دوره ۱۳، شماره پاییز و زمستان ۱۳۹۹

مجله فناوری اطلاعات در طراحی مهندسی
Information Technology in Engineering Design
http://ited.sinaweb.net

# موازی‌سازی الگوریتم فراابتکاری پروانه با استفاده از قابلیت‌های معماری مبتنی بر چند پردازنده گرافیکی

محمد کریم سهرابی *(۱) عباس زورق چیان (۲) فرزین یغمایی (۳)

(۱) گروه مهندسی کامپیوتر، واحد سمنان، دانشگاه آزاد اسلامی، سمنان، ایران*

(۲) گروه مهندسی کامپیوتر، واحد سمنان، دانشگاه آزاد اسلامی، سمنان، ایران

(۳) گروه مهندسی کامپیوتر، واحد سمنان، دانشگاه آزاد اسلامی، سمنان، ایران

(تاریخ دریافت : ۱۳۹۹/۹/۲۵ تاریخ پذیرش : ۱۴۰۰/٤/۲٦)

## چکیده:

بسیاری از مسائل و چالش‌های پیرامون ما از نوع بهینه‌سازی هستند و برای یافتن جواب بهینه در آنها می‌توان از روش‌های مبتنی بر الگوریتم‌های فراابتکاری استفاده کرد. دسته مهمی از الگوریتم‌های فراابتکاری، از رفتار جانداران برای بقا یا رفتارهای زیستی و فیزیکی الگوبرداری شده است. این الگوریتم‌ها توانایی بالایی برای یافتن جواب بهینه دارند. در مسائل پیچیده‌تر لازم است که اندازه جمعیت مساله به اندازه کافی بزرگ باشد تا خطای محاسباتی آن کاهش داده شود. چالش مهم در این مورد افزایش زمان اجرای الگوریتم مورد نظر برای یافتن جواب بهینه است. مزیت مهم الگوریتم‌های فراابتکاری مثل الگوریتم بهینه‌سازی پروانه در این است که یافتن جواب بهینه در این روش‌ها مبتنی بر جمعیت بوده و هر عضو جمعیت می‌تواند به طور مستقل فضای مساله را مورد جستجو قرار دهد تا جواب بهینه را اکتشاف نماید. بر مبنای این مزیت، می‌توان این الگوریتم‌های فراابتکاری را در بستر موازی مانند پردازنده‌های گرافیکی اجرا نمود و زمان اجرای آنها را کاهش داد. در این مقاله یک روش کارآمد برای موازی‌سازی الگوریتم بهینه‌سازی پروانه ارایه شده است که از امکانات دو پردازنده گرافیکی برای تسریع محاسبات استفاده می‌نماید. نتایج پیاده‌سازی بر روی دو پردازنده گرافیکی نشان می‌دهد که استفاده از دو پردازنده شتاب اجرایی را، نسبت به حالتی که از یک پردازنده گرافیکی استفاده شده است، به طور قابل ملاحظه‌ای بهبود می‌بخشد که متناسب با افزایش اندازه جمعیت اولیه این الگوریتم‌ها است.

**کلمات کلیدی:** موازی‌سازی، پردازنده گرافیکی، کودا، الگوریتم بهینه‌سازی پروانه

* عهده‌دار مکاتبات:

محمد کریم سهرابی

نشانی: گروه مهندسی کامپیوتر، واحد سمنان، دانشگاه آزاد اسلامی، سمنان، ایران

پست الکترونیکی: amir_sohraby@aut.ac.ir تلفن : ۰۹۳۷٤۷۸۹۰۸۰

## ۱-مقدمه

الگوبرداری از رفتار جانداران مخصوصا رفتارهای گروهی آنها در مواجهه با مشکلات محیطی یک روش جالب و در عین حال موثر برای حل برخی مسائل پیچیده نظیر مسائل بهینه‌سازی است. الگوریتم‌های فراابتکاری[۱] با الگوبرداری از پدیده‌های موجود در طبیعت تلاش می‌کنند تا راه‌حل‌های بهینه را استخراج نمایند. روش‌های فراابتکاری بر خلاف روش‌های مبتنی بر گرادیان[۲] نیازی به محاسبه گرادیان یا مشتق تابع هدف[۳] یا تابع هزینه ندارند و جواب‌هایی ارایه می‌دهند که در بیشتر موارد نزدیک جواب بهینه هستند [۱]. مکانیزم حل مساله دسته‌جمعی جانداران بر اساس قانون انتخاب طبیعی[٤] است به گونه‌ای که افراد شایسته شانس بقای بیشتری داشته و نقش آنها برای هدایت سایر افراد نیز بیشتر است [۲]. رفتارهای دسته‌جمعی در طبیعت یکی از روش‌های ایجاد الگوریتم‌های فراابتکاری می‌باشند و از آنها به عنوان الگوریتم‌های هوش جمعی یاد می‌شود. الگوریتم‌های هوش جمعی نوعی از الگوریتم‌های فراابتکاری مبتنی بر جمعیت به شمار می‌روند که در آنها، در هر مرحله تعدادی راه‌حل فضای جستجوی مساله را برای یافتن جواب بهینه مورد جستجو قرار می‌دهند [۳]. در الگوریتم‌های هوش جمعی، هر عضو جمعیت تلاش می‌نماید که با استفاده از اطلاعات خود و سایر اعضا به سمت پاسخ‌های بهینه مساله حرکت کند. در روش‌های مبتنی بر هوش جمعی اعضای شایسته‌تر نقش بیشتری در هدایت سایر اعضا به سمت جواب بهینه بر عهده دارند.

