1st National Conference on Applications of Intelligent Systems (Soft Computing) in Science and Technology
Islamic Azad University, Quchan Branch
4 - 5 , March , 2013
اولین همایش ملی
کاربرد سیستم های هوشمند (محاسبات نرم) در علوم وصنایع

الگوریتم بهینه‌سازی فاخته‌ی موازی مبتنی بر الگوی ارباب – بنده با استفاده از تکنیک ریز دانه بر روی واحدهای پردازش گرافیکی

سید مسعود عقیلی[۱]، الهامه زارعی[۲]، حسین دلداری[۳]، مجید وفایی جهان[٤]

۱- دانشگاه آزاد اسلامی واحد مشهد، s.masoud.aghili@gmail.com

۲- دانشگاه آزاد اسلامی واحد مشهد، elhame.z@gmail.com

۳- دانشگاه فردوسی مشهد، hdeldari@yahoo.com

٤- دانشگاه آزاد اسلامی واحد مشهد، vafaeijahan@mshdiau.ac.ir

چکیده

الگوریتم بهینه‌سازی فاخته مانند بسیاری دیگر از الگوریتم‌های فرا ابتکاریِ مبتنی بر جمعیت اولیه، ذاتاً موازی است و می‌تواند به طور کارآمدی بر روی واحدهای پردازش گرافیکی پیاده‌سازی شود. یکی از الگوهای موازی‌سازی الگوریتم‌های بهینه‌سازی الگوی ارباب – بنده می‌باشد. در این مقاله، موازی‌سازی الگوریتم فاخته روی واحدهای پردازش گرافیکی با استفاده از معماری دستگاهِ یکپارچه‌ی محاسباتی و مبتنی بر الگوی ارباب – بنده ارائه می‌شود. با توجه به ذات موازی الگوریتم بهینه‌سازی فاخته، موازی سازی با تکنیک ریز دانه انجام شده است. تابع برازندگی، خوشه‌بندی و همگرایی فاخته‌ها بر روی GPU پیاده سازی می‌شوند. برای افزایش کارایی الگوریتم موازی از حافظه‌ی اشتراکی، عملیات کاهش و عوامل دیگر تأثیر گذار استفاده شده است. آزمایش‌ها با مقایسه‌ی زمان اجرای الگوریتم سری و الگوریتم موازی و با تغییر تعداد ابعاد مسأله و همچنین جمعیت، با استفاده از شش تابع محک شناخته شده، انجام شده‌اند. نتایج آزمایش‌ها افزایش بهره‌ی سرعت الگوریتم موازی نسبت به سری را گزارش می دهند.

کلید واژه الگوریتم بهینه‌سازی فاخته (COA)، محاسبات موازی، واحدهای پردازش گرافیکی (GPU)، معماری دستگاهِ یکپارچه‌ی محاسباتی (CUDA™)، مدل ارباب – بنده.

۱- مقدمه

در حال حاضر محاسبات موازی با استفاده از واحدهای پردازش گرافیکی[1] (GPU) به عنوان روشی قدرتمند برای کاهش زمان اجرای برنامه‌های پیچیده با زمان اجرای طولانی، کاربرد زیادی دارند. اخیراً، از بین رویکردهای موازی-سازی الگوریتم‌های تکاملی، بسیاری از روش‌ها بر روی GPU پیاده‌سازی شده‌اند؛ بنابراین می‌توان در مطالعه‌ی مسائل موازی‌سازی برای الگوریتم بهینه سازی فاخته[2] (COA)، روی کارهای منتشر شده‌ی قبلی تکیه و رویکردهای طبقه-بندی شده را مرور کرد (۱۱).

اکثر روش‌های موازی سازی الگوریتم‌های تکاملی مبتنی بر الگوی ارباب – بنده[3] هستند. در (٤) به پیاده سازی تابع برازندگی بر روی GPU اکتفا شده است. در (۱۵) به غیر از تابع برازندگی، زیر توابع دیگر از الگوریتم مورد نظر بر روی GPU پیاده سازی شده‌اند. در رویکرد ارباب- بنده، گره‌ی ارباب الگوریتم را به صورت کامل اجرا می‌کند درحالی‌که گره‌های بنده ارزیابی تابع برازندگی را انجام می‌دهند. از این رو هر چه ارزیابی تابع برازندگی پر هزینه‌تر باشد پیاده سازی کارآمدتر خواهد بود و بخش بزرگ‌تری از کل زمان اجرا را به عهده می‌گیرد (۲). در (۳) یک الگوریتم تکاملی برای ارزیابی پنج تابع محک ساده ارائه شده است که برنامه نویسی تکاملی سریع خوانده می‌شود. در این رویکردِ ارباب – بنده، بعضی از اعمال مانند حلقه‌ی اصلی الگوریتم ژنتیک، عملیات تقاطع و انتخاب در CPU و توابع برازندگی و جهش در GPU اجرا می‌شوند. زمانی که اندازه‌ی جمعیت افزایش می‌یابد بیشترین بهره‌ی سرعت[4] به دست آمده ۵/۰۲ است. این معمولی‌ترین سازماندهی بر روی GPU است از این رو در هنگام اجرای موازیِ توابع، به هیچ تعاملی بین نخ‌ها نیاز نیست.

برای بهبود کارایی موازی سازی بر روی GPU علاوه بر استفاده‌ی درست از دستورالعمل‌های ریاضی، کنترل جریان و هم‌زمانی در برنامه می‌بایست شناخت دقیقی از سخت افزار کارت‌های گرافیکی نیز داشت. استفاده از حافظه‌ی مناسب هم بر سرعت دسترسی نخ‌ها به داده‌ها تأثیر می‌گذارد. سرعت اجرای دستورالعمل‌های ریاضی پیاده سازی شده در GPU ها نسبت به همتای استانداردشان که در CPU اجرا می‌شوند، بیشتر می‌باشد(۸).

