Usage of GPU for numerical optimization through the Artificial Bee Colony algorithm

Samuel Oreste Abreu Politecnico di Torino
Torino, Italia
s281568@studenti.polito.it

***Abstract*—This is the abstract.**

It consists of two paragraphs.

***Index Terms*—GPU; ABC; Numerical Optimization**

I. Introduction

The advent of computers came with the ability of performing mathematical calculations in a seamless manner, and brought with it the possibility of optimizing mathematical functions by finding its optimum set of values, this process is referred to as numerical optimization.

The realm of numerical optimization is vast and contains many algorithms whose inspiration can be traced back to the display of populations and the basic social rules that govern them. A particular family of algorithms are those that mimic the behaviour of insects that as a collective they appear to be intelligent, but when separated the individual agent does not appear to be autonomous, the organisms that fit this definition are referred to as swarm intelligence (SI).

In this work an SI algorithm known as Artificial Bee Colony (ABC) was chosen, which simulates the way in which honey bees split their work for foraging and securing food.

The novelty of this work is that of implementing the aforementioned algorithm in a GPU.

II. Background

A. Artificial Bee Colony

The Artificial Bee Colony algorithm [1] simulates the behaviour of bees whose main goal is to gather food. The different bees involved in this are:

- **Employed bee**: Bee that has found a food source and determined how good it is, it then goes to the hive to relay the food source's location to other bees through a "waggle dance".[2]
- **Onlooker bee**: Observes the employed bee's dance and determines if the food source is good enough.
- **Scout bee**: Once the current source has been depleted the bee is set to forage for other food sources, thus, it creates a random solution and, if its fitness is better than that of the previous solution the bee's task is set to employed, otherwise it changes to an onlooker.

The foundation of this algorithm lies in the exchange of information between foraging bees, this is were the intelligent behaviour is observed.

B. Implementation

Every single bee has an index $i \in [0, N-1] \cap \mathbb{N}$ that uniquely identifies them; $N = |SB| + |EB| + |OB|$ which relays that the total number of bees equals the sum of the amount of scout bees, employed bees and onlooker bees. (SB, EB and OB are vectors that contain the indices of scouting, employed and onlooker bees respectively).

The probability of any given employed bee to be chosen during the onlooker phase is:

$$P_i = \frac{fit_i}{\sum_{j \in EB} fit_j} \tag{1}$$

Where $i \in EB$.

The way in which new potential solutions are computed can be seen in the equation below.

$$v_{ij} = x_{ij} + \phi(x_{ij} - x_{kj}) \tag{2}$$

Where $i \in EB \cup OB$, $k \in EB$ and ϕ is a random number between 0 and 1.

There's a constraint however, $i \neq k$, which otherwise would not update the position.

0. Define the function to maximize and the amount of cycles to run for.
1. Initialize the bee's solutions.
2. Compute every bee's fitness.
3. The employed bee selects a random employed bee (With equal probability on selecting any of them), computes a potential solution (As seen in eq. 2 and its fitness, and, if it performs better, the new solution is adopted.
4. The onlooker bee selects a random employed bee (Giving preference to the bees with the best fitnesses as seen in eq. 1), computes a potential solution (As seen in eq. 2 and its fitness, and, if it performs better, the new solution is adopted.

5. In the event that a bee has not improved its solution for a given amount of cycles, the bee is turned into a scout bee, which sets its solution wiith random values.
6. Repeat from 3 until the cycles end.

C. Parallel implementation

In this parallel implementation every thread is interpreted to be a bee, the following .

1) Setting a bee's task: Since every thread is a bee, they have to be identified somehow, the proposed solution is that of using an enumerator type.

2) Obtaining random numbers: There were two possibilites:

- Creating and passing an array filled with pre-computed random values.
- Computing the random values in device.

Due to the potential large amount of memory being occupied the second option was chosen. (TODO: Check)

D. Code implementation

1) Obtaining random numbers.: The library used for obtaining random numbers is cuRAND.

2) Error detection: The library used for detecting errors is cuda_helper.h

III. Methodology

The application was developed using NVIDIA SDK's tools (cuda-gdb for debugging, cuda-memcheck for checking memory, among others)

A. Benchmark functions

For testing the algorithm's correcteness the following benchmark functions were implemented:

1) Rastrigin:

$$f(\mathbf{x}) = 10n + \sum_{i=1}^{n}(x_i^2 - 10cos(2\pi x_i)) \tag{3}$$

Where $\mathbf{x} \in \mathbb{R}^n$. The global minimum is $\mathbf{x} = [0, 0, ..., 0]$.