هوش جمعی را می‌توان برآیند هوشمندی کل اعضای یک جمعیت برای یافتن جواب بهینه در نظر گرفت. تاکنون الگوریتم‌های مختلفی بر پایه هوش جمعی به عنوان روش‌های موثر الگوریتم‌های فراابتکاری ارایه شده است که در بسیاری از آنها رفتار یک جاندار در طبیعت مورد الگوبرداری قرار گرفته شده است. مزیت مهم این نوع الگوریتم‌ها در آن است که فضای جستجوی مساله قابل کاوش به صورت موازی و مستقل توسط هر یک از اعضای جمعیت می‌باشد و اگر یک یا چند راه‌حل در بهینه‌های محلی مساله گرفتار شوند، سایر اعضا این شانس را دارند تا بهینه‌های سراسری را پیدا کنند [٤]. تاکنون الگوریتم‌های مختلفی بر پایه رفتار هوش جمعی ارایه و معرفی شده است که از جمله آنها می‌توان به بهینه‌سازی گرگ خاکستری [۵]، الگوریتم شیر مورچه [٦]، الگوریتم بهینه‌سازی وال [۷]، الگوریتم بهینه‌سازی غذایابی باکتری [۸]، الگوریتم بهینه‌سازی عنکبوت اجتماعی [۹]، بهینه‌سازی کفتار [۱۰] و بهینه‌سازی پروانه [۱۱] اشاره کرد. برای آنکه دقت این روش‌ها افزایش یابد، لازم است اندازه جمعیت و تعداد تکرارهای الگوریتم آنها به قدر کافی بزرگ در نظر گرفته شوند تا خطای آنها در محاسبات کاهش یابد.

یکی از چالش‌های اصلی این روش‌ها، افزایش زمان زمان اجرا، به ویژه در زمان‌هایی است که برای افزایش دقت، اندازه جمعیت آنها افزایش می‌یابد. به عبارت دیگر، یکی از مهمترین چالش‌های الگوریتم‌های فراابتکاری زمان اجرای آنها است که این چالش با افزایش اندازه جمعیت اولیه و پیچیدگی مساله بیشتر و حادتر می‌شود. یکی از روش‌های موثر غلبه بر این چالش و کاهش زمان اجرای الگوریتم‌های فراابتکاری، استفاده از روش‌های موازی‌سازی در بستر معماری کامپیوتر است. پردازنده‌های گرافیکی[۵] [۱۲] .

---

[۱] Meta-heuristic algorithms
[۲] Gradient-based
[۳] Objective function
[٤] Natural selection
[۵] Graphical Processing Unit (GPU)

برخلاف پردازنده‌های معمولی[۶] دارای هسته‌های متعددی برای پردازش هستند و معماری آنها به گونه‌ای توسعه داده شده است که قابلیت انجام عملیات ساده در حجم بسیار بالا دارند و به این ترتیب، توانایی زیادی برای موازی‌سازی و تسریع محاسبات از خود نشان می‌دهند. چارچوب کودا [۱۳] یک بستر نرم‌افزاری برای موازی‌سازی است که برای تسریع محاسبات از معماری پردازنده‌های گرافیکی استفاده می‌کند. در این مقاله، یک نسخه موازی و تسریع شده از الگوریتم بهینه‌سازی پروانه را در معماری موازی پردازنده-های گرافیکی ارائه می‌شود که میزان سرعت اجرای این الگوریتم را نسبت به حالت‌های سریال و موازی‌سازی شده با پردازنده‌های معمولی بیشتر می‌کند. پس از آن، یک معماری کارآمد مبتنی بر بیش از یک پردازنده گرافیکی ارائه می‌گردد که موازی‌سازی را در سطح چند پردازنده گرافیکی عملیاتی می‌کند تا تاثیر این معماری بر شتاب و زمان اجرای الگوریتم پیشنهادی را بررسی نماید.

سازمان‌دهی بخش‌های بعدی مقاله به این صورت است که در ابتدا الگوریتم فراابتکاری پروانه و معماری کودا به عنوان پیش نیازهای مساله، در بخش ۲ مورد بحث قرار می‌گیرند و سپس روش موازی شدن الگوریتم فراابتکاری پروانه در چارچوب معماری مورد نظر مقاله، در بخش ۳ تشریح می‌شود. در ادامه نتایج آزمایش‌های عملی که در بستر کودا انجام شده‌اند، در بخش ٤ مورد بررسی قرار می‌گیرند و نتایج پژوهش تحلیل می‌شوند. جمع‌بندی کلی و نهایی از روش مورد استفاده و نتایج به دست آمده از آن، در پایان مقاله و در بخش نتیجه‌گیری صورت می‌گیرد.

## ۲- پیش‌نیازهای پژوهش

در این بخش به توصیف الگوریتم بهینه‌سازی پروانه[۷] و معماری کودا[۸] به عنوان دو پیش نیاز اصلی این مقاله می‌پردازیم و نکات مهم مرتبط با آنها را مورد بررسی قرار می‌دهیم.

### ۲-۱-الگوریتم بهینه سازی پروانه

یکی از الگوریتم فراابتکاری گروهی الگوریتم بهینه‌سازی پروانه است که در آن از رفتار پروانه‌ها در انتشار و جذب فرومون شیمیایی الگوبرداری شده است. در این الگوریتم، هر پروانه یک راه‌حل برای مساله است و بر اساس میزان شایستگی خود مقداری فرومون در هوا منتشر می‌کند. هر پروانه در این الگوریتم می‌تواند دو نوع حرکت انجام دهد. حرکت یک پروانه به سوی بیشترین مقدار فرومون موجود در فضای مساله که توسط شایسته‌ترین پروانه منتشر می‌گردد، با یک احتمال تصادفی انجام می‌شود. در واقع این حرکت پروانه نوعی جستجوی محلی برای یافتن جواب بهینه است. پس از آن، هر پروانه دو پروانه از جمعیت را به صورت تصادفی انتخاب نموده و به سمت فرومون منتشر شده توسط آن پروانه‌ها جذب می‌شود. این حرکت پروانه‌ها در حقیقت نوعی جستجوی سراسری برای یافتن پاسخ بهینه کلی است. برای مدل‌سازی حرکت اول از رابطه (۱)، و برای مدل‌سازی حرکت دوم از رابطه (۲) استفاده می‌شود [۱۱]:

---

[۶] Central Processing Unit (CPU)
[۷] Butterfly optimization algorithm
[۸] CUDA

## ۲-۱-الگوریتم بهینه سازی پروانه

یکی از الگوریتم فراابتکاری گروهی الگوریتم بهینه‌سازی پروانه است که در آن از رفتار پروانه‌ها در انتشار و جذب فرومون شیمیایی الگوبرداری شده است. در این الگوریتم، هر پروانه یک راه‌حل برای مساله است و بر اساس میزان شایستگی خود مقداری فرومون در هوا منتشر می‌کند. هر پروانه در این الگوریتم می‌تواند دو نوع حرکت انجام دهد. حرکت یک پروانه به سوی بیشترین مقدار فرومون موجود در فضای مساله که توسط شایسته‌ترین پروانه منتشر می‌گردد، با یک احتمال تصادفی انجام می‌شود. در واقع این حرکت پروانه نوعی جستجوی محلی برای یافتن جواب بهینه است. پس از آن، هر پروانه دو پروانه از جمعیت را به صورت تصادفی انتخاب نموده و به سمت فرومون منتشر شده توسط آن پروانه‌ها جذب می‌شود. این حرکت پروانه‌ها در حقیقت نوعی جستجوی سراسری برای یافتن پاسخ بهینه کلی است. برای مدل‌سازی حرکت اول از رابطه (۱)، و برای مدل‌سازی حرکت دوم از رابطه (۲) استفاده می‌شود [۱۱]:

$$\boldsymbol{x_i^{t+1} = x_i^t + (r^2 \times g^* - x_i^t) \times f_i} \quad .۱$$

$$\boldsymbol{x_i^{t+1} = x_i^t + (r^2 \times x_j^t - x_k^t) \times f_i} \quad .۲$$

دراین روابط، $x_i^t$ و $x_i^{t+1}$ به ترتیب موقعیت فعلی و جدید یک پروانه هستند و $f_i$ نیز میزان جذابیت یک پروانه است که متناسب با میزان شایستگی آن پروانه است. همچنین، $x_j^t$ و $x_k^t$ موقعیت فعلی دو پروانه در جمعیت هستند که یک پروانه به سمت فرومون منتشر شده توسط آنها پرواز می‌کند.

## ۲-۲-معماری کودا

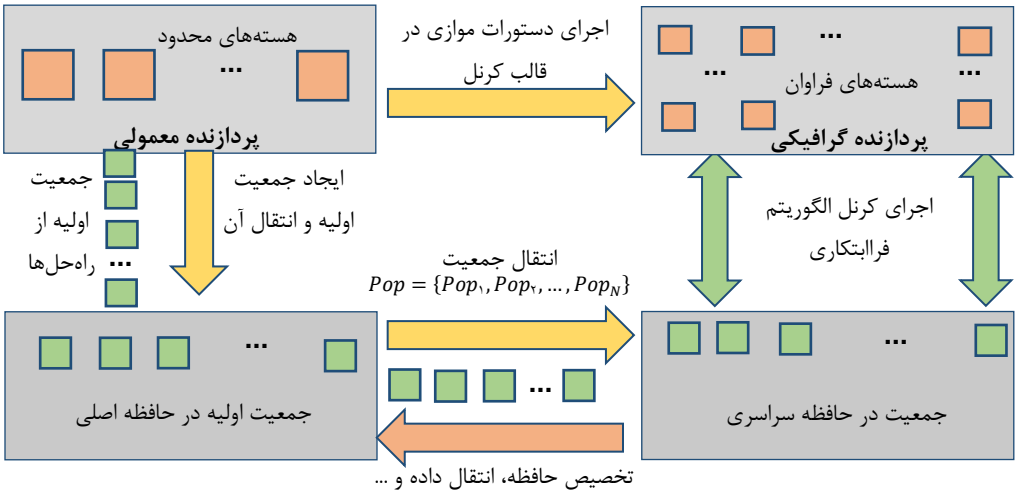
پردازنده‌های گرافیکی به علت داشتن هسته‌های پردازشی بیشتر دارای توان موازی‌سازی بالاتری نسبت به پردازنده‌های معمولی هستند و این موضوع موجب گسترش کاربردهای این پردازنده‌ها شده است. از این پردازنده‌ها برای موازی‌سازی و تسریع الگوریتم-های پیچیده استفاده می‌شود. پردازنده معمولی، یک پردازنده عمومی و سراسری است که با استفاده از آنها می‌توان انواع عملیات محاسباتی و پردازشی را انجام داد. این پردازنده در اختیار تمام برنامه‌ها قرار می‌گیرد. این در حالی است که پردازنده گرافیکی یک پردازنده اختصاصی برای برنامه‌های گرافیکی است. پردازنده‌های معمولی غالبا بیشتر از چند هسته محاسباتی ندارند، ولی پردازنده-های گرافیکی دارای هسته‌های محاسباتی متعددی هستند که امکان موازی‌سازی عملیات پردازشی را در سطح گسترده فراهم می‌کنند.. در مرجع [۱٤] مقایسه مصوری بین معماری پردازنده‌های گرافیکی و معمولی انجام شده است.

یک پردازنده گرافیکی برخلاف پردازنده معمولی حجم زیادی از محاسبات موازی گرافیکی مرتبط با تصاویر را انجام می‌دهد و بنابراین لازم است که دارای هسته‌های پردازشی بسیار بیشتری نسبت به یک پردازنده معمولی باشد. هسته‌های پردازنده معمولی دارای اندازه کوچکتری هستند. در پردازنده گرافیکی غالبا پردازش‌ها به علت سادگی عملیات، مانند جمع، ضرب، تفریق و معکوس کردن پیکسل‌های تصاویر، در یک مرحله انجام می‌شوند .پردازنده گرافیکی کاملاً به شکل اختصاصی عمل می‌نماید. حال آنکه

پردازنده معمولی تنوع گسترده‌ای از پردازش‌های مختلف، از پردازش‌های سیستمی گرفته تا برنامه‌های کاربردی، را اجرا می‌کند که باعث می‌شود عملیات محاسباتی آن پیچیده‌تر از پردازنده‌گرافیکی باشد و در نتیجه نیاز به واحد کنترل پیچیده‌تری داشته باشد. به این ترتیب، در پردازنده معمولی بخش زیادی از معماری به واحد کنترل و حافظه پنهان اختصاص داده می‌شود. ساختار حافظه در پردازنده گرافیکی سلسله مراتب بیشتری دارد زیرا در این معماری فرض بر این است که قسمت‌های برنامه یا داده‌ها می‌توانند در حین اجرا در واحدهای پردازشی مختلفی اجرا یا به کارگیری شوند. در معماری پردازنده گرافیکی هر بخش پردازنده دارای مقداری حافظه پنهان و واحد کنترل جداگانه است تا این بخش‌ها بتوانند به صورت مستقل به اجرای برنامه‌ها بپردازند. واحد اجرای برنامه‌ها که هسته نامیده می‌شود دارای حافظه جداگانه برای انجام یک پردازش است. ضمن آنکه هر چند هسته می‌توانند از نوع خاصی از حافظه مشترک برای تبادل داده‌ها استفاده کنند [۱۵].