COA یکی از الگوریتم‌های جدید در زمینه‌ی بهینه‌سازی می‌باشد. این الگوریتم همچون همه‌ی الگوریتم‌های تکاملی دیگر، مبتنی بر جمعیت اولیه است. خوشه‌بندی و همگرایی از قسمت‌های اصلی این الگوریتم می‌باشند. یک محدودیت بزرگ برای این الگوریتم و همچنین برای همه‌ی الگوریتم‌های مبتنی بر جمعیت، هزینه‌ی زیاد محاسباتی تابع برازندگی با مصرف زمان است (۷). مهم‌ترین هدف از موازی کردن این الگوریتم‌ها، کاهش زمان اجراست. در بخش ۲ به معرفی الگوریتم بهینه‌سازی فاخته پرداخته می‌شود. جزئیاتی در مورد GPU ها و برنامه نویسی آن در بخش ۳ شرح داده می‌شود. الگوریتم پیشنهادی در بخش ٤ مطرح می‌شود و آزمایش‌ها و نتیجه‌گیری در بخش‌های ۵ و ٦ بیان می‌شوند.

۲- الگوریتم بهینه‌سازی فاخته (٦)

همانند سایر الگوریتم‌های تکاملی، COA هم با یک جمعیت اولیه کار خود را شروع می‌کند. این جمعیت از فاخته‌ها در لانه‌ی تعدادی پرنده‌ی دیگر که آن‌ها را میزبان می‌نامیم تخم گذاری می‌کنند. تعدادی از این تخم‌ها که شباهت

[1] Graphic processing units
[2] Cuckoo Optimization Algorithm
[3] Master-Slave Model
[4] Speed Up

بیشتری به تخم‌های پرنده میزبان دارند شانس بیشتری برای رشد و تبدیل شدن به فاخته‌ی بالغ خواهند داشت. سایر تخم‌ها توسط پرنده‌ی میزبان شناسایی شده و از بین می‌روند. میزان تخم‌های رشد کرده، مناسب بودن لانه‌های آن منطقه را نشان می‌دهند. هرچه تخم‌های بیشتری در یک ناحیه قادر به زیست باشند و نجات یابند به همان اندازه سود (تمایل) بیشتری به آن منطقه اختصاص می‌یابد؛ بنابراین موقعیتی که در آن بیشترین تعداد تخم‌ها نجات یابند پارامتری خواهد بود که COA قصد بهینه‌سازی آن را دارد.

فاخته‌ها برای بیشینه کردن نجات تخم‌های خود دنبال بهترین منطقه می‌گردند. پس از آنکه جوجه‌ها از تخم بیرون آمدند و تبدیل به فاخته‌ی بالغ شدند، جوامع و گروه‌هایی تشکیل می‌دهند. هر گروه منطقه‌ی سکونتی برای زیست خود دارد. بهترین منطقه سکونت تمام گروه‌ها مقصد بعدی فاخته‌ها در سایر گروه‌ها خواهد بود. تمام گروه‌ها به سمت بهترین منطقه‌ی موجود فعلی مهاجرت می‌کنند. هر گروه در منطقه‌ای نزدیک بهترین موقعیت فعلی ساکن می‌شود. با در نظر گرفتن تعداد تخمی که هر فاخته خواهد گذاشت و همچنین فاصله فاخته‌ها از منطقه‌ی بهینه‌ی فعلی برای سکونت، تعدادی شعاع تخم‌گذاری محاسبه شده و شکل می‌گیرند. سپس فاخته‌ها شروع به تخم‌گذاری تصادفی در لانه‌هایی داخل شعاع تخم‌گذاری خود می‌کنند. این پروسه تا رسیدن به بهترین محل برای تخم‌گذاری (منطقه با بیشترین سود) ادامه می‌یابد. این محلِ بهینه جایی است که بیشترین تعداد فاخته‌ها در آن گرد می‌آیند.

وقتی جوجه فاخته‌ها رشد کردند و بالغ شدند مدتی در محیط‌ها و گروه‌های خودشان زندگی می‌کنند ولی وقتی زمان تخم‌گذاری نزدیک می‌شود به محل‌های سکونت بهتر، که در آنجا شانس زنده ماندن تخم‌ها بیشتر است مهاجرت می‌کنند. پس از تشکیل گروه‌های فاخته در مناطق مختلف زیستی (فضای جستجوی مسأله)، گروهِ دارای بهترین موقعیت به عنوان نقطه‌ی هدف برای سایر فاخته‌ها، جهت مهاجرت انتخاب می‌شود.

وقتی فاخته‌های بالغ در اقصی نقاط محیط زیست زندگی می‌کنند تشخیص اینکه هر فاخته به کدام گروه تعلق دارد کار سختی است. برای حل این مشکل، گروه‌بندی فاخته‌ها توسط روش خوشه‌بندی K-means انجام می‌شود.

پس از چند تکرار تمام جمعیت فاخته‌ها به یک نقطه‌ی بهینه با حداکثر شباهت تخم‌ها به تخم‌های پرندگان میزبان و همچنین به محل بیشترین منابع غذایی می‌رسند. این محل بیشترین سود کلی را خواهد داشت و در آن کم‌ترین تعداد تخم‌ها از بین خواهند رفت. همگرایی بیش از ۹۵٪ تمام فاخته‌ها به سمت یک نقطه، الگوریتم COA را به انتهای خود می‌رساند.

۳. معماری GPU و ابزار CUDA و سلسله مراتب حافظه

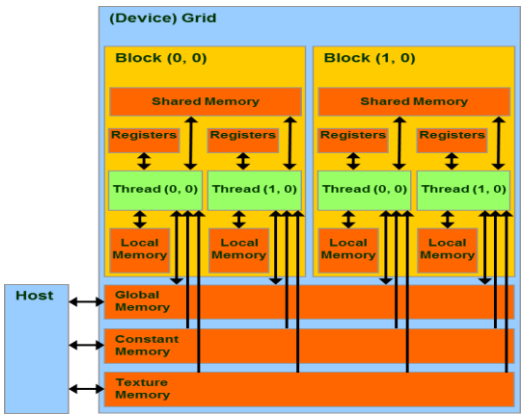
در نوامبر ۲۰۰۶ شرکت NVIDIA یک معماری محاسباتی موازی همه منظوره (CUDA)[1]، که یک ابزار سودمند برای توسعه‌ی برنامه‌های علمی در جهت محاسبات موازی می‌باشد، معرفی کرد. CUDA به توسعه دهندگان نرم افزار این اجازه را می‌دهد با دستگاه‌های GPU مرتبط شوند و کدهایی بنویسند که به طور مستقیم بر روی آن‌ها اجرا شوند. در واقع با فراهم کردن یک GPU سازگار و ابزار CUDA حتی در یک کامپیوتر نه چندان قوی می‌توان برنامه‌ای موازی را توسعه داد (۹).