The contour and surface for a bidimensional Rastrigin can be seen in fig. 1 and fig. 2.

2) Sphere:

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2 \tag{4}$$

Where $\mathbf{x} \in \mathbb{R}^n$. The global minimum is $\mathbf{x} = [0, 0, ..., 0]$.

The surface for a bidimensional spheric function can be seen in fig. 3.

3) Rosenbrock:

$$f(\mathbf{x}) = \sum_{i=1}^{n-1}(100(x_{i+1} - x_i^2)^2 + (1 - x_i^2)) \tag{5}$$

Where $\mathbf{x} \in \mathbb{R}^n$. The global minimum is $\mathbf{x} = [1, 1, ..., 1]$.

The surface for a bidimensional Rosenbrock can be seen in fig. 4.

B. Mapping the functions

Due to the benchmark functions being a minimization problem they are mapped into a maximization function by applying:

$$fitness_{max} = \frac{1}{fitness_{min}} \tag{6}$$

The behaviour of this mapping applied to the Rosenbrock function can be seen in fig. 5 and fig. 6.

IV. Results

V. Figures

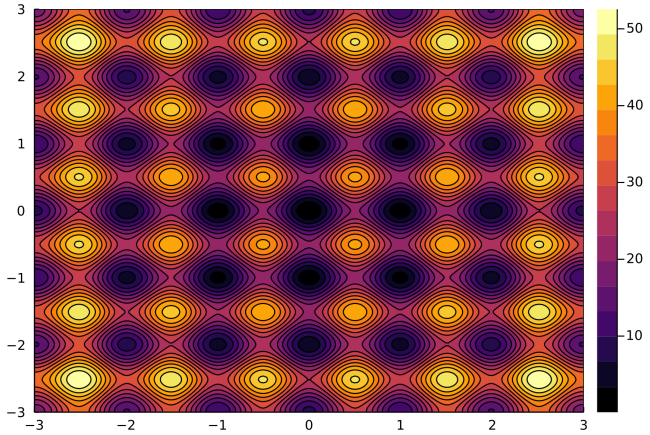

Figure 1: Contour of the Rastrigin function

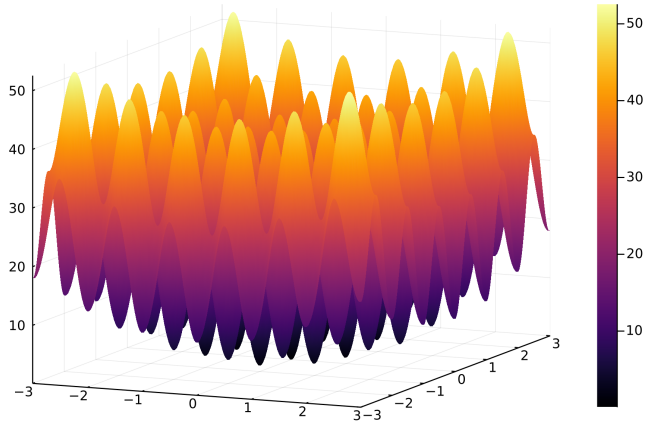

Figure 2: Surface of the Rastrigin function

VI. Conclusion

Parallel computation

VII. Further comments

A. Compilation step

In order to avoid the creation of a monolithic file that would contain all of the functions, the compilation is performed in a detached mode (Option -dc is added to the compiler's flags), which means that the compiler will create relocatable device code, which will be linked in a future step once all of the object files are generated.

