S., *in CUDA programming: a developer's guide to parallel computing with GPUs*, Newnes, 2012.

[28] Shi Y., "*Particle swarm optimization: developments, applications and resources*", in Proceedings of the 2001 congress on evolutionary computation, 2001.

[29] Shi Y. and Eberhart R. C., "*Parameter selection in particle swarm optimization*", in International conference on evolutionary programming, Berlin, Heidelberg, 1998.

[30] Eberhart R. C. and Shi Y., "*Evolving artificial neural networks*", in Proceedings of the international conference on neural networks and brain, PRC, 1998.

[31] Shi Y. and Eberhart R., "*A modified particle swarm optimizer*," in IEEE world congress on computational intelligence, Anchorage, AK, USA, USA, 1998.

[32] Swarmintelligence.org. Particle Swarm Optimization: Tutorial. [online] Available at: <http://www.swarmintelligence.org/tutorials.php> [Accessed 27 May 2019].

[33] Zhou Y. and Tan Y., "*GPU-based parallel particle swarm optimization*", in Proc. IEEE Congr. Evol. Comput. (CEC), Trondheim, Norway, pp. 1493–1500, 2009.

[34] Zhu W. and Curry J., "*Parallel ant colony for nonlinear function optimization with graphics hardware acceleration*", in Proc. IEEE Int. Conf. Syst. Man Cybern. (SMC), San Antonio, TX, USA, pp. 1803–1808, 2009.

[35] Umbarkar A. J., Joshi M. S. and Rothe N. M., "*Genetic algorithm on general purpose graphical*," ICTACT Soft, pp. 492-497, 2013.

[36] Zhou Y. and Tan Y., "*GPU-based parallel multiobjective particle swarm optimization*", Artificial Intelligence, pp. 125-141, 2011.

[37] Mussi L., Stefano C. and Daolio F., "*GPU-based road sign detection using particle swarm optimization*", in 9th IEEE International Conference on Intelligent Systems Design and Applications, 2009.

[38] Zhou Y. and Tan Y., "*GPU-based parallel particle swarm optimization*", in IEEE Congress on Evolutionary Computation, 2009.

[39] Veronese L. P. and Krohling R. A., "*Swarms flight: Accelerating the particles using C-CUDA*," in IEEE

Congress on Evolutionary Computation, 2009.

[40] Chang Y. L. and Fang J. P., "*Band selection for hyperspectral images based on parallel particle swarm optimization schemes*", in IEEE International Geoscience and Remote Sensing Symposium, 2009.

[41] Zhu W. and Curry J., "*Particle swarm with graphics hardware acceleration and local pattern search on bound constrained problems*", in IEEE Swarm Intelligence Symposium, 2009.

[42] Bastos-Filho C. J., Oliveira M. A., Nascimento D. N. and Ramos A. D., "*Impact of the random number generator quality on particle swarm optimization algorithm running on graphic processor units*", in 10th IEEE International Conference on Hybrid Intelligent Systems (HIS), 2010.

[43] Jambhlekar P. A., Mishra M. and Subramaniam S. V., "*Parallel implementation of MOPSO on GPU using OpenCL and CUDA*", in 18th IEEE International Conference on High Performance Computing (HiPC), 2011.

[44] Miguel C. M., Miguel A., Rodrguez-Vazquez J. J. and Antonio G. I., "*Accelerating particle swarm algorithm with GPGPU*", in IEEE 19th International Euromicro Conference on Parallel, Distributed and Network Based Processing, 2011.

[45] Zhu H. and Guo Y., "*Paralleling Euclidean particle swarm optimization in CUDA*", in 4th International Conference on Intelligent Networks and Intelligent Systems, 2011.

[46] Wenna L. and Zhenyu Z., "*A CUDA-based multi channel particle swarm algorithm*", in 4th International Conference on Control, Automation and Systems Engineering, 2011.

[47] Platos J., Snasel V., Jezowicz T., Kromerand P. and Abraham A., "*A PSO-based document classification algorithm accelerated by the CUDA platform*", in IEEE International Conference on Systems,Man and Cybernetics, 2012.

[48] Zhang B., Zheng H., Wei M., Wu R. and Sheng X., "*Particle swarm optimization of frequency selective surface*", in IEEE International Conference on Cross Strait Quad- Regional Radio Science and Wireless Technology, 2012.

[49] Sharma B., Thulasiram R. K. and Thulasiraman P., "*Portfolio management using particle swarm optimization on GPU*", in 10th IEEE International Symposium on Parallel and Distributed Processing with Applications, 2012.

[50] Calazan R. D. M., Nedjah N. and Mourelle L. D. M., "*A Cooperative Parallel Particle Swarm Optimization for High-Dimension Problems on GPUs*," in IEEE Computational Intelligence and 11th Brazilian Congress on Computational Intelligence (BRICS-CCI &CBIC), 2013.

[51] Cataveral I. and Rubio F., *Dealing with Swarm Intelligence on GPUs*, IEEE International Conference on Systems, Man and Cybernetics (SMC), 2019.

[52] Tan Y. and Ding K., *A Survey on GPU-Based Implementation of Swarm Intelligence Algorithms*, IEEE Transactions on Cybernetics, 46(9), pp.2028-2041, 2016.

[53] Woolley C., *GPU Optimization Fundamentals*, NVIDIA, 2013.

[54] Tan Y., *GPU-based Parallel Implementation of Swarm Intelligence Algorithms*, Morgan Kaufmann, 2016.

[55] Docs.nvidia.com, PTX ISA :: CUDA Toolkit Documentation. [online] Available at: http://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[56] Molga M. and Smutnicki C., "*Test functions for optimization needs In Using anEvolutionary Heuristics for Solving the Outdoor Advertising Optimization Problem*", Journal of Computer Sciences and Applications, 2005.

[57] Shi Y., "*Swarm Intelligence*", [Online]. Available at: <http://www.swarmintelligence.org/codes.php>

[58] "Benchmarks / Tech", notebookcheck, 26 05 2012. [Online]. Available: <https://www.notebookcheck.net/Intel-Core-i3-2330M-Notebook-Processor.52200.0.html>.