لازمه‌ی الگوی برنامه‌نویسی CUDA تفکیک مسأله‌ی مطرح شده به تعداد زیادی زیر مسأله است که این زیر مسائل می‌توانند به طور موازی حل شوند. از طرفی هر کدام از این زیر مسائل هم می‌توانند به مسائل کوچک‌تری

[1] Compute Unified Device Architecture (CUDA)

تقسیم و به طور موازی اجرا شوند. در مبحث CUDA به CPU و حافظه کامپیوتر اصطلاحاً میزبان[1] و به GPU و حافظه‌های آن دستگاه[2] گفته می‌شود. عنصر نرم‌افزاری که دستورالعمل‌هایی برای اجرای هر نخ توصیف می‌کند کرنل[3] نامیده می‌شود. بر خلاف اجرای توابع معمولی که روی CPU تنها یک بار اجرا می‌شوند، کرنل به تعداد n بار به صورت موازی با n نخ متفاوتِ CUDA اجرا می‌شود. مقدار n توسط برنامه‌نویس مشخص می‌شود. هنگام فراخوانی یک کرنل در میزبان، می‌بایست تعداد بلاک‌های نخ و تعداد نخ‌های هر بلاک مشخص شوند (۱۰).

GPU ها دارای چندین نوع حافظه با دسترسی‌های متفاوت می‌باشند. هر نخ به حافظه‌ی محلی مخصوص خود دسترسی خواندن – نوشتن دارد. حافظه‌ی اشتراکی برای همه‌ی نخ‌های داخل یک بلاک نخ قابل خواندن – نوشتن است و سریع‌ترین حافظه از لحاظ دسترسی نخ‌های همان بلاک است. حافظه‌ی سراسری برای همه‌ی نخ‌های پردازشی تعریف شده قابل دسترسی به صورت خواندن – نوشتن است. حافظه‌ی سراسری دارای فضای ذخیره‌سازی زیاد (در حد چند گیگابایت) است، اما زمان دسترسی پایین می‌باشد. حافظه‌ی پایدار شبیه به حافظه‌ی سراسری است با این تفاوت که از حافظه‌ی سراسری بسیار کوچک‌تر و برای همه‌ی نخ‌ها فقط قابل خواندن است. حافظه‌ی texture مانند حافظه‌ی پایدار برای همه نخ‌ها قابل دسترسی و فقط قابل خواندن است. این حافظه برای انجام محاسبات گرافیکی کاربرد حافظه‌ی نهان را دارد (۸). در شکل ۲ سلسله مراتب حافظه و ارتباط آن‌ها با حافظه‌ی host و همچنین ارتباط آن‌ها با نخ‌ها به تصویر کشیده شده است.



شکل ۱– الگوی حافظه‌ی GPU و ارتباط آن‌ها با نخ‌ها

٤. موازی سازی الگوریتم بهینه‌سازی فاخته

تنها وابستگی موجود بین فاخته‌ها و محل‌های سکونت آن‌ها مربوط به داده‌هایی است که باید در پایان هر نسل بین فاخته‌ها به اشتراک گذاشته شوند. از این رو می‌توان ادعا کرد که COA ذاتاً موازی است. در جدول ۱ زمان اجرای قسمت‌های مختلف COA در اجرای سری قابل مشاهده هستند و می‌توان به راحتی گلوگاه‌های زمانی COA را شناسایی کرد.

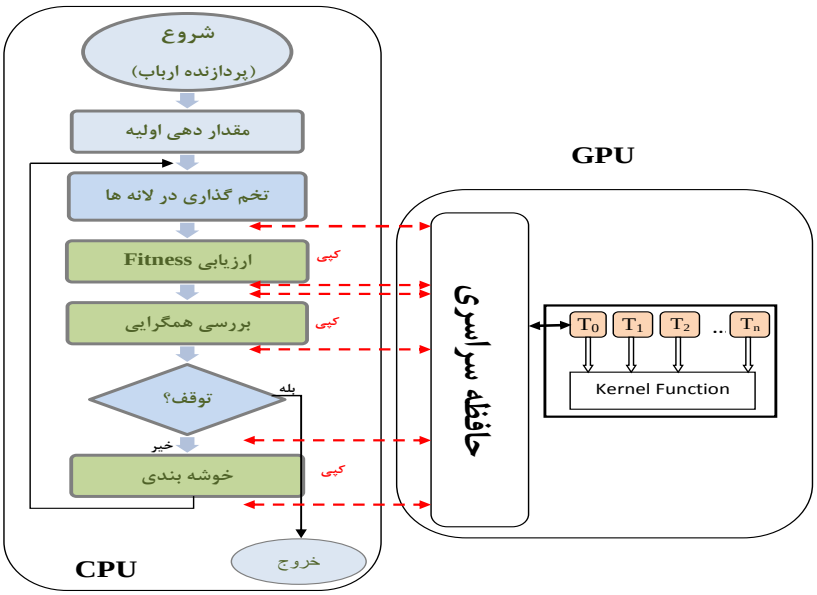
جدول ۱– محاسبه‌ی زمان قسمت‌های الگوریتم فاخته بر حسب درصد

phase	Percentage

[1] Host
[2] Device
[3] Kernel

initialization	0.013%
Egg laying	16.258%
Remove eggs in same positions	0.004%
Fitness function	42.801%
Kill cuckoo	0.761%
convergence	5.528%
Clustering cuckoo	29.782%
Select goal point	0.004%
Migration	4.849%
Total	100%

