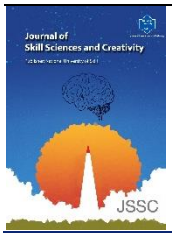


Journal of Skill Sciences and Creativity
National University of Skill
Summer 2025, Vol. 2, No. 1, p. 69-91
Journal Homepage: https://jssc.tvu.ac.ir/?lang=en
:10.48301/JSSC2025.548051.1040

Parallelization of the Flying Fox Optimization Algorithm Using CUDA Architecture in MATLAB

Farnaz Hoseini[1*], Masume Kheyri[2]

[1]Department of Computer Engineering, Technical and Vocational University (TVU), Tehran, Iran.
[2]Department Computer Engineering, Technical and Vocational University (TVU), Tehran, Iran.

ARTICLE INFO

Article Type:
Original Research

Received: 16.01.2025
Revised: 16.02.2025
Accepted: 11.12.2025

Keyword:
Flying Fox Optimization Algorithm
Parallel Computing
CUDA Architecture
GPU Processing
Metaheuristic Algorithms
MATLAB

***Corresponding Author:**
Farnaz Hoseini
Email: *f-hoseini@tvu.ac.ir*

ABSTRACT

The Flying Fox Optimization Algorithm is a metaheuristic method inspired by the foraging and movement behavior of flying foxes. These creatures, a type of bat, provide a suitable model for designing an efficient algorithm through their flight between trees and search for food resources. In this study, the main objective is to develop and evaluate the performance of this algorithm in the MATLAB environment while leveraging CUDA architecture to enable parallel processing and GPU computational power. For this purpose, a proposed version of the algorithm was implemented, and its execution speed and efficiency were compared with the serial version. Simulation results indicate that parallel execution of the algorithm on the GPU significantly reduces execution time, performing over 314 times faster than the serial implementation. This remarkable improvement is primarily due to the simultaneous execution of blocks and optimal utilization of GPU computational resources. The findings suggest that the parallelized version of this algorithm demonstrates higher efficiency compared to traditional methods and can be highly effective in solving complex scientific and engineering problems, including numerical simulations, big data processing, and real-time modeling. Therefore, using CUDA architecture can play a crucial role in enhancing both the speed and quality of metaheuristic computations.

EXTENDED ABSTRACT

Introduction

In recent years, a wide range of nature-inspired algorithms have been developed to solve optimization problems. The flying fox optimization algorithm or FFOA is inspired by the nature and behaviour of flying foxes, including fast flight and jumping from different locations, and has been effectively used in modelling. This algorithm can be adapted for various problems (such as continuous and discrete optimization problems). Also, this algorithm has the ability to find the best solution in complex problems by combining global and local search. This study was conducted in two stages with the aim of parallelizing the flying fox optimization algorithm using the CUDA architecture in the MATLAB environment. In the first stage, the initial formulation of the flying fox optimization algorithm was presented on the MATLAB software and the execution time of this algorithm in serial mode for different numbers of foxes was investigated. In the second stage, a parallel solution was presented to reduce the execution time of this algorithm. In the first stage, the initial formulation of the flying fox optimization algorithm was presented on the MATLAB software and the execution time of this algorithm in serial mode was examined for different numbers of foxes. In the second stage, a parallel solution was presented to reduce the execution time of this algorithm. Since the MATLAB kernel supports multithreading internally, it enables simultaneous execution of optimized calculations and better utilization of processing resources. In this study, the main objective is to develop and optimize an efficient computational algorithm and evaluate its performance in the CUDA architecture. The research focus is on improving the processing speed and optimal utilization of GPU computational resources. For this purpose, the proposed algorithm is designed and implemented and then its performance is analysed in terms of execution time and GPU efficiency.

Methodology

The Flying Fox Optimization Algorithm (FFOA) is a nature-based optimization algorithm inspired by flying foxes' foraging and group migration behavior. It is used to solve complex, multidimensional optimization problems. In this algorithm, a population of "flying foxes" is created, each representing a point in the search space. Flying foxes are a type of bat that fly from tree to tree searching for food sources. The algorithm generally consists of the following steps:

Initialization and initialization: A population of flying foxes is generated, each representing a possible solution to the problem. Initial values such as the number of foxes, the maximum number of iterations, and the range of variables are specified.

Objective function determination: An objective function value is calculated for each fox based on its position in the search space. The algorithm's goal is to find a position with the minimum or maximum value of the objective function.

Foxes' movement: Foxes change their positions using a movement model that depends on a combination of direct flight and local search. Movement is performed toward the best-identified positions (based on the objective function).

Position improvement: If the new position is better than the previous position, an update is performed. Behaviors such as exploration and extraction are used to ensure that the entire problem space is searched and that the optimal regions are focused.

Stopping criterion is reached: The algorithm continues until a certain number of iterations are reached or there is no significant change in the value of the objective function.

To mathematically simulate the FFOA algorithm, we must first model the key behaviors of this algorithm. These behaviors include flying foxes' movement, exploration, and extraction.

To implement the algorithm in parallel in MATLAB using the CUDA (Compute Unified Device Architecture) architecture and graphics processing units (GPU), we will use the capabilities of MATLAB's Parallel Computing Toolbox. This tool allows you to transfer calculations to the GPU, which can improve the performance of heavy algorithms. MATLAB software is considered one of the high-level programming software focused on computational techniques, due to its simple commands and functions and vector data environment. CUDA uses the concept of single-cycle switches to hide the latency of data paths and memory access. CUDA's execution management method is to divide groups of threads among blocks. Grids are composed of a collection of blocks. Threads can specify their location in a block and the location of a block within the grid with intrinsic data elements initialized by CUDA. Threads from different blocks belonging to a grid can be synchronized through atomic operations in a global memory space shared with other threads. NVIDIA GPU hardware (shown in Figure 1) is built using SPMD multiprocessing units.

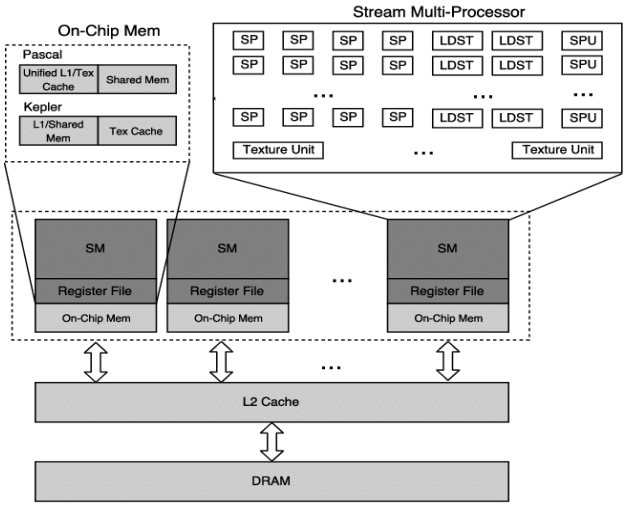


Figure 1: GPU from a hardware implementation perspective

Results and discussion

Speedup is obtained by dividing the serial execution time (execution on the CPU) and the parallel execution time (execution on the GPU). If the Speedup value is 2, it means that

parallel processing is twice as fast as serial processing, and if the Speedup is 1, there is no improvement. A Speedup greater than one indicates an improvement in the performance of the implementation method. The graph in Figure 2 shows the comparison of the execution time of the FFOA algorithm on GPU hardware versus CPU hardware. In this graph, the horizontal axis is the number of foxes and the vertical axis is the Speedup value. With the Speedup shown in the graph in Figure 10, it can be concluded that with the increase in the fox population, the execution speed of the algorithm on GPU hardware has increased by about 314.074 times.

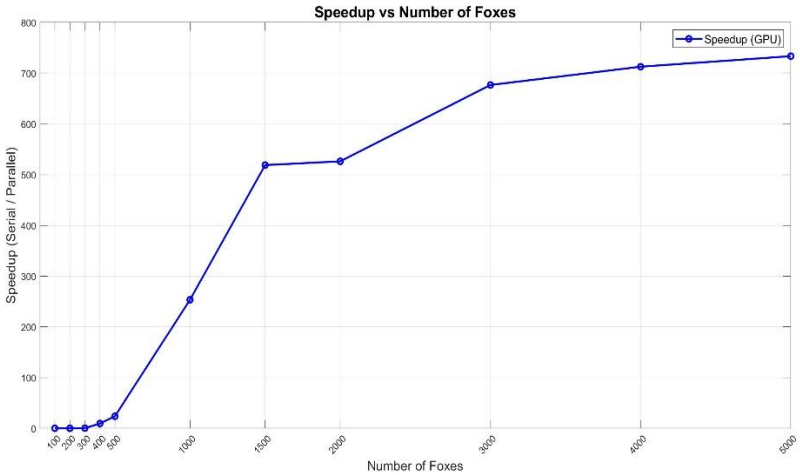


Figure 2: Speedup obtained from running the FFOA algorithm in both serial (on CPU) and parallel (on GPU) modes for different fox populations

Conclusion

In this paper, a new approach to implementing the flying fox optimization algorithm in parallel on the MATLAB environment was presented. As the results showed, in serial mode, with the increase in the fox population, the CPU calculation execution time also increases, while the execution time of this algorithm in parallel mode has been greatly reduced by using the CUDA architecture features and the parallel processing power provided by the GPU. Also, comparing the execution time of the FFOA algorithm in serial and parallel modes showed that implementing this algorithm on GPU hardware using the simultaneous execution capability of blocks can increase the execution speed by about 314.074 times compared to CPU hardware. Due to the user-friendliness of GPU hardware, the architecture of this hardware is constantly growing and developing, and in the near future, by using this hardware, we can witness significant progress in reducing the execution time of heavy calculations on parallel environments.