## ۳ –روش پیشنهادی

زمان اجرای الگوریتم‌های فراابتکاری مبتنی بر جمعیت، مانند الگوریتم پروانه، در حالتی که فضای جستجو ساده و اندازه جمعیت کوچک است قابل توجه نیست، اما با افزایش پیچیدگی فضای جستجوی مساله، لازم است که اندازه جمعیت افزایش یابد تا فضای مساله برای یافتن جواب‌های بهینه به خوبی مورد جستجو قرار گیرد. افزایش اندازه جمعیت، زمان اجرای الگوریتم را به شدت افزایش می‌دهد. ماهیت مستقل راه‌حل‌ها در فضای جستجوی الگوریتم‌های فراابتکاری، امکان اجرای موازی و تسریع این الگوریتم‌ها را فراهم می‌نماید. پردازنده‌های گرافیکی به دلیل تعداد فراوان هسته‌های به کار رفته در آنها، توانایی به مراتب بیشتری نسبت به پردازنده‌های معمولی، برای موازی‌سازی و در نتیجه تسریع الگوریتم‌ها دارند. در شکل۱، نمونه‌ای از معماری موازی قابل مشاهده است که در آن فقط از یک پردازنده گرافیکی برای موازی سازی و تسریع استفاده می‌گردد.




**شکل۱. اجرای الگوریتم‌های فراابتکاری در معماری دارای یک پردازنده گرافیکی و یک پردازنده معمولی**

در معماری شکل ۱ برای پردازش و اجرای الگوریتم‌های فراابتکاری یک همکاری بین پردازنده معمولی و پردازنده گرافیکی انجام می‌شود و برای اجرای موازی الگوریتم فراابتکاری و فرآیند زیر صورت می‌پذیرد:

- تنظیم متغیرهای اولیه الگوریتم فراابتکاری مثل اندازه جمعیت اولیه، تعداد تکرار الگوریتم، و تعداد بلاک‌ها
- ایجاد جمعیت اولیه‌ای از راه‌حل‌ها در حافظه میزبان به صورت تصادفی و ارزیابی آنها توسط تابع هدف مساله
- تعریف تعدادی رشته[۹] با شماره منحصر به فرد در کودا
- تعریف حافظه‌های مورد نیاز در کودا
- انتقال جمعیت اولیه به حافظه پردازنده گرافیکی
- اجرای یک کرنل از دستورات الگوریتم فراابتکاری پروانه توسط مجموعه‌ای از رشته‌ها بر روی جمعیت اولیه
- تکرار متوالی اجرای کرنل و به روزرسانی شایسته‌ترین راه‌حل الگوریتم پروانه در هر تکرار
- انتقال شایسته‌ترین راه‌حل از حافظه سراسری پردازنده گرافیکی به حافظه اصلی پردازنده معمولی
- پاکسازی حافظه‌های به کار رفته کودا در تخصیص حافظه جمعیت اولیه
- محاسبه زمان اجرا و ارزیابی

الگوریتم ۱ نشان دهنده شبه کد این فرآیند است.

**Begin**
Initialize the parameters of the algorithm, population_size and MaxIt
Model the initial population as $Pop =$
$[Pop_1^1 Pop_2^1 \dots Pop_d^1 Pop_1^2 Pop_2^2 \dots Pop_d^2 \dots Pop_1^N Pop_2^N \dots Pop_d^N]$
Let b$est = min_{\forall 1 \le i \le N}(CostFunction(Pop_i^p))$
Create initial population randomly in the main memory of CPU
Set CUDA memory for vector Pop: *cudaMalloc((void**) & Pop, sizeof(float) * N)*
Transfer Initial Pop from main memory of CPU to global memory of GPU
*cudaMemcpy(Pop, Popdev, sizeof(float) * N, cudaMemcpyHostToDevice)*
**For** i = ۱ to MaxIt **do**
**Begin**
Create set of threads $T = \{T_1, T_2, T_3, \dots, T_N\}$
Let *index = blockIdx.x*blockDim.x+threadIdx.x*
Let $P(index) = \ll Pop(index.d), Pop(index.d+1), \dots, Pop(index.2d-1) \gg$
Calculate optimal member of the population:
**If** *Cost (*$Pop(index)$*)<*$best$ **then** $best = Cost(Pop(index))$
Update solutions using BOA algorithm Formula
**End**
Transfer best solutions of GPU to the main memory of CPU
Free all memory of CUDA
**End**

[۹] Thread

**الگوریتم ۱. شبه کد فرآیند پیشنهادی برای موازی‌سازی الگوریتم پروانه با یک پردازنده گرافیکی و یک پردازنده معمولی**

همان طور که در شبه کد الگوریتم ۱ مشاهده می‌شود، برای موازی سازی با یک پردازنده گرافیکی و یک پردازنده معمولی، در ابتدا لازم است که هر راه‌حل به صورت مناسب کدگذاری شود و سپس جمعیت این راه‌حل‌ها در یک آرایه قرار داده شود تا بتوان آن را در یک مرحله بین حافظه اصلی و سراسری جابجا نمود تا زمان اجرای الگوریتم کاهش یابد. زیرا انتقال داده بین پردازنده معمولی و گرافیکی زمان اجرا را افزایش داده و از کارایی روش پیشنهادی می‌کاهد. در روش پیشنهادی کل جمعیت درون یک آرایه قرار می‌گیرد که هر d مولفه آن برابر یک عضو جمعیت پروانه‌ها است. به عنوان نمونه مولفه ۰ تا ۱-d معرف عضو اول جمعیت و مولفه d تا ۱-۲d معرف عضو شماره دو جمعیت است. به این ترتیب با در اختیار داشتن شماره یک عضو جمعیت می‌توان مشخص نمود که چه بازه‌ای از خانه‌های آرایه متعلق به آن عضو جمعیت پروانه‌ها است. این مساله در زمان تخصیص رشته‌ها اهمیت به سزایی دارد. فرض کنید که جمعیت پروانه‌ها در یک آرایه مانند Pop و مطابق رابطه (۳)، نگهداری می‌شود:

$$\boldsymbol{Pop} = [\boldsymbol{Pop_1^1 Pop_2^1 \dots Pop_d^1 Pop_1^2 Pop_2^2 \dots Pop_d^2 \dots Pop_1^N Pop_2^N \dots Pop_d^N}] \quad .۳$$