با توجه به زمان اجرای زیر توابع و مناسب بودن آن‌ها برای پیاده سازی موازی، کرنل‌ها قابل شناسایی می‌باشند. مرحله‌ی تخم گذاری، به علت وجود مشکلاتی در تولید اعداد تصادفی در حافظه‌ی کارت گرافیکی، موازی‌سازی نشده است. عملیات‌های محاسبه‌ی تابع برازندگی، همگرایی و خوشه بندی برای موازی سازی بر روی کارت گرافیکی انتخاب شده‌اند. ساده‌ترین تکنیک موازی سازی تابع برازندگی و دیگر توابع زمان‌گیر COA استفاده از الگوی ارباب – بنده است (۱۲). به این ترتیب که پردازنده‌ی ارباب، CPU و پردازنده‌های بنده، نخ‌های پردازشی GPU در نظر گرفته می‌شود؛ بنابراین مقدار دهی‌های اولیه بر روی میزبان انجام می‌گیرد و هرگاه لازم به موازی‌سازی باشد باید داده‌هایی را از روی حافظه‌ی میزبان به حافظه‌ی دستگاه منتقل گردد و بعد از انجام عملیات موازی روی GPU، نتایج بر روی حافظه‌ی میزبان کپی شوند. شکل ۲، فلوچارت COA مبتنی بر الگوی ارباب – بنده بر روی GPU می‌باشد.



شکل ۲- فلوچارت COA موازی با الگوی ارباب – بنده بر روی GPU

به علت اینکه الگوریتم پیشنهادی هم از حافظه‌ی سراسری و هم حافظه‌ی اشتراکی برای محاسبات موازی در همه‌ی کرنل‌ها استفاده می‌کند، این روش با عنوان PCOA-GSH نام گذاری شده است. هر نخ دارای یک شماره منحصر به فرد در بین کل نخ‌های تعریف شده می‌باشد. از طرفی هر نخ در بین نخ‌های یک بلاک نخ نیز دارای شماره منحصر به فرد دیگری است. به عبارت دیگر هر نخ به حافظه‌ی سراسری و اشتراکی با دو شماره‌ی شناسایی متفاوت دسترسی

دارد. بنابراین با داشتن این دو شماره، نخ می‌تواند داده‌ی مشخصی را از حافظه‌ی سراسری خوانده و برای انجام محاسبات، به مکان خاصی در حافظه‌ی اشتراکیِ بلاکش منتقل کند.

٤. ١- تکنیک‌های ریز دانه و درشت دانه(١٦)

شاید پایه‌ی تعریف موازی کردن برنامه‌ها، تقسیم کردن آن‌ها به قسمت‌های کوچک‌تر و حل آن قسمت‌ها به طور هم‌زمان بر روی چند پردازنده می‌باشد. در صورت امکان خود این قسمت‌های کوچک می‌توانند به قسمت‌های کوچک‌تر تقسیم شوند. بسته به مقدار وابستگی بین داده‌ها در اجرای موازی، مدت زمانی که صرف ارتباطات بین پردازنده‌ها می‌شود به زمان اجرا اضافه می‌گردد. در برنامه‌ها، با توجه به میزان وابستگی داده‌ای، هر چه برنامه به قسمت‌های کوچک‌تر و بیشتر تقسیم شود از زمان محاسبات برای هر پردازنده کاسته و به زمان ارتباطات اضافه می‌شود. مفهوم دانه دانه بودن[1] به همین موضوع اشاره می‌کند. در برنامه‌های درشت دانه برای هر پردازنده پردازش سنگین و ارتباطات کم هستند. در ریز دانه به علت کوچک بودن قسمت‌های محاسباتی، برای هر پردازنده محاسبات کم و ارتباطات بین آن‌ها زیاد است. اینکه بین ریز دانه و درشت دانه کدام یک بهتر است نمی‌توان نظر ثابتی داد. باید توجه داشت که هر چه وابستگی بین داده‌ها زیاد باشد نسبت زمان محاسبات به زمان ارتباطات کمتر می‌شود و کارایی پایین می‌آید. درشت دانه از جهتی که همگامی آسان‌تر و ارتباطات کمتر رخ می‌دهد بهتر است ولی عیب این روش بالا بودن حجم پردازش برای هر پردازنده است.

برای حرکت از درشت دانه به سمت ریز دانه باید وظایف هر نخ را به قسمت‌های کوچک‌تری تقسیم کرد به عنوان مثال به جای اینکه یک نخ ٨١٩٢ مرتبه اجرا شود ٨١٩٢ نخ تعریف کرد که همگی به طور موازی محاسبات را بر روی یک ورودی تابع محک انجام می‌دهند. در COA تنها ارتباط بین داده‌های یک عضو خانواده زمانی است که بعد از انجام محاسبات باید نتایج برای بدست آوردن جواب نهایی جمع آوری شوند. به عبارت دیگر ارتباطی بین نخ‌ها در حین محاسبات وجود ندارد و این موازی بودن ذاتی COA را اثبات می‌کند. این تغییر موازی کردن از درشت دانه به ریز دانه باعث کاهش زمان الگوریتم موازی و بهبود در کارایی می‌شود.

٤. ١- کرنل برازندگی

آرایه‌ی ورودی به کرنل برازندگی آرایه‌ای یک بعدی است که از نگاشت یک آرایه دو بعدی حاصل می‌شود. این آرایه دو بعدی به تعدادِ جمعیت، سطر و به تعداد ابعاد مسأله، ستون دارد. پس تعداد نخ‌های پردازشی باید به همین تعداد باشند. تعداد بلاک‌های نخ برابر با جمعیت فاخته‌ها و تعداد نخ‌های هر بلاک نخ برابر با ابعاد مسأله در نظر گرفته شده است. با توجه به فضای ذخیره سازی حافظه اشتراکی و محدودیت در تعداد نخ‌های یک بلاک نخ که حداکثر ١٠٢٤ نخ هستند این پیاده سازی برای مسائلی تا ابعاد ١٠٢٤ قیاس پذیر است. بعد از اینکه نخ‌های هر بلاک نخ محاسبات را بر روی داده‌ی خود انجام دادند نتیجه را در آرایه‌ای در حافظه اشتراکی بلاکشان ذخیره می‌کنند. برای پیدا کردن جواب نهایی لازم است عملیات کاهش بر روی آرایه‌ی مذکور صورت گیرد. در شکل ٣ نحوه فراخوانی کرنل برازندگی در میزبان نشان داده شده است.