نشریه علوم مهارتی و خلاقیت
دانشگاه ملی مهارت
تابستان۱٤۰٤، دوره۲، شماره ۱، ٦۹-۲۱
آدرس نشریه: https://jssc.tvu.ac.ir/
doi:10.48301/JSSC2025.548051.1040

موازی سازی الگوریتم بهینه سازی روباه پرنده با استفاده از معماری CUDA در MATLAB

فرناز حسینی[1*]، معصومه خیری[2]

۱- گروه مهندسی کامپیوتر، دانشگاه ملی مهارت، تهران، ایران.
۲- گروه مهندسی کامپیوتر، دانشگاه ملی مهارت، تهران، ایران.

اطلاعات مقاله

نوع مقاله: مقاله پژوهشی

دریافت مقاله: ۱۴۰۳/۱۰/۲۷
بازنگری مقاله: ۱۴۰۳/۱۱/۲۸
پذیرش مقاله: ۱۴۰۴/۰۹/۲۰

کلید واژگان:
الگوریتم بهینه‌سازی روباه پرنده
محاسبات موازی
معماری CUDA
پردازنده گرافیکی (GPU)
الگوریتم‌های فراابتکاری
MATLAB

***نویسنده مسئول:** فرناز حسینی
پست الکترونیکی:
f-hoseini@tvu.ac.ir

چکیده

الگوریتم بهینه‌سازی روباه پرنده یک روش فراابتکاری الهام‌گرفته از رفتار جست‌وجو و حرکت روباه‌های پرنده است. این موجودات که نوعی خفاش هستند، با پرواز میان درختان و جست‌وجوی منابع غذایی، الگوی مناسبی برای طراحی یک الگوریتم کارآمد ارائه می‌دهند. در این پژوهش هدف اصلی، توسعه و ارزیابی عملکرد این الگوریتم در محیط MATLAB و با بهره‌گیری از معماری CUDA است تا امکان استفاده از پردازش موازی و توان محاسباتی GPU فراهم شود. برای این منظور، نسخه پیشنهادی الگوریتم پیاده‌سازی شده و سرعت اجرا و بهره‌وری آن با نسخه سریال مقایسه شده است.نتایج شبیه‌سازی نشان می‌دهد که اجرای موازی الگوریتم روی GPU با افزایش جمعیت روباه‌ها زمان اجرا را به‌طور قابل‌توجهی کاهش می‌دهد و نسبت به اجرای سریال بیش از ۳۱۴ برابر سریع‌تر عمل می‌کند. این بهبود چشمگیر عمدتاً به دلیل اجرای هم‌زمان بلاک‌ها و استفاده بهینه از توان محاسباتی GPU است. یافته‌ها بیانگر آن است که نسخه موازی‌شده این الگوریتم در مقایسه با روش‌های سنتی کارایی بالاتری دارد و می‌تواند در حل مسائل پیچیده علمی و مهندسی، از جمله شبیه‌سازی‌های عددی، پردازش داده‌های حجیم و مدل‌سازی‌های بلادرنگ، بسیار مؤثر باشد. بنابراین، استفاده از معماری CUDA می‌تواند نقش مهمی در افزایش سرعت و کیفیت محاسبات فراابتکاری ایفا کند.

مقدمه

در سال‌های اخیر طیف گسترده ای از الگوریتم‌های الهام گرفته از طبیعت برای حل مشکلات بهینه‌سازی توسعه یافته است[۱]. الگوریتم بهینه سازی روباه پرنده [۱] یا FFOA از طبیعت و رفتار روباه‌های پرنده شامل پرواز سریع و پرش از مکان‌های مختلف الهام گرفته شده و به طور مؤثر در مدل سازی استفاده شده است[۲]. این الگوریتم می‌تواند برای مسائل مختلف (مانند مسائل بهینه سازی پیوسته و گسسته) تطبیق داده شود[۳]. همچنین این الگوریتم با ترکیب جستجوی سراسری و محلی، توانایی یافتن بهترین جواب در مسائل پیچیده را دارد[۴]. از کاربردهای الگوریتم FFOA می‌توان به طراحی سیستم‌های انرژی مهندسی برق، مسائل مسیریابی، برنامه‌ریزی و تخصیص منابع و بهینه‌سازی در یادگیری ماشین و پردازش موازی اشاره کرد [۵]. عملکرد الگوریتم FFOA تا حدودی مشابه به الگوریتم های دیگری مانند الگوریتم ازدحام ذرات (PSO[۲]) و الگوریتم خفاش (Bat Algorithm) است، اما با مدل‌سازی دقیق‌تر، روباه پرنده تلاش می‌کند کارایی بهتری در برخی از مسائل خاص داشته باشد[۶]. امروزه بنا به دو دلیل اصلیِ زمان و حافظه، نیاز به عملیات موازی در الگوریتم‌ها و کامپیوترها بیش از پیش احساس می‌شود. برنامه‌هایی که نیاز به حجم بالایی از حافظه دارند اغلب به زمان طولانی برای اجرا نیاز دارند، در حالی که نرم افزار MATLAB با پشتیبانی از قابلیت‌های موازی توانسته محیط مناسبی را برای عملیات موازی در زمان کم‌تر فراهم کند[۷]. فناوری CUDA[۳] یک معماری بر پایه پردازش موازی است که در سال ۲۰۰۷ برای اولین بار توسط شرکت NVIDIA ابداع شد[۸] CUDA یک موتور قدرتمند محاسباتی برای پردازش GPU کارت گرافیک‌های NVIDIA می‌باشد. در واقع CUDA یک پلتفرم نرم‌افزاری است که با استفاده از آن صدها تراشه گرافیکی مجزای NVIDIA می‌توانند در کنار هم به پردازش موازی حجم بالایی از اطلاعات بپردازند [۹]. تکنولوژی CUDA به وسیله نرم افزارهای زیادی از جمله MATLAB پشتیبانی می‌شود که می‌تواند سرعت سیستم را برای اجرای برنامه‌های سنگین افزایش دهد (Reis et al., 2020). CUDA امکان دسترسی به دستورات مجازی و واحد حافظه‌ی محاسبات موازی در CUDA GPU را نیز فراهم می‌کند [۱۰]. ما پیش‌تر از از قابلیتِ GPU در MATLAB برای اجرای کارآمد تشخیص لبه فازی استفاده کردیم [۱۱]. همچنین برای حل مسالهی کوله‌پشتی ۰ و ۱[۴] یک راه‌کار موازی ارائه دادیم و حل این مسالهی NP Hard را یکبار توسط الگوریتم کرم شب‌تاب [۱۲] و یکبار الگوریتم جهش قورباغه (Hoseini, 2023) در بستر موازی توسط معماری CUDA انجام دادیم. امروزه از قابلیت‌های موازی سازی در بسیاری از حوزه‌ها، خصوصا در یادگیری عمیق[۱۳]، الگوریتم‌های بهینه سازی [۱۴] و مباحث پزشکی و معماری [۱۵] و ... استفاده می شود. سؤال اصلی این تحقیق این است که چگونه می‌توان الگوریتم بهینه سازی روباه پرنده را به‌گونه‌ای طراحی و پیاده سازی کرد که در معماری CUDA بهینه عمل کرده و کارایی محاسباتی آن نسبت به روش‌های سنتی بهبود یابد. این موضوع به دلیل نیاز روزافزون به بهینه‌سازی الگوریتم‌های محاسباتی در مواجهه با مسائل علمی و مهندسی پیچیده، اهمیت دارد. استفاده از توان پردازشی GPU و قابلیت اجرای موازی در معماری CUDA می‌تواند

[1] Flying Fox Optimization Algorithm
[2] Particle Swarm Optimization
[3] Compute Unified Device Architecture
[4] The 0–1 Knapsack Problem

زمان اجرای الگوریتم‌های فراابتکاری را کاهش داده و بهره‌وری آن‌ها را افزایش دهد. علاوه بر این، ارتقای عملکرد این الگوریتم‌ها می‌تواند در حوزه‌هایی مانند شبیه‌سازی‌های عددی، تحلیل داده‌های حجیم و مدل‌سازی‌های بلادرنگ تأثیر قابل‌توجهی داشته باشد. ازاین‌رو، در این پژوهش بر توسعه و ارزیابی الگوریتم روباه پرنده در بستر CUDA تمرکز شده است تا میزان کارایی آن در مقایسه با روش‌های سنتی مورد بررسی قرار گیرد. در این راستا مطالعه‌ای با هدف موازی سازی الگوریتم بهینه‌سازی روباه پرنده با استفاده از معماری CUDA در محیط MATLAB در دو مرحله انجام گرفت. در مرحله اول تدوین اولیه الگوریتم بهینه‌سازی روباه پرنده بر روی نرم افزار MATLAB ارائه گردید و زمان اجرای این الگوریتم در حالت سریال به ازای تعداد روباه‌های مختلف بررسی و در مرحله دوم به منظور کاهش زمان اجرای این الگوریتم یک راه کار موازی ارائه شد. مراحل کلی الگوریتم بهینه‌سازی روباه پرنده و فرآیند موازی‌سازی آن در MATLAB با استفاده از CUDA شامل تعیین پارامترهای اولیه، ایجاد جمعیت اولیه، محاسبه ارزش روباه‌های پرنده، به‌روزرسانی موقعیت آن‌ها بر اساس رفتار شکار، و بررسی شرایط توقف است. در صورت عدم همگرایی، الگوریتم به‌صورت موازی بر روی GPU اجرا شده و این فرآیند برای تعداد مشخصی از نسل‌ها تکرار می‌شود تا بهترین جواب ممکن به دست آید.