دراین رابطه، هر d مولفه در آرایه، یک عضو از جمعیت را مشخص می‌نماید و با توجه به اینکه تعداد اعضای جمعیت برابر N است، به N رشته نیاز داریم که هر یک d مولفه از آرایه جمعیت را مدیریت نموده و این مولفه‌ها را از حافظه اصلی به حافظه سراسری منتقل نماید. برای آنکه یک رشته بتواند به d مولفه در آرایه جمعیت دسترسی داشته باشد لازم است دارای آدرس‌دهی منحصر به فرد باشد که ضابطه آدرسی‌دهی هر رشته در رابطه (٤)، نمایش داده شده است:

$$\boldsymbol{index = blockIdx.x * blockDim.x + threadIdx.x} \quad .٤$$

دراین رابطه، شماره منحصر به فرد هر رشته بر اساس شماره بلاک آن رشته، آدرس درون بلاکی و اندازه یک بلاک مشخص شده است. در این رابطه، *blockIdx.x*، *blockDim.x* و *threadIdx.x* به ترتیب شماره یک بلاک که رشته درون آن قرار دارد، اندازه یک بلاک‌هایی که رشته‌ها را تعریف می‌نمایند، و شماره درون بلاکی هر رشته در راستای افقی هستند. با استفاده از این رابطه، *index* یا شماره منحصر به فرد هر رشته محاسبه می‌گردد و بر اساس آن مشخص می‌شود که هر رشته به کدام d مولفه در آرایه جمعیت دسترسی دارد. روش این محاسبه در رابطه (۵)، نمایش داده شده است.

$$\boldsymbol{Pop(index)} = \ll \boldsymbol{Pop(index.d), Pop(index.d+1), \dots, Pop(index.d+d-1)} \gg \quad .۵$$

به عنوان مثال پروانه‌های شماره صفر و یک جمعیت که به ترتیب توسط مولفه‌های موجود در رابطه‌های (٦) و (۷) تعیین ترتیب با رشته‌های شماره صفر و یک تعیین می‌شوند به می‌شوند:

$$\boldsymbol{Pop(0)} = \ll \boldsymbol{Pop(0), Pop(1), \dots, Pop(d-1)} \gg \quad .٦$$

$$\boldsymbol{Pop(1)} = \ll \boldsymbol{Pop(d), Pop(d+1), \dots, Pop(2d-1)} \gg \quad .۷$$

در این مرحله مجموعه‌ای از رشته‌ها وجود دارد که هر رشته به بخشی از آرایه جمعیت دسترسی دارد و خانه‌های که یک رشته به آنها دسترسی دارد در واقع معرف یک پروانه است. با کدینگ مساله می‌توان اعضای جمعیت را که به صورت تصادفی در حافظه اصلی پردازنده معمولی ایجاد شده‌اند به حافظه سراسری ارسال نمود. برای این کار لازم است در ابتدا یک بردار به اندازه جمعیت اولیه در حافظه سراسری پردازنده گرافیکی رزرو شود.

هر رشته کودا در روش پیشنهادی می‌تواند بر روی یک هسته اجرا شود و این هسته موقعیت یا بردار موقعیت یک پروانه را زمان‌بندی می‌کند و توسط این رشته که با شماره ***index*** مشخص شده موقعیت پروانه با استفاده از رابطه‌های (۸) و (۹) به روزرسانی می‌گردد. در این رابطه، و ***Pop(index)*** موقعیت یک راه‌حل مساله است.

$$\boldsymbol{Pop(index) = Pop(index) + (r^2 \times g^* - x_i^t) \times f_i} \quad (8)$$

$$\boldsymbol{Pop(index) = Pop(index) + (r^2 \times Pop(index+1) - Pop(index+2)) \times f_i} \quad (9)$$

چون در پردازنده گرافیکی تعداد هسته‌ها بسیار بیشتر از یک پردازنده معمولی است و پردازنده گرافیکی یک پردازنده اختصاصی است که می‌توان از آن فقط برای پردازش یک برنامه یا یک الگوریتم خاص استفاده نمود، این نوع پردازنده‌ها برای موازی‌سازی الگوریتم پروانه مناسب هستند. از طرفی چون این پردازنده‌ها نیز محدودیت‌های خود را داشته و توانایی پردازش آنها محدود است و تعداد هسته‌های آنها در بسیاری از موارد از تعداد راه‌حل‌های مساله کمتر است، از این رو یک صف از راه‌حل‌های مساله ایجاد شده که برای رزرو هسته‌های پردازنده با هم زمان‌بندی می‌شوند. برای رفع این چالش، یک روش پیشنهادی مبتنی بر بیش از یک پردازنده گرافیکی در این مقاله ارائه می‌شود. چارچوب معماری این روش، یک بار بر اساس به کارگیری دو پردازنده گرافیکی بر روی دو سیستم مجزا در یک شبکه واحد، و بار دیگر با استفاده از چهار پردازنده گرافیکی بر روی چهار سیستم مجزا در یک شبکه ارائه شده و مورد ارزیابی قرار گرفته است. الگوریتم ۲ شبه کد الگوریتم پیشنهادی برای این روش را در حالت به کارگیری دو پردازنده گرافیکی بر روی دو سیستم مجزا نشان می‌دهد.