[1] Granularity

1: Mapping *Array2Dim* to *Array1Dim*
2: Copy *Array1Dim* on Device Global Memory
3: Copy *Array_Profits* on Device Global Memory
4: Define *blockNum* = *CuckooPop*
5: Define *threadNum* = *Number_Of_problem's_Parameters*
6: *Fitness_Kernel*<<<*blockNum,threadNum*>>>(*Array1Dim,Array_profits*)
7: Copy *Array_Profits* on Host Memory

شکل ۳- نحوه‌ی فراخوانی کرنل برازندگی در میزبان

٤. ۲- کرنل همگرایی

ورودی این کرنل آرایه‌ای است یک بعدی که از نگاشت آرایه دو بعدی، که سطرهایش جمعیت فاخته‌ها و ستون‌هایش تعداد ابعاد مسأله می‌باشد، حاصل شده است. نگاشت آرایه دو بعدی به یک بعدی به صورت ستونی می‌باشد. زیرا عملیات در پیدا کردن همگرایی نتایج بر روی ستون‌ها می‌باشد. بنابراین مقادیر هر ستون باید در داخل حافظه‌ی اشتراکی یک بلاک قرار گیرند. در اینجا هم به هر عنصر آرایه‌ی ورودی یک نخ اختصاص داده می‌شود. تعداد بلاک‌ها برابر با تعداد ابعاد مسأله و تعداد نخ‌های هر بلاک برابر با تعداد ستون‌ها یعنی به تعداد جمعیت فاخته می‌باشد. به دلیل محدودیت در تعداد نخ‌های یک بلاک، تعداد جمعیت نباید از ۱۰۲٤ فاخته تجاوز کند. در انتهای کرنل، برای یافتن مقدار مجموع همگرایی، عمل کاهش در هر بلاک نخ انجام می‌گیرد. در شکل ٤ نحوه فراخوانی کرنل همگرایی در میزبان مشاهده می‌شود.

1: Mapping *Array2Dim* to *Array1Dim*
2: Copy *Array1Dim* on Device Global Memory
3: Copy *Result_value* on Device Global Memory
4: Define *blockNum* = *Number_Of_problem's_Parameters*
5: Define *threadNum* = *CuckooPop*
6: *Convergence_Kernel*<<<*blockNum,threadNum*>>>(*Array1Dim,Result_value*)
7: Copy *Result_value* on Host Memory

شکل ٤- نحوه‌ی فراخوانی کرنل همگرایی در میزبان

به خاطر محدودیت در تعداد نخ‌های یک بلاک، تعداد جمعیت نباید از ۱۰۲٤ فاخته تجاوز کند. در انتهای کرنل، برای یافتن مقدار مجموع همگرایی، عمل کاهش در هر بلاک نخ انجام می‌گیرد.

۲.۴- کرنل خوشه بندی

در COA از روش K-means برای خوشه بندی فاخته‌ها استفاده شده است. ورودی این کرنل، از نگاشت آرایه‌ی دو بعدی به یک بعدی به دست می‌آید. نتیجه‌ی عملیات خوشه بندی تعیین کردن شماره خوشه‌ی هر یک از فاخته‌هاست. بنابراین هر بلاک باید مقدار شماره خوشه برای هر فاخته را برگرداند. پس می‌توان تعداد نخ‌های هر بلاک را برابر با تعداد ابعاد مسأله در نظر گرفت. تعداد بلاک‌های نخ برابر با تعداد جمعیت در نظر گرفته می‌شود. شکل ۵ فراخوانی کرنل خوشه بندی در میزبان را نشان می‌دهد.

1: Mapping *Array2Dim* to *Array1Dim*
2: Copy *Array1Dim* on Device Global Memory
3: Copy *Array_NumCluster* on Device Global Memory
4: Define *blockNum = CuckooPop*
5: Define *threadNum = Number_Of_problem's_Parameters*
6: *K-means_Kernel<<<blockNum,threadNum>>>(Array1Dim,Array_NumCluster)*
7: Copy *Array_NumCluster* on Host Memory

شکل ۵- نحوه‌ی فراخوانی کرنل خوشه بندی در میزبان

۵- آزمایش‌ها و تحلیل نتایج

در این بخش، نسخه‌ی ارائه شده بر روی بعضی از توابع محک رایج، که به منظور مقایسه‌ی روش‌های مختلف بهینه سازی استفاده می‌شوند، پیاده سازی شده است. نتایج دو آزمایش متفاوت مورد بررسی قرار گرفته‌اند. اولین آزمایش، رفتار الگوریتم ارائه شده را با ثابت نگه داشتن تعداد ابعاد مسأله و تعداد تکرارها و افزایش تعداد فاخته‌ها مورد بررسی قرار می‌دهد. آزمایش بعدی با ثابت نگه داشتن تعداد فاخته‌ها و تعداد تکرار و افزایش تعداد ابعاد مسأله الگوریتم موازی را مورد بررسی قرار می‌دهد. توابع محک مورد استفاده برای انجام شبیه سازی‌ها در زیر لیست شده‌اند.

$$F1 = \sum_{i=1}^{n} x_i^2$$

$$F2 = \sum_{i=1}^{n} \sum_{j=1}^{i} x_j^2$$

$$F3 = \sum_{i=1}^{n-1} (100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2)$$

$$F4 = 10n + \sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i))$$

$$F5 = \sum_{i=1}^{n} (-x_i \sin(\sqrt{|x_i|}))$$

$$F6 = -a \cdot \exp\left(-b. \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n} \cos(cx_i)\right) + a + \exp(1)$$

بعضی از خصوصیات اصلی توابع محک نام برده در جدول ۲ ذکر شده‌اند. این خصوصیات شامل فضای جستجو و مقدار مینیمم سراسری هر تابع تست می‌باشد.