پیشینه تحقیق

در ادامه، مقالاتی جدید مرتبط با بهینه‌سازی الگوریتم‌های محاسباتی در معماری CUDA ارائه شده است.

کاوه و مسگری [۱۶] با عنوان بهینه‌سازی بهبود یافته‌ی الگوریتم کرم شب‌تاب برای دقت بالاتر در شبکه‌های عصبی پس‌انتشار Levenberg–Marquardt با شتاب‌دهی CUDA به بهینه‌سازی شبکه‌های عصبی پس‌انتشار Levenberg–Marquardt با استفاده از الگوریتم بهبود یافته‌ی کرم شب‌تاب پرداخت شده است. نتایج نشان می‌دهد که این مدل با شتاب‌دهی CUDA، دقت پیش‌بینی زلزله را تا ۲۰٪ افزایش می‌دهد.

سونگ [۱۷] سال ۲۰۲۴ با عنوان بهینه‌سازی و شتاب‌دهی الگوریتم‌های پردازش تصویر بر اساس معماری موازی CUDA به بهینه‌سازی و شتاب‌دهی الگوریتم‌های پردازش تصویر با استفاده از معماری موازی CUDA پرداخته شده است. نتایج نشان می‌دهد که با طراحی هوشمندانه‌ی الگوریتم‌های موازی و بهینه‌سازی دسترسی به حافظه، می‌توان کارایی پردازش تصویر را به‌طور قابل‌توجهی افزایش داد.

کوریشتی [۱۸] سال ۲۰۲۴ با عنوان تحقیق در مورد بهینه‌سازی CUDA C++ به بررسی بهینه‌سازی‌های مختلف در برنامه‌نویسی CUDA C++ پرداخته شده است. نتایج نشان می‌دهد که با استفاده از تحلیل ایستا و راهنمایی‌های خودکار، می‌توان کارایی هسته‌های CUDA را بهبود بخشید.

هونگ و همکاران [۱۹] سال ۲۰۲۴ با عنوان شتاب‌دهی الگوریتم‌های کوانتومی برای پیش‌بینی انرژی خورشیدی با استفاده از CUDA به شتاب‌دهی الگوریتم‌های کوانتومی برای پیش‌بینی انرژی خورشیدی با استفاده از CUDA پرداخته شده است. نتایج نشان می‌دهد که با ترکیب منابع کوانتومی و کلاسیک، می‌توان دقت پیش‌بینی را افزایش داد.

ده رانگو و همکاران [۲۰] سال ۲۰۲۳ با عنوان تحلیل عملکرد و بهینه سازی پیاده سازی CUDA از سلول‌های خودکار سه‌بعدی XCA-Flow به تحلیل عملکرد و بهینه‌سازی پیاده‌سازی CUDA از سلول‌های خودکار

سه‌بعدی XCA-Flow پرداخته شده است. نتایج نشان می‌دهد که با اعمال بهینه سازی‌های مختلف، می‌توان کارایی این پیاده‌سازی را بهبود بخشید.

ژو و همکاران [۲۱] سال ۲۰۲۳ با عنوان روشی برای بهبود الگوریتم بهینه‌سازی اجتماع ذرات با استفاده از معماری CUDA به بهبود الگوریتم PSO با بهره‌گیری از پردازنده‌های گرافیکی و معماری CUDA پرداخته شده است. نتایج نشان‌دهنده‌ی افزایش سرعت محاسبات تا ۹۸,۱۵ برابر در حل تابع Rastrigin است.

نارایانا و همکاران [۲۲] سال ۲۰۲۲ با عنوان پیاده‌سازی الگوریتم بهینه‌سازی مورچه گربه‌ای با استفاده از CUDA به پیاده سازی الگوریتم بهینه سازی مورچه‌گربه‌ای با استفاده از CUDA پرداخته شده است. نتایج نشان می‌دهد که با مدیریت مؤثر هسته‌ها، حافظه و نخ‌ها، می‌توان کارایی این الگوریتم را بهبود بخشید.

حجاریان و حسینی [۱۱] سال ۲۰۱۶ با عنوان موازی‌سازی الگوریتم کرم شب‌تاب با استفاده از معماری CUDA در محیط MATLAB به پیاده سازی الگوریتم بهینه سازی کرم شب‌تاب در محیط MATLAB با بهره‌گیری از معماری CUDA پرداخته شده است. نتایج نشان می‌دهد که اجرای موازی این الگوریتم بر روی GPU می‌تواند زمان اجرا را تا ۶۵۰ برابر نسبت به حالت سریال کاهش دهد.

سینگ و همکاران [۲۳] سال ۲۰۱۴ با عنوان الگوریتم بهینه‌سازی ازدحام ذرات موازی مبتنی بر GPU/CUDA به ارائه‌ی یک الگوریتم بهینه سازی ازدحام ذرات موازی با استفاده از GPU/CUDA پرداخته شده است. نتایج نشان می‌دهد که این الگوریتم می‌تواند سرعت بهینه‌سازی را به‌طور قابل‌توجهی افزایش دهد.

جوهار و همکاران [۲۴] سال ۲۰۱۳با عنوان بررسی الگوریتم‌های ژنتیک و تحلیل انواع پیاده‌سازی آن بر روی واحد پردازش گرافیکی به تحلیل و بررسی پیاده‌سازی‌های مختلف الگوریتم‌های ژنتیک بر روی GPU با استفاده از معماری CUDA پرداخته شده است. نتایج نشان می‌دهد که استفاده از CUDA می‌تواند به بهبود کارایی و کاهش زمان اجرای این الگوریتم‌ها منجر شود.

اویسو و همکاران [۲۵] سال ۲۰۱۱ با عنوان تسریع الگوریتم ژنتیک حالت دائمی بر اساس معماری CUDA پیاده‌سازی الگوریتم ژنتیک حالت دائمی در محیط CUDA مورد بررسی قرار گرفته است. نتایج حاکی از آن است که این پیاده‌سازی می‌تواند سرعت اجرا را تا ۳ تا ۶ برابر نسبت به پیاده‌سازی مشابه بر روی CPU افزایش دهد.

علاوه بر مطالعات فوق در سال‌های اخیر، طیف وسیعی از الگوریتم‌های فرابتکاری نوین با الهام از پدیده‌های طبیعی و ریاضی تو سعه یافته‌اند که هدف آن‌ها ارتقاء کارایی در حل مسائل بهینه سازی پیچیده بوده است. از جمله این روش‌ها می‌توان به الگوریتم جستجوی بازی‌های گرسنگی (HGS[1]) (Yang et al., 2021) که با بهره‌گیری از مدل‌های رفتاری بقا در شرایط بحرانی، تعادل مؤثری بین اکتشاف و بهره‌برداری ایجاد می‌کند. الگوریتم بهینه سازی ریاضی (AOA[2]) (Abualigah et al., 2021) با تکیه بر عملیات پایه‌ای ریاضی به ساختاری

[1] Hunger Games Search
[2] Arithmetic Optimization Algorithm

تطبیقی در مسیر جستجو دست یافته و الگوریتم کپک مخاطی (SMA[1]) (Li et al., 2020) با مدل‌سازی دینامیک جستجوی غذای این ارگانیسم، رویکردی پایدار در برابر گیر افتادن در بهینه‌های محلی ارائه می‌دهد. همچنین، الگوریتم شکارچیان دریایی (MPA[2])(Faramarzi et al., 2020) با مدل‌سازی رفتار شکار در زیست‌گاه‌های اقیانوسی توانسته است نرخ همگرایی بالایی را به‌ویژه در مسائل چندقله‌ای فراهم آورد. در کنار این روش‌ها، طیف گسترده‌ای از الگوریتم‌های ترکیبی، یادگیری‌محور و شبکه‌مبنا نیز در سال‌های اخیر مطرح شده‌اند که هر یک با هدف بهبود دقت، سرعت و همگرایی طراحی گردیده‌اند و نشان‌دهنده روند رو به رشد توسعه روش‌های فرابتکاری در حوزه هوش محاسباتی هستند.

این مقالات نشان‌دهنده‌ی پیشرفت‌های اخیر در بهینه‌سازی الگوریتم‌های محاسباتی با استفاده از معماری CUDA هستند.