**Begin**
Initialize the parameters of the algorithm, population_size and MaxIt
Model the initial population as $Pop = [Pop_1^1 Pop_2^1 \dots Pop_d^1 Pop_1^2 Pop_2^2 \dots Pop_d^2 \dots Pop_1^N Pop_2^N \dots Pop_d^N]$
Let $best = min_{\forall 1 \le i \le N}(CostFunction(Pop_i^p))$
Create initial population randomly in the main memory of CPU
Set CUDA memory for vector Pop for two GPUs as:
*cudaMalloc((void**) & Pop, sizeof(float) * N/۲) in GPU۱*
*cudaMalloc((void**) & Pop, sizeof(float) * N/۲) in GPU۲*
Transfer population_size/۲ of Pop from main memory of CPU to global memory of GPU۱
Transfer the other population_size/۲ of Pop from main memory of CPU to global memory of GPU۲
*cudaMemcpy(Pop, Popdev, sizeof(float) * N, cudaMemcpyHostToDevice);*
**For** i = ۱ to MaxIt do
**Begin**
Create set of threads $T = \{T_1, T_2, T_3, \dots, T_{N/2}\}$ in GPU۱
Create set of threads $T = \{T_1, T_2, T_3, \dots, T_{N/2}\}$ in GPU۲
*Index۱ = blockIdx.x * blockDim.x + threadIdx.x*
*Index۲ = blockIdx.x * blockDim.x + threadIdx.x*

```
    Index {Pop۱, Pop۲, … , Pop_{N/۲}} in CUDA cores of GPU۱
    Index {Pop_{N/۲} + ۱, Pop_{N/۲} + ۱, … , Pop_N} in CUDA cores of GPU۲
End
Let Pop۱(index) = ≪ Pop۱(index.d), Pop۱(index.d + ۱), … , Pop۱(index.d + d − ۱ ) ≫
Let Pop۲(index) = ≪ Pop۲(index.d), Pop۲(index.d + ۱), … , Pop۲(index.d + d − ۱ ) ≫
Calculate optimal member of the {Pop۱, Pop۲, … , Pop_{N/۲}} as best۱
Calculate optimal member of the {Pop_{N/۲} + ۱, Pop_{N/۲} + ۱, … , Pop_N} as best۲
Update Solutions by BOA Algorithm Formula
If Cost (Pop۱(index))<best۱ then best۱ = Cost(Pop۱(index))
Else If Cost (Pop۲(index))<best۲ then best۲ = Cost(Pop۲(index))
Else If Cost(best ۱) < Cost(best۲) then best = best۱
Else best = best ۲
Transfer the best solutions of Device (GPU۱,GPU۲) to main memory (CPU)
Free All memories of CUDA
End
```

**الگوریتم ۲. شبه کد الگوریتم پیشنهادی برای موازی‌سازی الگوریتم پروانه با دو پردازنده گرافیکی و یک پردازنده معمولی**

بدیهی است که با افزایش تعداد سیستم‌های مستقل موجود در شبکه به عدد چهار و همچنین با تقسیم سایز جمعیت پروانه‌ها به چهار قسمت مساوی برای توزیع کمی یکسان بر روی پردازنده‌های گرافیکی، الگوریتم شماره ۲ را می‌توان با تغییراتی جزئی برای حالت ذکر شده اخیر تعمیم داد.

### ٤-پیاده سازی و تحلیل

در این بخش نتایج آزمایشهای انجام شده بر اساس پیاده‌سازی روش مقاله ارائه می‌گردد و بر روی نتایج ارائه شده تحلیل و بررسی صورت می‌گیرد.

### ٤-١- توابع ارزیابی

برای ارزیابی هر الگوریتم تکاملی دو شاخص اصلی میزان خطای یافتن بهینه سراسری و نرخ کاهش خطای یافتن بهینه سراسری بر حسب تکرار یا همگرایی مطرح است. توابع ارزیابی که در این پژوهش از آنها استفاده شده اسفیر[۱۰]، آکلی[۱۱]، گریونک[۱۲]، و رستریجین[۱۳] می‌باشند. تابع ارزیابی اسفیر یکی از توابع ارزیابی پرکاربرد در بحث سنجش دقت و همگرایی الگوریتم‌های فراابتکاری و تکاملی است که فقط یک کمینه سراسری به عنوان بهینه سراسری در نقطه $x = y = ۰$ و مقدار $f^*(۰,۰) = ۰$ دارد و چون بهینه محلی ندارد فضای جستجوی نسبتاً ساده‌ای دارد. گریونک یک تابع ارزیابی پیچیده مانند تابع ارزیابی آکلی است که دارای یک کمینه سراسری در نقطه $x = y = ۰$ با مقدار بهینه $f^*(۰,۰) = ۰$ و بی‌شمار کمینه محلی در فضای جستجوی خود است. در این تابع

[۱۰] Sphere
[۱۱] Ackley
[۱۲] Griewank
[۱۳] Rastrigin

ارزیابی کمینه‌های محلی به صورت متقارن پیرامون بهینه سراسری قرار گرفته و آن را احاطه نموده‌اند. تابع ارزیابی رستریجین مانند تابع‌های ارزیابی گریونک و آکلی دارای بهینه‌های محلی در کنار بهینه سراسری می‌باشد. وجه اشتراک سه تابع ارزیابی رستریجین، گریونک و آکلی فضای جستجوی پیچیده‌تر نسبت به توابع ارزیابی گریونک و جمع مربعات[۱۴] است [۱۶].

## ٤-۲- بستر موازی‌سازی

فناوری کودا یک چارچوب نرم‌افزاری در بستر سخت‌افزار است که امکان تسریع الگوریتم‌ها در قالب موازی‌سازی را فراهم می‌کند. در اینجا برای پیاده‌سازی الگوریتم پیشنهادی که یک نسخه موازی شده در کودا است از پردازنده گرافیکی NVidia GeForce GTX ۱۰٦۰ استفاده می‌گردد.

## ٤-۳- شاخص ارزیابی

نسبت زمان اجرا در پردازنده معمولی به زمان اجرا در پردازنده گرافیکی، که به ترتیب معادل زمان اجرای سریال و زمان اجرای موازی الگوریتم هستند، به عنوان شاخص شتاب مورد توجه قرار می‌گیرد که ضابطه آن در رابطه (۱۰) ارائه شده است:

$$\boldsymbol{Speedup} = \frac{\boldsymbol{Runime\ CPU(Serial)}}{\boldsymbol{Runime\ GPU(Parallel)}} \quad .۱۰$$