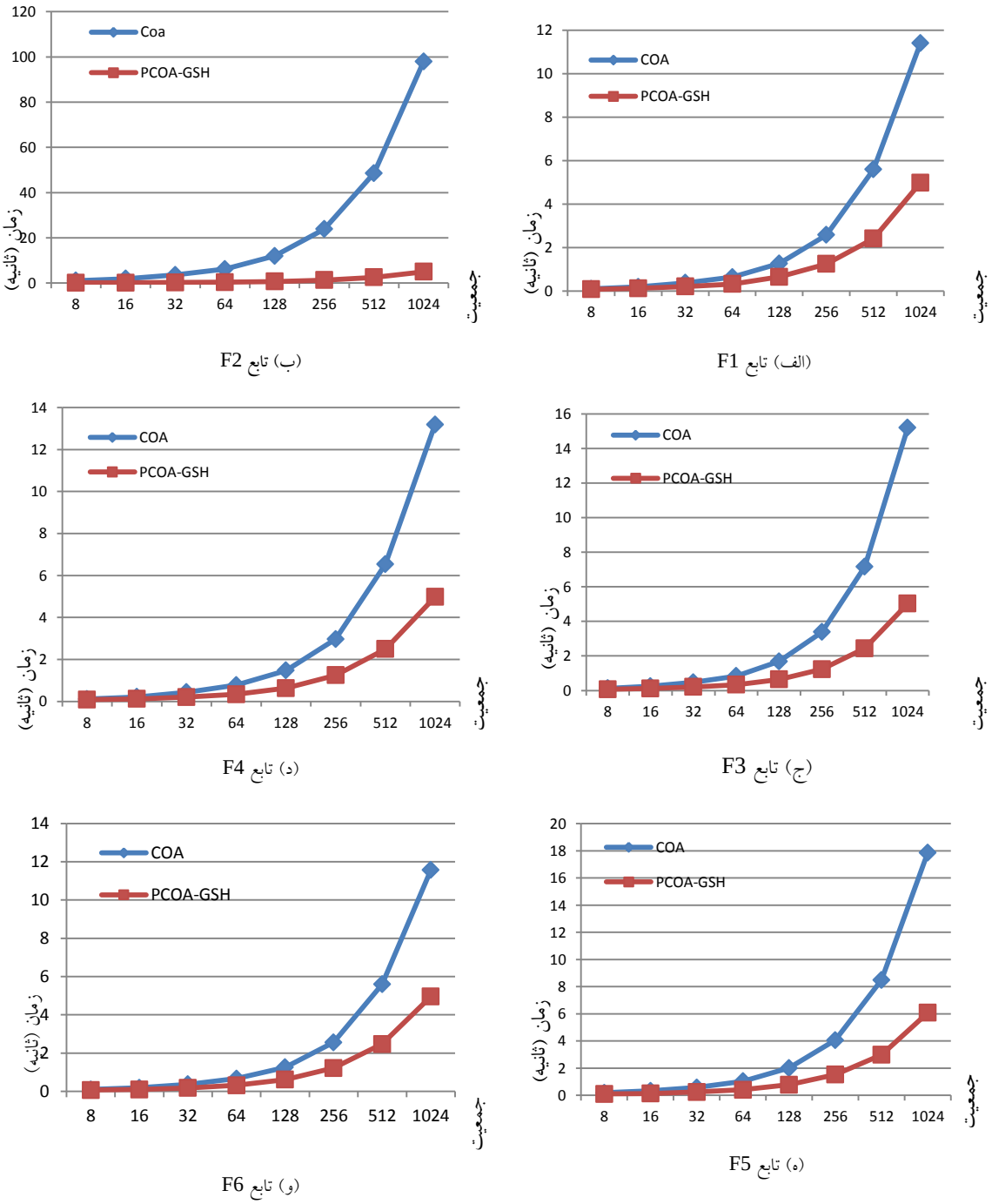
در این آزمایش‌ها از یک پردازنده‌ی intel® core™ i7-2630QM CPU @ 2.00GHz با حافظه‌ی ٦/٠٠ گیگا بایت و همچنین یک کارت گرافیک NVIDIA GeForce GT 540M با حافظه‌ی ١/٠٠ گیگا بایت، سری Fermi با ظرفیت ٢/٠ استفاده شده است. پیاده سازی‌ها با استفاده از آخرین نسخه‌ی CUDA (تا این زمان نسخه ۵ منتشر شده است) و Visual studio 2010 بر روی windows 7 توسعه یافته‌اند. تعداد حداکثر تخم‌های هر فاخته برای همه‌ی آزمایش‌ها ۳ تخم می‌باشد. زمان متوسط بر حسب ثانیه برای ۳۰ اجرا اندازه‌گیری شده است و از آن جایی که نتایجِ انحراف معیار برای مقادیر اندازه‌گیری شده، کوچک بود، گزارش نشده است.

جدول ۲- خصوصیات توابع محک نام برده

نام تابع	فضای جستجو	مقدار مینیمم سراسری
F1	$[-5.12, 5.12]^n$	0
F2	$[-65.536, 65.536]^n$	0
F3	$[-2.048, 2.048]^n$	0
F4	$[-5.12, 5.12]^n$	0
F5	$[-500, 500]^n$	-418.9829
F6	$[-32.768, 32.768]^n$	0

۵. ۱- آزمایش اول

در این آزمایش تنوع فاخته‌ها از یک فاخته تا ۱۰۲٤ فاخته متغیر است. ابعاد مسأله ثابت و با ۱۰۰ مقدار دهی شده است. تعداد تکرار نیز ثابت و برابر با ۱۰۰ در نظر گرفته شده است. شکل ٦ نتایج آزمایش اول را نشان می‌دهد.