روش پیشنهادی

روش کار در این مطالعه به این صورت است که در مرحله اول برای پیاده سازی الگوریتم بهینه‌سازی روباه پرنده بر روی CPU ابتدا به معرفی کامل این الگوریتم در بخش ۱-۲ انجام شده و بررسی رفتار روباه‌ها پرداخته و گام‌های اجرایی آن را مورد بررسی قرار می‌دهیم و با در دست داشتن مستندات لازم الگوریتم را بر روی نرم افزار MATLAB پیاده سازی می‌کنیم. در مرحله دوم برای پیاده سازی موازی این الگوریتم نیز از نرم افزار MATLAB استفاده کرده در بخش ۲-۲ پیاده‌سازی الگوریتم FFOA بر روی GPU را مورد بررسی قرار می‌دهیم.

مراحل کلی الگوریتم بهینه‌سازی روباه پرنده و فرآیند موازی‌سازی آن با استفاده از CUDA در MATLAB را می‌توان به صورت یک دیاگرام ساده به ترتیب زیر در نظر گرفت که شامل ۹ مرحله است که خلاصه‌ی این مراسم در فلوچارت شکل ۱ نمایش داده شده است.

۱. شروع

۲. تعیین پارامترهای اولیه (تعداد جمعیت، ابعاد مسئله، تعداد نسل‌ها)

۳. ایجاد جمعیت اولیه روباه‌های پرنده

۴. محاسبه ارزش هر روباه پرنده در جمعیت

۵. به‌روزرسانی موقعیت روباه‌ها بر اساس رفتار شکار و حرکت

۶. بررسی شرایط توقف (حداکثر تعداد نسل‌ها یا همگرایی)

اگر شرایط توقف برآورده شود: پایان

[1] Slime Mould Algorithm
[2] Marine Predators Algorithm

در غیر این صورت: انتقال به مرحله بعدی

۷. اجرای الگوریتم به صورت موازی بر روی GPU با استفاده از CUDA

۸. تکرار مراحل ۴ تا ۷ برای تعداد مشخصی از نسل‌ها

۹. نتایج نهایی (بهترین جواب به دست آمده)

شکل ۱. دیاگرام مربوط به مراحل روش پیشنهادی

- **پیاده سازی الگوریتم بهینه‌سازی روباه پرنده بر روی CPU**

الگوریتم بهینه‌سازی روباه پرنده[1] یا FFOA یک الگوریتم بهینه‌سازی مبتنی بر طبیعت است که از رفتار جستجوی غذا و مهاجرت گروهی روباه‌های پرنده الهام گرفته شده است. این الگوریتم در زمینه حل مسائل بهینه پیچیده و چندبعدی کاربرد دارد. در این الگوریتم، یک جمعیت از "روباه‌های پرنده" ایجاد می‌شود که هر کدام نمایانگر یک نقطه در فضای جستجو هستند. روباه‌های پرنده نوعی از خفاش‌ها هستند که از درختی به درخت دیگر پرواز کرده و به جستجوی منابع غذایی می‌پردازند. مراحل کار الگوریتم به طور کلی شامل مراحل زیر می‌باشد:

شروع و مقداردهی اولیه: جمعیتی از روباه‌های پرنده تولید می شوند که هر کدام نمایانگر یک جواب ممکن برای مسئله است. مقادیر اولیه مانند تعداد روباه‌ها، حداکثر تعداد تکرارها، و محدوده متغیرها مشخص می‌شود.

[1] Flying Fox Optimization Algorithm

تعیین تابع هدف: برای هر روباه بر ا ساس موقعیت خود در ف ضای جـ ستجو، مقدار تابع هدف محا سبه می شود. هدف الگوریتم این است که موقعیتی پیدا کند که کمینه یا بیشینه مقدار تابع هدف را داشته باشد.

حرکت روباه‌ها: روباه‌ها با اسـتفاده از یک مدل حرکتی که به ترکیب پرواز مسـتقیم و جسـتجوی محلی بسـتگی دارد، موقعیت خود را تغییر می‌دهند. حرکت به سمت بهترین موقعیت‌های شنا سایی شده (بر ا ساس تابع هدف) صورت می‌گیرد.

بهبود موقعیت‌ها: اگر موقعیت جدید بهتر از موقعیت قبلی باشـد، به‌روزرسـانی انجام می‌شـود. رفتارهایی مانند اکتشاف[1] و استخراج[2] برای تضمین جستجوی کل فضای مسئله و تمرکز روی نواحی بهینه استفاده می‌شوند.

دسـتیابی به معیار توقف: الگوریتم تا زمانی ادامه پیدا می‌کند که به تعداد تکرار مشـخصـی برسـد یا تغییرات معنی‌داری در مقدار تابع هدف رخ ندهد.

برای شـبیه‌سـازی ریاضـی الگوریتم FFOA، باید ابتدا رفتارهای کلیدی این الگوریتم را مدل‌سـازی کنیم. این رفتارها شامل حرکت روباه‌های پرنده، اکتشاف و استخراج است.

برای شروع و مقداردهی اولیه در ابتدا باید متغیرهای الگوریتم تعریف گردند. فرض می‌کنیم:

N :تعداد روباه‌ها (جمعیت).

$X_i\ (t)$: موقعیت روباه i در تکرار t

$V_i\ (t)$: سرعت روباه i در تکرار t

$F\ (X)$: مقدار تابع هدف برای موقعیت X

X_{best} : بهترین موقعیت شناخته‌شده تا کنون.

برای مقداردهی اولیه در ابتدا موقعیت $X_i\ (0)$ و سـرعت $V_i\ (0)$ برای هر روباه به صـورت تصـادفی و مطابق با رابطه (۱) مقداردهی می‌شوند (Połap & Woźniak, 2021) .

$$X_i\ (0) \sim Uniform\ (X_{min}, X_{max}) \quad (۱)$$

برای به‌روزرسـانی موقعیت‌ها، موقعیت جدید $X_i\ (t+1)$: هر روباه با اسـتفاده از موقعیت فعلی و سـرعت آن مطابق با رابطه (۲) تعیین می‌شود.

$$X_i\ (t+1) = X_i\ (t) + V_i\ (t) \quad (۲)$$

برای به‌روزرسانی سرعت، سرعت روباه با توجه به ترکیبی از سه عامل جاذبه به سمت بهترین موقعیت X_{best} ، یک عامل تصادفی برای اکتشاف فضای جدید و یک فاکتور کاهش سرعت برای جلوگیری از نوسانات توسط رابطه (۳) به‌روزرسـانی می‌شـود. در رابطه زیر w برابر با ضـریب اینرسـی (کنترل پایداری حرکت)، c_1 و c_2 برابر با

[1] Exploration
[2] Exploitation

ضـرایب یادگیری (کنترل شـدت جاذبه)، r_1 وr_2 برابر با اعداد تصـادفی در بازه $[0,1]$ و X_{rand} یک موقعیت تصادفی برای اکتشاف است.

$$V_i\,(t+1) = w.V_i\,(t) + c_1.r_1\,.\big(X_{best} - X_i(t)\big) + c_2.r_2\,.(X_{rand} - X_i(t)) \quad (۳)$$

برای ارزیابی تابع هدف نیز مقدار تابع هدف به ازای هر روباه توسط رابطه (۴) محاسبه می‌شود.

$$f\big(X_i(t+1)\big) \quad (۴)$$
$$X_{best} = argmin_i\, f(x_i(t))$$

الگوریتم تا زمانی اجرا می‌شـود که شـرط توقف حاصـل گردد. برای توقف یا تعداد تکرارها باید به حداکثر مقدار خود برسد یا تغییرات در بهترین مقدار تابع هدف از یک آستانه کمتر شود. به طور خلاصه برای شبیه‌سازی این الگوریتم ابتدا مقداردهی اولیه انجام شده و موقعیت‌ها و سرعت‌ها به‌صورت تصادفی انتخاب می‌شوند، سپس برای هر تکرار در حلقه‌ی اصـلی موقعیت‌ها و سـرعت به‌روزرسـانی و بهترین موقعیت شـناسـایی و ثبت می‌شـود و در نهایت بهترین جواب و مقدار تابع هدف به عنوان خروجی بدسـت می‌آید. کدهای مربوط به پیاده‌سـازی الگوریتم FFOA بر روی CPU در محیط MATLAB در بخش پیوست‌ها (پیوست A) نمایش داده شده است. شکل ۲ بخشی از خروجیِ الگوریتم اجرایی بر روی CPU را به ازای جمعیت ۱۰۰۰ روباه نمایش می‌دهد.

```
Iteration 175/200, Best Fitness: 0.014490
Iteration 176/200, Best Fitness: 0.014490
Iteration 177/200, Best Fitness: 0.014490
Iteration 178/200, Best Fitness: 0.014490
Iteration 179/200, Best Fitness: 0.014490
Iteration 180/200, Best Fitness: 0.014490
Iteration 181/200, Best Fitness: 0.014490
Iteration 182/200, Best Fitness: 0.014490
Iteration 183/200, Best Fitness: 0.014490
Iteration 184/200, Best Fitness: 0.014490
Iteration 185/200, Best Fitness: 0.014490
Iteration 186/200, Best Fitness: 0.014490
Iteration 187/200, Best Fitness: 0.014490
Iteration 188/200, Best Fitness: 0.014490
Iteration 189/200, Best Fitness: 0.014479
Iteration 190/200, Best Fitness: 0.014479
Iteration 191/200, Best Fitness: 0.014408
Iteration 192/200, Best Fitness: 0.014408
Iteration 193/200, Best Fitness: 0.014408
Iteration 194/200, Best Fitness: 0.014408
Iteration 195/200, Best Fitness: 0.014408
Iteration 196/200, Best Fitness: 0.014408
Iteration 197/200, Best Fitness: 0.014408
Iteration 198/200, Best Fitness: 0.014408
Iteration 199/200, Best Fitness: 0.014408
Iteration 200/200, Best Fitness: 0.014408
Optimization Completed:
Best Fitness: 0.014408
Best Position: -0.00070466   0.0037677   0.0083958   0.070153   0.0082492   -0.096047  -0.0061096   0.0055934  -0.0063063 -0.00026509
```