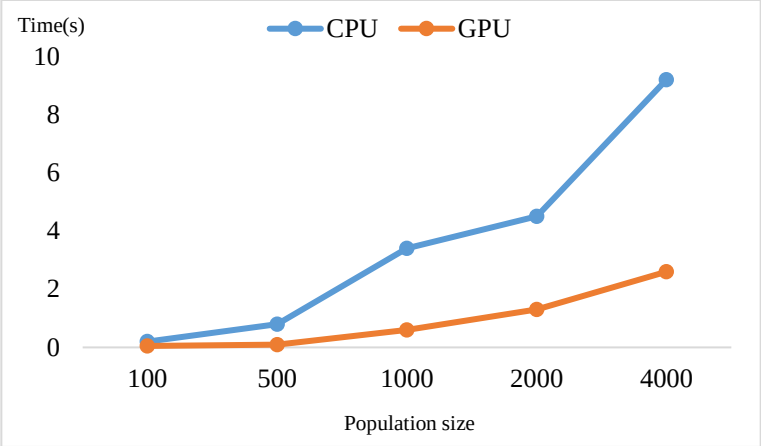
## ٤-٤- تحلیل خروجی‌ها

در این بخش در ابتدا برای موازی‌سازی الگوریتم بهینه‌سازی پروانه از یک پردازنده گرافیکی GeForce GTX ۱۰٦۰ استفاده شده است که با یک پردازنده معمولی اینتل ۵ هسته‌ای با حافظه اصلی ۸ گیگابایت مرتبط شده است. نتایج اجرای الگوریتم بر روی این معماری به ازای اندازه‌های مختلف جمعیت به دست آمده و مورد مقایسه قرار گرفته است. در ادامه نتایج آزمایش‌های بر روی معماری پیشنهادی با دو و چهار پردازنده گرافیکی نیز به دست آمده و تاثیر وجود دو پردازنده گرافیکی برای اجرای الگوریتم مورد بررسی و بحث قرار گرفته است. تعداد دفعات تکرار برابر ۲۰۰ در نظر گرفته شده است. روش پیشنهادی ما شامل دو بخش ذیل در پیاده‌سازی‌ها و تحلیل‌ها است:

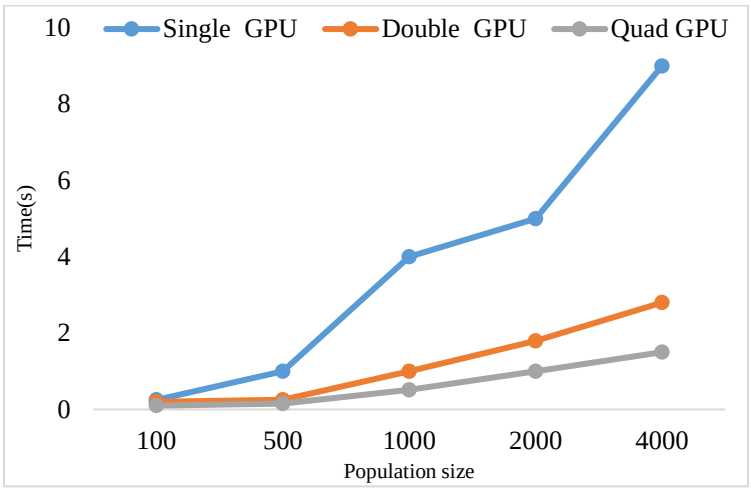
- اجرای الگوریتم پروانه بر روی یک پردازنده گرافیکی GeForce GTX ۱۰٦۰ و یک پردازنده معمولی Intel
- اجرای الگوریتم پروانه بر روی دو و چهار پردازنده گرافیکی GeForce GTX ۱۰٦۰ و یک پردازنده معمولی Intel

شکل‌های ۲ و ۳ به ترتیب متوسط زمان اجرای روش پیشنهادی روی توابع ارزیابی با معماری یک، دو و چهار پردازنده گرافیکی به ازای اندازه‌های جمعیت ۱۰۰، ۵۰۰، ۱۰۰۰، ۲۰۰۰ و ٤۰۰۰، با تکرار ۲۰۰ را نشان می‌دهند. با توجه به این نمودارها می‌توان نتیجه گرفت که در معماری پیشنهادی زمان اجرا کاهش یافته و این کاهش به دلیل تقسیم و غلبه‌ای است که بر روی جمعیت پروانه‌ها انجام شده است.

[۱۴] Sum square

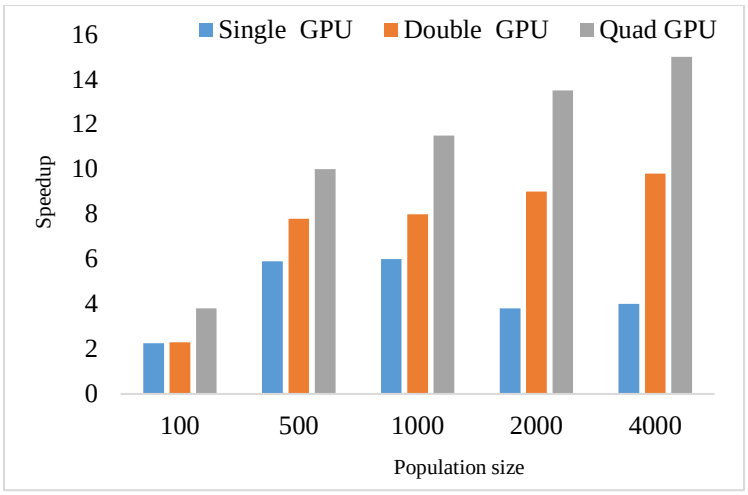



**شکل۲. زمان اجرای الگوریتم پروانه بر روی یک پردازنده گرافیکی**




**شکل۳. زمان اجرای الگوریتم موازی شده پروانه بر روی یک، دو و چهار پردازنده گرافیکی**

شکل ٤ نتایج آزمایش‌های صورت گرفته برای ارزیابی شتاب اجرایی روش پیشنهادی را نشان می‌دهد. با توجه به نمودارهای این شکل می‌توان مشاهده کرد که با معماری پیشنهادی با دو پردازنده گرافیکی، شتاب اجرایی بیشتری نسبت به الگوریتم موازی شده پروانه با یک پردازنده گرافیکی دارد و این شتاب متناسب با اندازه جمعیت است. در جمعیت‌های با اندازه کوچک‌تر شتاب روش پیشنهادی با دو پردازنده گرافیکی حتی می‌تواند کمتر از معماری با یک پردازنده گرافیکی باشد که دلیل آن این است که تقسیم جمعیت به دو دسته برای جمعیت‌های کوچک سربار زمانی ایجاد می‌نماید که وقتی جمعیت بزرگ در نظر گرفته شود، اندازه جمعیت تقسیم شده و این زمان در مقابل زمان موازی‌سازی کاهش خواهد یافت.