(الف) تابع F1

(ب) تابع F2

(ج) تابع F3

(د) تابع F4

(ه) تابع F5

(و) تابع F6

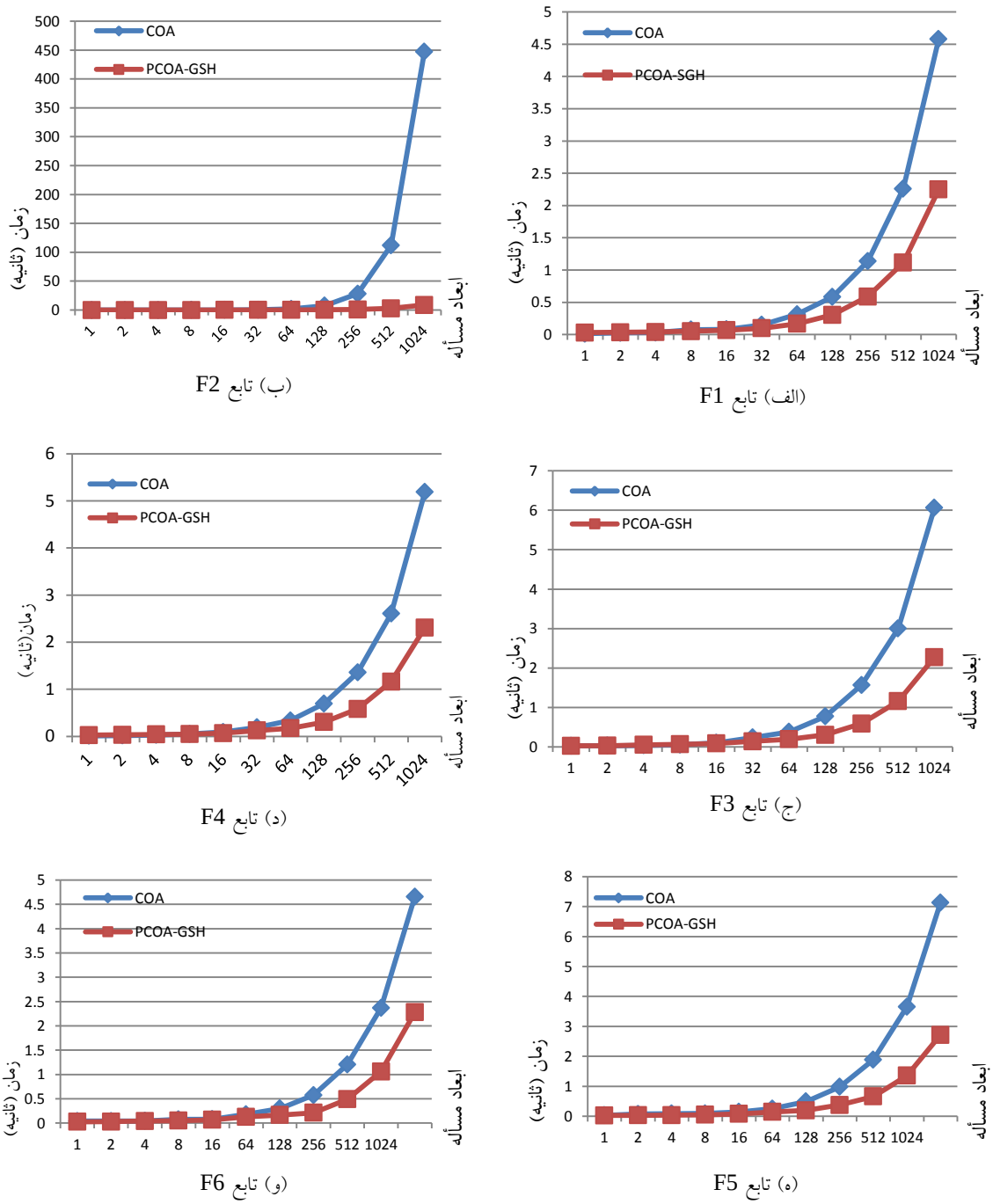
شکل ٦- نتایج آزمایش اول بر روی ٦ تابع محک نام برده.

در شکل ٦ قسمت (الف) برای تابع F1 در حالتی که جمعیت فاخته‌ها تا ١٠٢٤ افزایش یابد بهره‌ی سرعت برابر است با ٢/٩٠ است. قسمت (ب)، تابع F2، تفاوت میان اجرای سری و موازی است که برای افزایش جمعیت تا ١٠٢٤، بهره‌ی سرعت برابر است با ١٩/٥٤ قسمت (ج) شکل مذکور نتایج را برای تابع F3 نشان می‌دهد که برای افزایش جمعیت تا ١٠٢٤، بهره‌ی سرعت برابر به ٣/٠٣ و برای قسمت (د) شکل، با افزایش جمعیت تا ١٠٢٤ نتایج بهره‌ی سرعت برای تابع F4 برابر است با ٢/٦٥. قسمت (ه) برای تابع F5 در حالتی که جمعیت فاخته‌ها تا ١٠٢٤

افزایش یابد بهره‌ی سرعت برابر است با ۲/۹۳ است. قسمت (ب) شکل ۵، تابع F6 تفاوت میان اجرای سری و موازی است که برای افزایش جمعیت تا ۱۰۲٤، بهره‌ی سرعت برابر است با ۲/۳۷.

۵. ۲- آزمایش دوم

در این آزمایش تنوع ابعاد مسأله از یک تا ۱۰۲٤ بُعد متغیر است. تعداد جمعیت فاخته‌ها ثابت و حداکثر برابر با ٤۵ در نظر گرفته شده است. تعداد تکرار مانند آزمایش قبل ثابت فرض شده است. شکل ۷ نتایج این آزمایش زا گزارش می-دهد.



شکل ۷- نتایج آزمایش اول بر روی ٦ تابع محک نام برده.