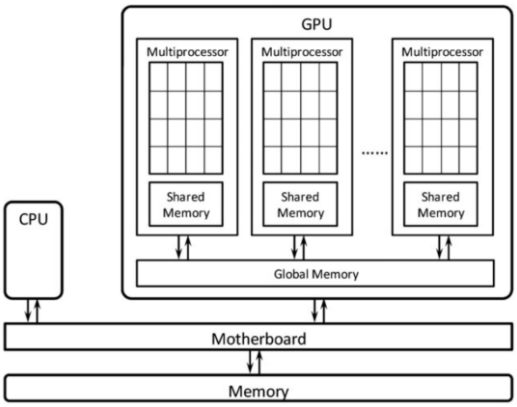
شکل ۲ . خروجی الگوریتم بهینه‌سازی روباه پرنده (FFOA) بر روی CPU به ازای ۱۰۰۰ روباه

- **پیاده سازی الگوریتم بهینه‌سازی روباه پرنده بر روی GPU**

برای پیاده‌سـازی موازیِ الگوریتم در متلب با اسـتفاده از معماری CUDA (Compute Unified Device Architecture) و پردازنده‌های گرافیکی (GPU)، از قابلیت‌های Parallel Computing Toolbox متلب

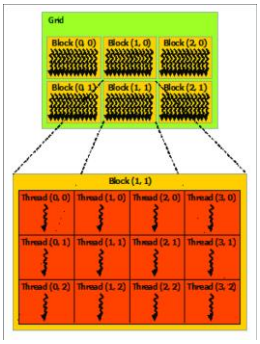
استفاده خواهیم کرد. این ابزار امکان انتقال محاسبات به GPU را فراهم می‌کند، که می‌تواند عملکرد الگوریتم‌های سنگین را بهبود بخشد. نرم‌افزار MATLAB با توجه به دستورها و توابع ساده و محیط داده‌ایِ برداری که دارد، یکی از نرم افزارهای برنامه نویسی سطح بالا به حساب می‌آید که بروی تکنیک‌های محاسباتی متمرکز شده است. CUDA از مفهوم سوئیچ‌های تک چرخه‌ای استفاده می‌کند تا تاخیر مسیر داده و دسترسی به حافظه را پنهان کند. متد مدیریتیِ اجرای CUDA تقسیم گروه‌هایی از نخ‌ها در میان بلاک‌ها است. گرید[1]ها از مجموعه‌ای از بلاک‌ها تشکیل شده‌اند. نخ‌ها می‌توانند مکان خودشان را در یک بلاک و مکان بلاکِ درون گرید را با المان‌های داده‌ی ذاتی که بوسیله CUDA مقداردهی اولیه شده، مشخص کنند. نخ‌های بلاک‌های مختلفِ متعلق به یک گرید می‌توانند از طریق عملیات اتمیک، در فضای حافظه سراسری که با بقیه نخ‌ها به اشتراک گذاشته شده است هماهنگ شوند. در برنامه‌نویسی موازی، مفاهیم نخ (Thread)، ترد (Thread) و بلاک (Block) و حافظه اشتراکی (SP[2]) اهمیت زیادی دارند. نخ‌ها واحدهای اجرایی کوچک در پردازش موازی هستند که هر یک می‌توانند بخشی از محاسبات را به طور مستقل انجام دهند. در پیاده سازی‌های موازی مانند CUDA یا OpenMP، نخ‌ها در یک واحد پردازشی به نام بلاک گروه‌بندی می‌شوند. هر بلاک معمولاً به تعدادی نخ اشاره دارد که به طور همزمان در پردازنده‌های مختلف اجرا می‌شوند. به این ترتیب، بلاک‌ها می‌توانند به‌طور موازی در یک پردازنده یا در واحدهای مختلف پردازشی اجرا شوند، که باعث تسریع عملیات‌های پیچیده می‌شود. یکی دیگر از مفاهیم کلیدی در پردازش موازی حافظه اشتراکی (SP) است. حافظه اشتراکی به فضایی اشاره دارد که توسط نخ‌های مختلف در یک بلاک به اشتراک گذاشته می‌شود و به آن‌ها این امکان را می‌دهد که داده‌ها را سریع‌تر و بدون نیاز به ارسال آن‌ها به حافظه اصلی، بین خود منتقل کنند. این نوع حافظه می‌تواند تأثیر چشم‌گیری بر عملکرد پردازش‌های موازی بگذارد زیرا می‌تواند تا حد زیادی از زمان تأخیر دسترسی به حافظه جلوگیری کند و باعث افزایش کارایی سیستم‌های موازی شود. در تحقیق حاضر، استفاده بهینه از این مفاهیم می‌تواند به اجرای سریع‌تر و کارآمدتر الگوریتم FFOA در محیط‌های موازی مانند MATLAB کمک کند و در نهایت به افزایش کارایی سیستم‌ها و بهبود نتایج علمی و صنعتی منجر شود.سخت افزار GPU NVIDIA (نمایش داده شده در شکل ۳) با استفاده از واحدهای چند پرداز شی SPMD ساخته شده است. نخ‌ها به گروه‌های ۳۲ بیتی که بسته نامیده می‌شوند گروه‌بندی شده و به SPها(جریان پردازنده) نگاشت می شوند. یک بسته واحدِ پایه برای زمان‌بندی است. همه‌ی ۳۲ بیت در یک بسته باید دستور یکسانی را اجرا کنند حتی اگر داده‌ی آنها متفاوت باشد. یک نخ به یک SP نگاشت می شود. بسته‌ها بوسیله MIU[3]ها ساخته، مدیریت و زمان‌بندی می شوند. تعداد نخ‌ها در هر بسته معمولا دو برابر تعداد SP ها است. تعداد نخ‌هایی که به یک بسته تعلق دارند با یک آدرس برنامه یکسان شروع می‌شوند و برای انشعاب و اجرا بصورت مستقل عمل می کنند.

[1] Grid
[2] Streaming Processor
[3] Multifunction Interface Unit



شکل ۳. GPU از دیدگاه پیاده‌سازی سخت افزار

در CUDA، بلاک واحد قابل زمان‌بندی ا ست که می‌تواند به چندپردازنده تخ صیص داده شود. زمانی که یک برنامه CUDA که کرنل[1] نامیده می شود روی GPU اجرا می شود هرنخ م شخصه خا صی دارد که به طور پویا تعیین می‌شـود. این مقادیر، هماهنگ کننده‌های بلاکِ خودشـان در گرید و هماهنگ کننده‌های نخِ خودشـان درون بلاک ه ستند. یک تمرین عمومی برای نگا شت نخ‌ها به مجموعه‌های م سئله این ا ست که یک نخ دا شته باشـیم که مسـئول هر المان درون داده خروجی باشـد. برای انجام این کار، ابعاد بلاک‌ها و نخ‌ها معمولا ابعاد مجموعه داده خروجی را نشان می‌دهند. در شکل ۴ مفهوم نخ، بلاک و گرید به تصویر کشیده شده است.



شکل ۴. نمایش معماری داخلی GPU (نمایش یک گرید به همراه بلاک‌ها و تردهای آن)

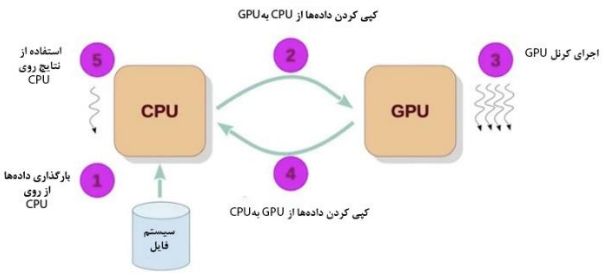
در ابتدای شـروع موازی‌سـازیِ الگوریتمِ FFOA باید مطمئن شـد سـیسـتم فعلی از معماری CUDA توسـط GPU پ شتیبانی می‌کند. با ا ستفاده از د ستور gpuDevice می‌توان اطلاعات GPU را برر سی کرد. نتیجه‌ی اجرای دستور gpuDevice توسط سیستم پیشنهادی در شکل ۵ نمایش داده شده است.