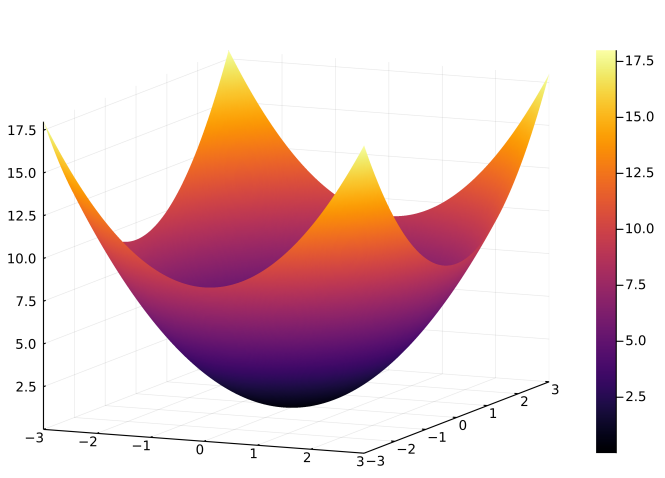

Figure 3: Surface of the spheric function

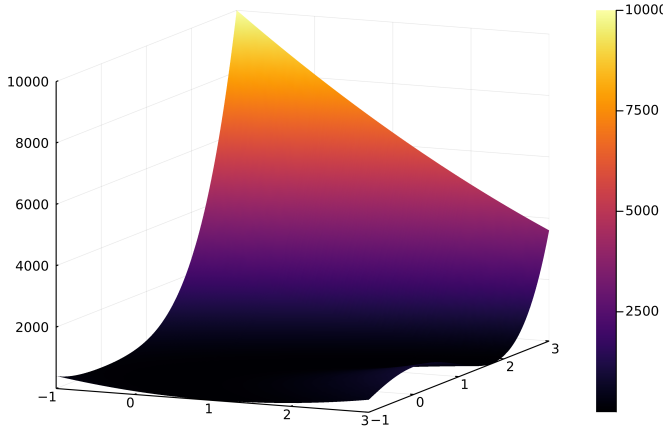

Figure 4: Surface of the Rosenbrock function

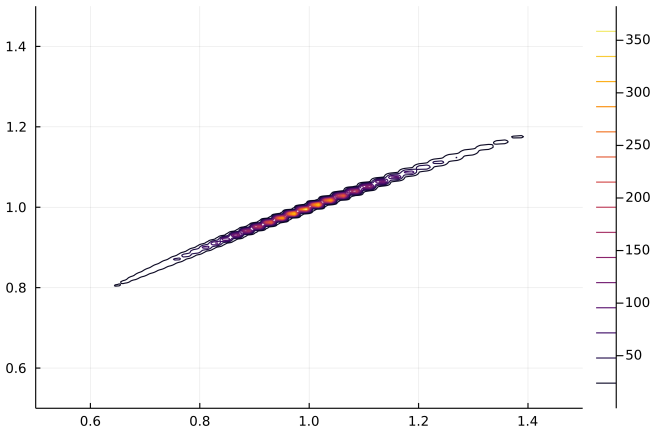

Figure 5: Contour of the inverse Rosenbrock function

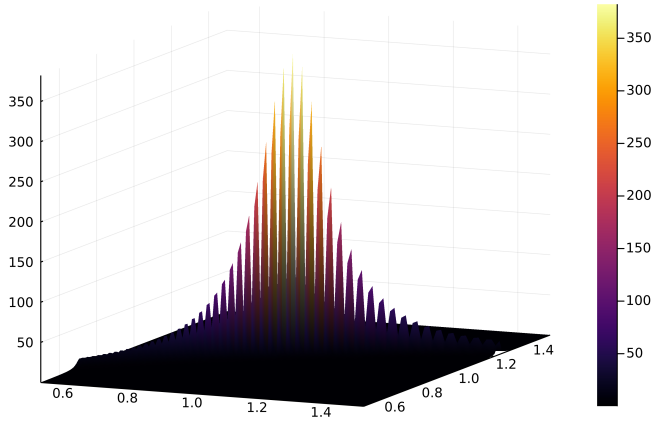

Figure 6: Surface of the inverse Rosenbrock function

B. Debugging code

The intermediate files created during the compilation and linking of the program are required for understanding the mapping between the machine code and source code, this is of utmost importance for using the cuda-gdb and cuda-memcheck tools. For this behaviour to occur the flags -g, -G and -keep have to be set for both during the generation of object files and the linking stage.

C. Disabling GPU timeout

Whenever the profiler was used to obtain information about the code it would take longer than regular execution, this is due to the fact that there's an associated overhead produced by the profiler to obtain and extract the data at strategic points. Many online resources pointed that the solution lied in disabling the Graphical User Interface, however this would not solve the issue (Uninstalling X and setting the execution level to multiuser). The solution instead was found by modifying the device's settings manually (Setting /sys/kernel/debug/gpu.0/timeouts_enabled to N). After manually disabling the setting, if the device's properties are accessed (Through cudaGetDeviceProperties() -> kernelExecTimeoutEnabled) it gives a misleading result, claiming that execution timeout is still enabled.

References

[1] D. Karaboga and B. Basturk, "A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (ABC) algorithm," *J. Glob. Optim.*, vol. 39, no. 3, pp. 459–471, Oct. 2007.

[2] K. Frisch, *Dance language and orientation of bees*. Cambridge: HUP, 1966.