**شکل٤. شتاب اجرای الگوریتم موازی شده پروانه بر روی یک، دو و چهار پردازنده گرافیکی**

با توجه به آزمایش‌های انجام شده می‌توان نتیجه گرفت:

- افزایش جمعیت الگوریتم پروانه باعث می‌شود زمان اجرا در معماری یک پردازنده گرافیکی و بیش از یک پردازنده گرافیکی افزایش یابد اما این افزایش زمان اجرا در معماری با بیش از یک پردازنده گرافیکی از معماری یک پردازنده گرافیکی کمتر است.
- شتاب روش پیشنهادی با بیش از یک پردازنده گرافیکی با افزایش جمعیت دائما افزایش می‌یابد اما با یک پردازنده گرافیکی، این شتاب با افزایش اندازه جمعیت در ابتدا افزایشی بوده و بعد از آن روند کاهشی دارد.

## ٥-نتیجه گیری

برای افزایش دقت در الگوریتم‌های فراابتکاری لازم است اندازه جمعیت افزایش داده شود تا خطای محاسبات کاهش یابد که نمونه آن را می‌توان در الگوریتم بهینه‌سازی پروانه مشاهده نمود. چالش مهم افزایش اندازه جمعیت در الگوریتم‌های فراابتکاری کاهش سرعت/ همگرایی و افزایش زمان اجرای الگوریتم است. در این مقاله یک روش جدید برای تسریع الگوریتم فراابتکاری پروانه با معماری پردازنده گرافیکی ارایه شد و از یک معماری مبتنی بر دو و چهار پردازنده گرافیکی برای شتاب دادن استفاده شد. نتایج آزمایش‌های تجربی نشان داد که معماری پیشنهادی موجب بهبود سرعت الگوریتم بهینه‌سازی پروانه شده است. در پژوهش‌های آتی به دنبال آن هستیم که از معماری مبتنی بر بیش از یک پردازنده گرافیکی برای حل سایر مسائل مرتبط با حوزه هوش مصنوعی و بهینه‌سازی استفاده کنیم. موازی‌سازی سایر الگوریتم‌های بهینه‌سازی مانند الگوریتم‌های گرگ خاکستری و کفتار و موازی‌سازی مساله استخراج قوانین تداعی در داده‌کاوی با معماری دو (یا چند) پردازنده گرافیکی می‌توانند نمونه‌هایی از این پژوهش‌های آتی باشند.

## مراجع:


١. Cook, C., Zhao, H., Sato, T., Hiromoto, M., & Tan, S. X. D. (٢٠١٩). GPU-based Ising computing for solving max-cut combinatorial optimization problems. *Integration*.

٢. Djenouri, Y., Djenouri, D., Belhadi, A., Fournier-Viger, P., Lin, J. C. W., & Bendjoudi, A. (٢٠١٩). Exploiting GPU parallelism in improving bees swarm optimization for mining big transactional databases. *Information Sciences*, *٤٩٦*, ٣٢٦-٣٤٢.

٣. Ho, D., Liang, E., Stoica, I., Abbeel, P., & Chen, X. (٢٠١٩). Population Based Augmentation: Efficient Learning of Augmentation Policy Schedules. *arXiv preprint arXiv: ١٩٠٥,٠٥٣٩٣*.

٤. Rios, E., Ochi, L. S., Boeres, C., Coelho, V. N., Coelho, I. M., & Farias, R. (٢٠١٨). Exploring parallel multi-GPU local search strategies in a metaheuristic framework. *Journal of Parallel and Distributed Computing*, *١١١*, ٣٩-٥٥.

٥. Mirjalili, S., Mirjalili, S. M., & Lewis, A. (٢٠١٤). Grey wolf optimizer. *Advances in engineering software*, *٦٩*, ٤٦-٦١.

٦. Mirjalili, S. (٢٠١٥). The ant lion optimizer. *Advances in engineering software*, *٨٣*, ٨٠-٩٨.

٧. Mirjalili, S., & Lewis, A. (٢٠١٦). The whale optimization algorithm. *Advances in engineering software*, *٩٥*, ٥١-٦٧.

٨. Das, S., Biswas, A., Dasgupta, S., & Abraham, A. (٢٠٠٩). Bacterial foraging optimization algorithm: theoretical foundations, analysis, and applications. In: Abraham A., Hassanien AE., Siarry P., Engelbrecht A. (eds) *Foundations of Computational Intelligence* Volume ٣. *Studies in Computational Intelligence*, *vol ٢٠٣*. Springer, Berlin, Heidelberg.

٩. Cuevas, E., & Cienfuegos, M. (٢٠١٤). A new algorithm inspired in the behavior of the social-spider for constrained optimization. *Expert Systems with Applications*, *٤١(٢)*, ٤١٢-٤٢٥.

١٠. Dhiman, G., & Kumar, V. (٢٠١٧). Spotted hyena optimizer: a novel bio-inspired based metaheuristic technique for engineering applications. *Advances in Engineering Software*, *١١٤*, ٤٨-٧٠.

١١. Arora, S., & Singh, S. (٢٠١٩). Butterfly optimization algorithm: a novel approach for global optimization. *Soft Computing*, *٢٣*(٣), ٧١٥-٧٣٤.

١٢. Cano, A., & Krawczyk, B. (٢٠١٨, July). Learning Classification Rules with Differential Evolution for High-Speed Data Stream Mining on GPU s. In *٢٠١٨ IEEE Congress on Evolutionary Computation (CEC)* (pp. ١-٨). IEEE.

١٣. Sivakumar, S., Vidyanandini, S., Nayak, S. R., & Sundar, S. (٢٠١٩). Parallel Computation of a MMDBM Algorithm on GPU Mining with Big Data. In *Cloud Computing for Geospatial Big Data Analytics* (pp. ١٣٧-١٥٣). Springer, Cham.

١٤. Cheng, J., Grossman, M., & McKercher, T. (٢٠١٤). Professional Cuda C Programming. *John Wiley & Sons*.

١٥. CUDA, C. (٢٠١٣). Programming Guide v٥. ٥. *NVIDIA Corporation, July*.

١٦. Cheng, R., Li, M., Tian, Y., Xiang, X., Zhang, X., Yang, S., ...& Yao, X. (٢٠١٨). *Benchmark functions for the cec'٢٠١٨ competition on many-objective optimization*.