[1] Kernel

```
>> gpuDevice

ans =

  CUDADevice with properties:

                      Name: 'NVIDIA GeForce RTX 3050 6GB Laptop GPU'
                     Index: 1
         ComputeCapability: '8.6'
            SupportsDouble: 1
     GraphicsDriverVersion: '551.61'
                DriverModel: 'WDDM'
            ToolkitVersion: 11.8000
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152 (49.15 KB)
        MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 6441926656 (6.44 GB)
           AvailableMemory: 5398986752 (5.40 GB)
               CachePolicy: 'balanced'
      MultiprocessorCount: 20
             ClockRateKHz: 990000
               ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
          CanMapHostMemory: 1
           DeviceSupported: 1
           DeviceAvailable: 1
            DeviceSelected: 1
```

شکل ۵. نتیجه‌ی اجرای دستور gpuDevice

نرم افزار MATLAB امکان استفاده از آرایه های GPU را با تابع gpuArray فراهم می‌کند. عملیات محاسباتی روی این آرایه‌ها به صورت خودکار توسط GPU اجرا می‌شوند. هدف از موازی‌سازی تابع هدف انتقال محاسبات سنگین مانند ارزیابی تابع هدف بر روی GPU می‌باشد. پس از انجام محاسبات، نتایج به CPU منتقل می‌شوند تا بتوان از آن‌ها در مراحل بعدی استفاده کرد. خلاصه‌ای از فرآیند موازی‌سازی در شکل ۶ نمایش داده شده است.



شکل ۶. نمایش نحوه تبادل داده‌ها بین CPU و GPU

قطعه کد مربوط به موازی سازی الگوریتم FFOA در بخش پیوست‌ها (پیوست B) نمایش داده شده است. در این قطعه کد با استفاده از gpuArray، داده‌ها به حافظه GPU منتقل می شوند. عملیات ریاضی مانند جمع و تفریق روی این آرایه‌ها به صورت خودکار توسط GPU انجام شده و از تابع arrayfun برای اجرای محاسبات

روی هر عنصر آرایه در GPU استفاده می‌شود. در این قطعه کد بخشِ مربوط به محاسبه تابع هدف برای هر روباه به صورت موازی روی GPU اجرا می‌شود، سپس توسط دستور gather نتایج محاسبات GPU به حافظه CPU منتقل می‌گردد. شکل ۷ نیز بخشی از خروجیِ الگوریتم اجرایی بر روی GPU را به ازای جمعیت ۱۰۰۰ روباه نمایش می‌دهد. ذکر این نکته ضروری‌ست که برای استفاده از این قطعه کد، باید GPU سیستم مورد استفاده از معماری CUDA پشتیبانی کرده، همچنین درایورهای مناسب و Parallel Computing Toolbox نیز نصب باشند.

```
Iteration 76: Best Fitness = 1.2608e-05
Iteration 77: Best Fitness = 1.2608e-05
Iteration 78: Best Fitness = 1.2608e-05
Iteration 79: Best Fitness = 1.2608e-05
Iteration 80: Best Fitness = 1.2608e-05
Iteration 81: Best Fitness = 1.2608e-05
Iteration 82: Best Fitness = 3.5681e-06
Iteration 83: Best Fitness = 3.5681e-06
Iteration 84: Best Fitness = 3.5681e-06
Iteration 85: Best Fitness = 3.5681e-06
Iteration 86: Best Fitness = 3.5681e-06
Iteration 87: Best Fitness = 3.5681e-06
Iteration 88: Best Fitness = 3.5681e-06
Iteration 89: Best Fitness = 3.5681e-06
Iteration 90: Best Fitness = 3.5681e-06
Iteration 91: Best Fitness = 3.5681e-06
Iteration 92: Best Fitness = 3.5681e-06
Iteration 93: Best Fitness = 3.5681e-06
Iteration 94: Best Fitness = 3.5681e-06
Iteration 95: Best Fitness = 3.5681e-06
Iteration 96: Best Fitness = 3.5681e-06
Iteration 97: Best Fitness = 3.5681e-06
Iteration 98: Best Fitness = 3.5681e-06
Iteration 99: Best Fitness = 3.5681e-06
Iteration 100: Best Fitness = 3.5681e-06
Best Position Found:
   -0.0019   -0.0003

Best Fitness Value: 3.5681e-06
```

شکل ۷ . خروجی الگوریتم بهینه‌سازی روباه پرنده (FFOA) بر روی GPU به ازای ۱۰۰۰ روباه

نتایج تجربی

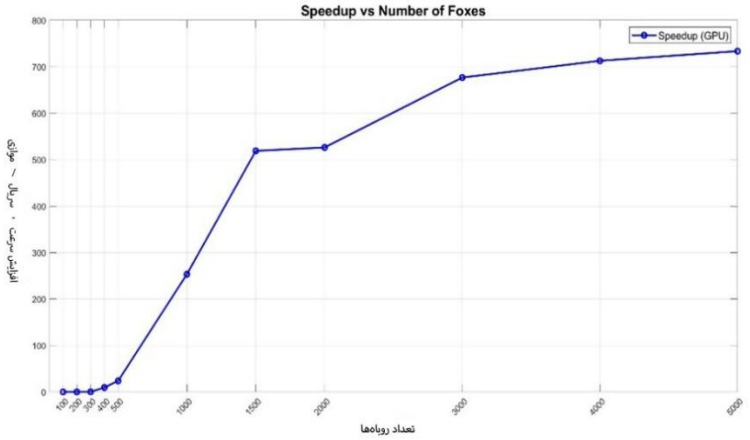
در این بخش نتایج مربوط به مقایسه زمان اجراها بر روی دو بستر CPU وGPU و محاسبه SpeedUp نمایش داده شده است. برای مقایسه زمان اجرای الگوریتم با استفاده از CPU و GPU، باید زمان اجرای هر نسخه از الگوریتم را اندازه‌گیری کرد. در MATLAB، می‌توان از دستور tic و toc برای اندازه‌گیری زمان اجرا استفاده کرد. در بخش پیوست‌ها، پیوست C قطعه کدی را نمایش می‌دهد که الگوریتم FFOA را با CPU و GPU اجرا کرده و زمان اجرای آن‌ها را مقایسه می‌کند. برای مشاهده تفاوت واضح بین CPU و GPU، تعداد روباه‌ها (جمعیت) و ابعاد مسئله باید به حد کافی بالا باشد. عملکرد GPU به مدل کارت گرافیک و حجم حافظه آن بستگی دارد. علاوه بر این، اگر انتقال داده‌ها بین CPU و GPU زیاد باشد، ممکن است عملکرد GPU تحت تأثیر قرار گیرد زیرا سربار انتقال داده‌ها می‌تواند باعث کاهش کارایی شود. در این تحقیق، برای مقایسه زمان

اجرای الگوریتم FFOA در دو حالت سریال و موازی، تعداد تردها برابر با ۱۰۰ در نظر گرفته شده است. در هر بار اجرای الگوریتم، تعداد بلاک‌ها بر اساس تعداد جمعیت روباه‌ها محاسبه می‌شود. در شکل ۸ در نقاط ابتدایی، اندازه تردها ۱۰۰ در نظر گرفته شده است و همچنین اندازه جمعیت روباه‌ها نیز برابر با ۱۰۰ در نظر گرفته شده است. تعداد بلاک‌ها در این حالت برابر با ۱۰۰ تقسیم بر ۱۰۰ است که در این مثال برابر با یک بلاک می‌شود؛ به این معنی که تعداد روباه‌ها برابر با تعداد تردها است. در این حالت، به دلیل سربار ناشی از انتقال داده‌ها به حافظه پردازنده گرافیکی، زمان اجرای محاسبات روی سخت‌افزار GPU بیشتر از زمان اجرای آن روی CPU است. این سربار ناشی از نیاز به انتقال داده‌های زیاد از CPU به GPU و بالعکس است که می‌تواند زمان اضافی برای جابجایی داده‌ها و هم‌زمانی پردازش‌ها ایجاد کند. به رابطه دیگر، زمانی که داده‌ها باید بارها و بارها بین حافظه اصلی (RAM) و حافظه پردازنده گرافیکی (GPU) منتقل شوند، این انتقال‌ها می‌توانند عملکرد GPU را تحت تأثیر قرار دهند و مزایای استفاده از GPU را در پردازش‌های سریع محدود کنند. اما با افزایش تعداد روباه‌ها و تعداد بلاک‌ها، زمان اجرای الگوریتم به طور تدریجی کاهش می‌یابد. این کاهش زمان به دلیل اجرای همزمان بلاک‌ها روی سخت‌افزار GPU است که به موجب آن پردازش‌های متعدد به طور موازی و بهینه انجام می شود. در این حالت، افزایش تعداد بلاک‌ها و تردها کمک می‌کند تا از ظرفیت پردازشی بالا و موازی GPU استفاده بهینه شود. با این حال، باید به این نکته توجه داشت که با افزایش تعداد بلاک‌ها و تردها، پیچیدگی مدیریت حافظه و هماهنگی بین پردازش‌ها نیز افزایش می‌یابد که می‌تواند خود به یکی از چالش‌های اجرای مؤثر الگوریتم تبدیل شود. در نتیجه، اگرچه GPU به طور کلی می‌تواند در پردازش‌های موازی سرعت بیشتری را ارائه دهد، اما مشکلاتی نظیر سربار انتقال داده‌ها و پیچیدگی‌های مدیریت حافظه می‌تواند عملکرد آن را تحت تأثیر قرار دهد. این مسائل باید در نظر گرفته شوند تا از توانایی‌های کامل GPU بهره‌برداری شود و بتوان آن را در کاربردهای پیچیده‌تر و مقیاس‌های بزرگتر به‌طور مؤثرتر به کار برد.