در شکل ۷ قسمت (الف)، مقایسه زمان اجرای الگوریتم فاخته سری و موازی برای تابع F1، در حالتی که ابعاد مسأله تا ۱۰۲٤ افزایش یابد بهره‌ی سرعت به ۲/۰٤ می‌رسد. قسمت (ب) شکل ۷ تفاوت میان اجرای سری و موازی تابع F2 است که برای افزایش ابعاد مسأله تا ۱۰۲٤، بهره‌ی سرعت برابر است با ۵۰/۷۸. قسمت (ج) نتایج مقایسه زمان اجرای الگوریتم فاخته سری و موازی را برای ارزیابی F3 نشان می‌دهد که برای افزایش ابعاد مسأله تا ۱۰۲٤، بهره‌ی سرعت برابر است با ۲/٦۷ و برای قسمت (د) شکل، مقایسه زمان اجرای الگوریتم سری و موازی فاخته برای تابع F4 است و همان طور که مشخص است، با افزایش ابعاد تا ۱۰۲٤ نتایج بهره‌ی سرعت برابر با ۲/۲۵ است. قسمت (ه) نتایج مقایسه زمان اجرای الگوریتم فاخته سری و موازی را برای ارزیابی F5 نشان می‌دهد که برای افزایش ابعاد مسأله تا ۱۰۲٤، بهره‌ی سرعت برابر است با ۲/٦۲ و برای قسمت (و) شکل، مقایسه زمان اجرای الگوریتم سری و موازی فاخته برای تابع F6 است و همان طور که مشخص است، با افزایش ابعاد تا ۱۰۲٤ نتایج بهره‌ی سرعت برابر با ۲/۰۵ است.

٦- نتیجه‌گیری

الگوریتم بهینه‌سازی فاخته‌ی موازی ریزدانه مبتنی بر الگوی ارباب – بنده بر روی GPU پیاده سازی شده است. در الگوی ارباب – بنده، پردازنده ارباب CPU و پردازنده‌های بنده GPU در نظر گرفته شده است. برنامه در گره‌ی ارباب مقداردهی اولیه و اجرا می‌شود و در مراحلی که احتیاج به موازی‌سازی باشد داده‌های مورد نیاز از طریق حافظه‌ی سراسری به GPU منتقل می‌شوند. مراحلی از الگوریتم که در اجرا، زمان بیشتری از CPU را به خود اختصاص می‌دهند به وسیله‌ی عنصری نرم افزاری به نام کرنل روی GPU فراخوانی می‌شود. نخ‌های پردازشی، کرنل را به صورت هم‌زمان اجرا می‌کنند هر نخ داده‌های مرتبط با شناسه خود را از آرایه در حافظه سراسری می‌خواند و در حافظه اشتراکی کپی می‌کند و نتایج در انتهای کرنل با عمل کاهش محاسبه و از طریق حافظه سراسری به گره‌ی ارباب بر می‌گردانند. در الگوریتم پیشنهادی محاسبه تابع برازندگی، همگرایی و خوشه‌بندی بر روی GPU پیاده سازی شده است.

نتایج آزمایش‌ها مشخص می‌کند:

۱- تغییر ابعاد مسأله و جمعیت الگوریتم به سمت بهره‌ی سرعت بهتر برای نسخه‌ی GPU پیش می‌رود. واضح است که افزایش تعداد تکرار برای محاسبه‌ی تخمین توابع مذکور نتایج مشابهی را در بر دارد. با توجه به آزمایش‌های انجام شده می‌توان به این نتیجه رسید که در مواقعی که هم تعداد ابعاد مسأله و هم تعداد جمعیت زیاد باشند بهره‌ی سرعت بهتری مشاهده می‌شود.

۲- در طبقه‌بندی حافظه‌ها در GPU این نتیجه حاصل شد که دسترسی نخ‌ها به حافظه‌ی سراسری نسبت به حافظه‌های اشتراکی و محلی و همچنین تکرار عملیات کپی کردن داده‌ها بین CPU و GPU از طریق حافظه‌ی سراسری با تأخیر بسیار زیادی صورت می‌گیرد، از این رو تا جایی که امکان دارد از حافظه اشتراکی استفاده می‌شود و تنها برای ارتباط با میزبان حافظه سراسری به کار گرفته می‌شود.

مراجع

[1] E. Cantú-Paz. 1998. A survey of parallel genetic algorithms, Calculateurs Paralleles. Vol. 10,No. 2. Paris.

[2] M.G. Arenas, A.M. Mora, G. Romero, and P.A. Castillo. 2011. GPU Computation in Bioinspired Algorithms: A Review, in Proc. IWANN (1).

[3] M.L. Wong, T. Wong, and K. Fok. 2005. Parallel evolutionary algorithms on graphics processing unit, in Proc. Congress on Evolutionary Computation.
[4] S. Harding and W. Banzhaf. 2007. Fast Genetic Programming and Artificial Developmental Systems on GPUs", in Proc. HPCS.
[5] S. Zhang and Z. He. 2009. Implementation of Parallel Genetic Algorithm Based on CUDA", in Proc. ISICA.
[6] R. Rajabioun. 2011. Cuckoo Optimization Algorithm", presented at Appl. Soft Comput., Vol 11, No 8.
[7] M. Waintraub1, R. Schirru and C.M.N.A. Pereira. 2009. parallel particle swarm optimization algorithm in nuclear problems, in International Nuclear Atlantic Conference, INAC.
[8] NVIDIA, nVIDIA CUDA Programming Guide v5.0 , nVIDIA Corporation, 2012, http://www.nvidia.com.
[9] L. Mussi, F. Daolio, and S. Cagnoni. 2011. Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture, presented at Inf. Sci.,Vol 181, No 20.
[10] NVIDIA, nVIDIA CUDA C Programming - Best Practices Guide v5.0 , nVIDIA Corporation, 2012, http://www.nvidia.com.
[11] E. Alba and M. Tomassini. 2002. Parallelism and evolutionary algorithms, presented at IEEE Trans. Evolutionary Computation, Vol. 6, No 5.
[12] W. Zhu and J. Curry. 2009. Parallel Ant Colony with Local Pattern Search for Nonlinear Function Optimization with Graphics Hardware Acceleration, in Proc. SMC.
[13] NVIDIA, nVIDIA Thrust Quick Start Guide v5.0 , nVIDIA Corporation, 2012, http://www.nvidia.com.
[14] R.L. Haupt, S.E. Haupt. 2004. Practical Genetic Algorithms, second ed., John Wiley& Sons, New Jersey.
[15] Yu, Q., Chen, C., Pan, Z. 2005. Parallel genetic algorithms on programmable graphics hardware. In: L.W. et al. (ed.) Advances in Natural Computation, Lecture Notes in Computer Science, vol. 3612. Springer Berlin / Heidelberg.
[16] V. Kumar, A. Grama, A. Gupta, and G. Karypis. 1994. Introduction to Parallel Computing.