شکل ۸ نشان می‌دهد که با افزایش تعداد جمعیت روباه‌ها سرعت پردازش الگوریتم در GPU بیشتر از CPU شده و افزیش چشم گیری داشته است. اگر تعداد جمعیت روباه‌ها به اندازه کافی بزرگ نباشد به علت سربار ناشی از انتقالِ بین دو سخت افزار GPU و CPU سرعت اجرای الگوریتم در GPU کمتر از CPU می شود. الگوریتم FFOA در حالت موازی بر روی بستر MATLAB با قابلیت پشتیبانی از معماری CUDA اجرا شده است. زمان اجرای محاسبات در هر دو حالت سریال و موازی بر حسب ثانیه می‌باشد. برای انجام این محاسبات در حالت سریال از پردازنده مرکزی Intel Core i9 و در حالت موازی از پردازنده گرافیکی با مشخصات NVIDIA GeForce RTX 3050 6GB Laptop GPU استفاده شده است. برای بررسی اینکه با پردازش موازی اجرای الگوریتم چند برابر سریع‌تر شده است، مطابق با رابطه (۵) از محاسبه Speedup استفاده می‌کنیم.

$$\text{Speedup} = \text{Serial Time} / \text{Parallel Time} \quad (۵)$$

Speedup از تقسیم زمان اجرای سریال[1] (اجرا بر روی CPU) و زمان اجرای موازی[2] (اجرا بر روی GPU) به دست می‌آید. اگر مقدار Speedup برابر با ۲ باشد، به این معناست که پردازش موازی دو برابر سریع‌تر از پردازش سریال شده و اگر Speedup برابر با ۱ باشد، هیچ بهبودی حاصل نشده است. در واقع Speedup بزرگ‌تر از یک نشان‌دهنده بهبود عملکرد روش اجرایی دارد. نمودار شکل ۱۰ مقایسه زمان اجرای الگوریتم FFOA بر روی سخت افزار GPU را در مقابل اجرا بر روی سخت افزار CPU نشان می‌دهد. در این نمودار محور افقی برابر با تعداد روباه‌ها و محور عمودی برابر با مقدار Speedup است. با Speedup نمایش داده شده در نمودار شکل ۱۱ می‌توان نتیجه گرفت که با افزایش جمعیت روباه‌ها سرعت اجرای الگوریتم بر روی سخت افزار GPU حدود ۳۱۴,۰۷۴ برابر افزایش یافته است. خروجی شکل زیر شامل چندین بخش کلیدی است. ابتدا، موقعیت بهینه روباه‌ها که نشان‌دهنده بهترین جواب ممکن است. همچنین، ارزش یا fitness هر روباه که کیفیت راه‌حل پیشنهادی را مشخص می‌کند که در طول نسل‌ها به‌روزرسانی می‌شود. تعداد نسل‌ها و زمان اجرای الگوریتم نیز به‌عنوان پارامترهای مهم وارد می‌شوند که به‌ویژه در حالت موازی روی GPU زمان اجرا کاهش می‌یابد.



شکل ۸ . Speedup بدست آمده از اجرای الگوریتم FFOA در دو حالت سریال (بر روی CPU) و موازی (بر روی GPU) به ازای جمعیت روباه‌های متفاوت

جدول ۱ مقایسه‌ای از Speedup روش پیشنهادی با برخی الگوریتم‌های فراابتکاری جدید انجام شده است. نتایج بدست آمده نشان می‌دهد که نسخه‌ای بهبود یافته و موازی‌شده از الگوریتم کرم شب‌تاب که با استفاده از بهینه‌سازی موازی در محیط چند رشته‌ای[3] سرعت اجرای الگوریتم را بهبود داده است عملکرد بهتری نسبت به سایر مقالات بررسی شده داشته است. تمرکز اصلی روی کاهش زمان اجرا[4] و افزایش بهره‌وری در حل مسائل

[1] Serial Time
[2] Parallel Time
[3] Multi-threading
[4] Runtime

پیچیده بهینه‌سازی بوده است. نتایج جدول ۱ نشان داده‌اند که نسخه موازی شده نسبت به نسخه اصلی و دیگر الگوریتم‌ها دارای Speedup بالاتری است.

جدول ۱. مقایسه Speedup روش پیشنهادی با برخی از الگوریتم‌های فرا ابتکاری جدید

نام الگوریتم	نویسندگان	مرجع	سال ارائه	تکنیک مورد استفاده	Speedup
HGS	یانگ و همکاران	[۲۶]	۲۰۲۱	الهام‌گرفته از رفتار جستجوی گرسنگی حیوانات، تعادل بین اکتشاف و بهره‌برداری، عملکرد عالی در توابع مقیاس بزرگ	3.4x
AOA	ابوعلیگاه و همکاران	[۲۷]	۲۰۲۱	بهره‌گیری از مفاهیم ریاضی محض برای ایجاد تعادل بین جستجوی محلی و جهانی	2.9x
SMA	چن و همکاران	[۲۸]	۲۰۲۰	الگوریتمی بر پایه حرکت کپک مخاطی که از همگرایی پیشرونده بهره می‌برد	3.5x
MPA	فرامرزی و همکاران	[29]	۲۰۲۰	مبتنی بر رفتار شکارچیان دریایی، عملکرد قوی در توابع پیچیده و کاربردهای صنعتی	3.8x
FFOA	حسینی و خیری	روش پیشنهادی	-	الگوریتم بهینه‌سازی روباه پرنده	5.2x

نتیجه‌گیری و پیشنهادهای آتی

در این مقاله یک راه کار جدید برای پیاده سازی الگوریتم بهینه‌سازی روباه پرنده در حالت موازی بر روی محیط MATLABارائه شد. همان طورکه نتایج نشان داد در حالت سریال با افزایش جمعیت روباه‌ها، زمان اجرای محاسبات CPU نیز افزایش می‌یابد، در حالی که زمان اجرای این الگوریتم در حالت موازی با استفاده از امکانات معماری CUDA و قدرت پردازش موازی ارائه شده توسط GPU بشدت کاهش یافته است. هم چنین مقایسه زمان اجرای الگوریتم FFOA در دو حالت سریال و موازی نشان داد که پیاده‌سازی این الگوریتم بر روی سخت افزارGPU با استفاده از قابلیت همزمانی اجرای بلاک‌ها می تواند سرعت اجرا را حدود ۳۱۴,۰۷۴ برابر نسبت به سخت افزارCPU افزایش دهد. به دلیل کاربر پسند بودنِ سخت افزارهای GPU معماریِ این سخت افزارها دائما در حال رشد و توسعه می‌باشد و می‌توان در آینده‌ای نزدیک با به کارگیری این سخت افزارها شاهد پیشرفت چشم‌گیری در کاهش زمان اجرای محاسبات سنگین بر روی محیط های موازی بود. استفاده از این الگوریتم در محیط‌های محاسباتی موازی مانند MATLAB می‌تواند سرعت پردازش و دقت نتایج را به‌طور چشمگیری افزایش دهد. این امر به ویژه در مسائلی که نیاز به محاسبات پیچیده و زمان‌بر دارند، بسیار حائز اهمیت است و

می‌تواند به ارتقاء بهره‌وری در صنایع مختلف کمک کند. به همین دلیل، گسترش این رویکرد در کاربردهای علمی و صنعتی می‌تواند تأثیرات مثبتی بر عملکرد سیستم‌ها و حل مسائل پیچیده در این حوزه‌ها داشته باشد. با توجه به بهبودهای حاصل‌شده، این روش می‌تواند در طیف گسترده‌ای از مسائل علمی و صنعتی کاربرد داشته باشد. از جمله این کاربردها می‌توان به شبیه‌سازی‌های عددی در علوم مهندسی، پردازش داده‌های حجیم در یادگیری ماشین و هوش مصنوعی، تحلیل تصاویر پزشکی برای تشخیص بیماری‌ها، مدل‌سازی سیستم‌های پیچیده در فیزیک و شیمی، و حتی پردازش داده‌های بلادرنگ در حوزه‌هایی مانند خودروهای خودران و سامانه‌های پیش‌بینی آب‌وهوا اشاره کرد. این بهینه‌سازی‌ها می‌توانند زمان محاسبات را کاهش داده و امکان تحلیل داده‌های بزرگ را با دقت و سرعت بالاتر فراهم کنند، که در نهایت به بهبود تصمیم‌گیری در سیستم‌های مبتنی بر داده منجر می‌شود. در نهایت، این تحقیق به وضوح اهمیت استفاده از الگوریتم‌های پیشرفته و تکنیک‌های محاسباتی موازی را در بهبود عملکرد سیستم‌ها و حل مسائل پیچیده نشان می‌دهد. با توجه به مزایای اثبات شده این روش‌ها، پژوهشگران و متخصصان باید به پیاده‌سازی آن‌ها توجه بیشتری داشته باشند تا در راستای ارتقاء بهره‌وری و دقت در تحقیقات علمی و صنعتی گام‌های مؤثری برداشته شود.

تحقیقات آینده می‌توانند به بهینه‌سازی الگوریتم FFOA با توجه به مقیاس‌های بزرگتر پرداخته و راهکارهایی برای افزایش کارایی آن در پردازش‌های پیچیده‌تر ارائه دهند. همچنین مطالعه کاربرد الگوریتم FFOA در صنایع مختلف مانند انرژی، خودرو، یا پزشکی می‌تواند به شناسایی مشکلات خاص و چالش‌های اجرایی آن در محیط‌های صنعتی کمک کند. استفاده از سایر زبان‌های برنامه‌نویسی موازی در کنارMATLAB ، پیاده‌سازی این الگوریتم در سایر زبان‌ها و پلتفرم‌های موازی مانند Python یا CUDA می‌تواند عملکردهای بهینه‌تری را ارائه دهد که نیاز به بررسی دارد. برای اعتبارسنجی نتایج به دست‌آمده، پیشنهاد می‌شود که تحقیقات آینده شامل پیاده‌سازی الگوریتم در شرایط واقعی و شبیه‌سازی‌های پیشرفته باشد تا تفاوت‌های عملکردی در محیط‌های عملی شبیه‌سازی شود.

مراجع

1. Afzal, A., Buradi, A., Jilte, R., Shaik, S., Kaladgi, A. R., Arıcı, M., Lee, C. T., & Nižetić, S. (۲۰۲۳). Optimizing the thermal performance of solar energy devices using meta-heuristic algorithms: A critical review. *Renewable and Sustainable Energy Reviews*, ۱۷۳, .۱۱۲۹۰۳. https://doi.org/10.1016/j.rser.2022.112903

2. Zervoudakis, K., & Tsafarakis, S. (۲۰۲۵). Customer segmentation using flying fox optimization algorithm. *Journal of Combinatorial Optimization*, ٤٩(١), ٥. https://doi.org/10.1007/s10878-024-01243-6 .

3. Zervoudakis, K., & Tsafarakis, S. (۲۰۲۳). A global optimizer inspired from the survival strategies of flying foxes. *Engineering with Computers*, ۳۹(۲), .١٦١٦-١٥٨٣ . https://doi.org/10.1007/s00366-021-01554-w

4. Aalloul, R., Elaissaoui, A., Benlattar, M., & Adhiri, R. (۲۰۲۳). Emerging parameters extraction method of PV modules based on the survival strategies of flying foxes optimization (FFO). *Energies*, ۱٦(۸), .۳٥۳۱. https://doi.org/10.3390/en16083531

5. Mohammed, H., & Rashid, T. (۲۰۲۳). FOX: a FOX-inspired optimization algorithm. *Applied Intelligence*, ٥۳(۱), .۱۰٥۰-۱۰۳۰. https://doi.org/10.1007/s10489-022-03533-0

6. Afif, M., Said, Y., & Atri, M. (۲۰۲۰). Computer vision algorithms acceleration using graphic processors NVIDIA CUDA. *Cluster Computing*, ۲۳(٤), .۳۳٤۷-۳۳۳٥ , https://doi.org/10.1007/s10586-020-03090-6

7. Kondratyuk, N., Nikolskiy, V., Pavlov, D., & Stegailov, V. (2021). GPU-accelerated molecular dynamics: State-of-art software performance and porting from Nvidia CUDA to AMD HIP. *The International Journal of High Performance Computing Applications*, *35*(4), 312-324. https://doi.org/10.1177/10943420211008288

8. Reis, L., Bispo, J., & Cardoso, J. M. (2020). Compilation of MATLAB computations to CPU/GPU via C/OpenCL generation. *Concurrency and Computation: Practice and Experience*, *32*(22), e5854. https://doi.org/10.1002/cpe.5854
9. Sallow, A. B. (2021). Implementation and Analysis of Fractals Shapes using GPU-CUDA Model. *Academic Journal of Nawroz University*, *10*(2), 1-10. https://doi.org/10.25007/ajnu.v10n1a1030
10. Hoseini, F., & Shahbahrami, A. (2015). An efficient implementation of fuzzy edge detection using GPU in MATLAB. 2015 International Conference on High Performance Computing & Simulation (HPCS), https://doi.org/10.1109/HPCSim.2015.7237100
11. Hajarian, M., Shahbahrami, A., & Hoseini, F. (2016). A parallel solution for the 0–1 knapsack problem using firefly algorithm. 2016 1st Conference on Swarm Intelligence and Evolutionary Computation (CSIEC), https://doi.org/10.1109/CSIEC.2016.7482134
12. Hoseini, F. (2023). Speeding up the 0-1 Knapsack Problem Using Shuffled Frog Leaping Algorithm. *Journal of Advances in Computer Research*, *2*(2), 21. https://sanad.iau.ir/journal/acr/Article/704229?jid=704229
13. Benhari, M., & Hosseini, R. (2024). An Intelligent Ensemble Model of Uncertainty Management in Belief Network for the Classification of Microscopic Images to Detect Cervical Cancer. *Karafan Journal*, *21*(1), (In persian) 89-69. https://doi.org/10.48301/kssa.2023.404913.2625

14. Hoseini, F., Sepehrzadeh, H., & Kheyri, M. (2024). Presenting an" Adaption Ahead" Optimization Algorithm for Training Models Used in Deep Learning. *Karafan Journal*. (In persian)https://doi.org/۱۰,٤۸۳۰۱/kssa.۲۰۲٤,٤۲۷۰۱۲,۲۷۷۳

15. Karvan, F. (2025). Development of a Model of Self-motivated Learning in Design Based on Cognitive Knowledge and Information Processing Style with a Mediating Role of Attitude to Architecture. *Karafan Journal*, *21*(4), 227-249.(In persian) https://doi.org/10.48301/kssa.2024.416749.2711
16. Kaveh, M., & Mesgari, M. S. (2023). Application of meta-heuristic algorithms for training neural networks and deep learning architectures: A comprehensive review. *Neural Processing Letters*, *55*(4), 4519-4622. https://doi.org/10.1007/s11063-022-11055-6
17. Song, D. (2024). Optimization and acceleration of image processing algorithms based on CUDA parallel architecture. Third International Conference on Electronic Information Engineering, Big Data, and Computer Technology (EIBDCT 2024), https://doi.org/10.1117/12.3031375
18. Kuricheti, M. (2024). *Using modern C++ to improve CUDA programs* University of California, Davis]. https://doi.org/10.1137/120903683
19. Hong, Y.-Y., Lopez, D. J. D., & Wang, Y.-Y. (2024). Solar irradiance forecasting using a hybrid quantum neural network: A comparison on gpu-based workflow development platforms. *IEEE Access*, *12*, 145079-145094. https://doi.org/10.1109/ACCESS.2024.3472053
20. De Rango, A., Furnari, L., Senatore, A., Mendicino, G., Giordano, A., Macri, D., Utrera, G., & D'Ambrosio, D. (2023). Performance Analysis and Optimization of the CUDA Implementation of the Three-Dimensional Subsurface XCA-Flow Cellular Automaton. 2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), https://doi.org/10.1109/PDP59025.2023.00048
21. Zhuo, Y., Zhang, T., Du, F., & Liu, R. (2023). A parallel particle swarm optimization algorithm based on GPU/CUDA. *Applied Soft Computing*, *144*, 110499. https://doi.org/10.1016/j.asoc.2023.110499
22. Narayana, S., Chandanapalli, S. B., Rao, M. S., & Srinivas, K. (2022). Ant cat swarm optimization-enabled deep recurrent neural network for big data classification based on map reduce framework. *The Computer Journal*, *65*(12), 3167-3180. https://doi.org/10.1093/comjnl/bxab135
23. Chauhan, D., Singh, S., Singh, S., & Banga, V. K. (2014). Speedup of type-1 fuzzy logic systems on graphics processing units using cuda. International Conference on Communication, Computing and Systems, https://doi.org/10.1145/2833258.2833306
24. Johar, F. M., Azmin, F. A., Suaidi, M. K., Shibghatullah, A. S., Ahmad, B. H., Salleh, S. N., Abd Aziz, M. Z. A., & Shukor, M. M. (2013). A review of genetic algorithms and parallel genetic algorithms on graphics processing unit (GPU). 2013 IEEE International Conference on Control System,

Computing and Engineering, https://doi.org/10.1109/ICCSCE.2013.6719971

25. Oiso, M., Yasuda, T., Ohkura, K., & Matumura, Y. (2011). Accelerating steady-state genetic algorithms based on CUDA architecture. 2011 IEEE congress of evolutionary computation (CEC), https://doi.org/10.1109/CEC.2011.5949685
26. Yang, Y., Chen, H., Heidari, A. A., & Gandomi, A. H. (2021). Hunger games search: Visions, conception, implementation, deep analysis, perspectives, and towards performance shifts. *Expert systems with applications*, *177*, 114864. https://doi.org/10.1016/j.eswa.2021.114864
27. Abualigah, L., Diabat, A., Mirjalili, S., Abd Elaziz, M., & Gandomi, A. H. (2021). The arithmetic optimization algorithm. *Computer methods in applied mechanics and engineering*, *376*, 113609. https://doi.org/10.1016/j.cma.2020.113609
28. Li, S., Chen, H., Wang, M., Heidari, A. A., & Mirjalili, S. (2020). Slime mould algorithm: A new method for stochastic optimization. *Future generation computer systems*, *111*, 300-323. https://doi.org/10.1016/j.future.2020.03.055
29. Faramarzi, A., Heidarinejad, M., Mirjalili, S., & Gandomi, A. H. (2020). Marine Predators Algorithm: A nature-inspired metaheuristic. *Expert systems with applications*, *152*, 113377. https://doi.org/10.1016/j.eswa.2020.113377