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

Samuel Oreste Abreu - s281568

Abstract

Studies the implementation of the Artificial Bee Colony algorithm in a GPU, various benchmark tests showing how well the algorithm performed.

1 Introduction

The advent of computers came with the ability of performing 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, the code of which can be found in `https://github.com/richter43/GPU-ABC`.

2 Background

2.1 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.

2.2 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, $i \neq k$ (Otherwise the position would not be updated).

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 with random values.
6. Repeat from 3 until the cycles end.

2.3 Parallel implementation

In this parallel implementation every thread is interpreted to be a bee and every block a hive, the choices during the code's development are discussed in the following points:

2.3.1 Setting a bee's task

Knowing that every thread is a bee, they still have to be identified somehow, the proposed solution is that of using an enumerator type and storing them in shared memory.

2.3.2 Obtaining random numbers

There were two possibilites:

- Creating and moving into the device's global memory an array filled with pre-computed random values.
- Computing the random values directly in the device.

Due to the large amount of data that had to be stored in the first case the second option was chosen out of convenience.

2.3.3 Re-assigning task to a scout bee

A bee's scouting task will only last for a single iteration, after which its job will change depending on the solution's fitness, in the event it's better the bee will become employed, otherwise it will become an onlooker.

2.3.4 Multihive computation

As previously stated, a block inside of the GPU corresponds to a hive [3], operating independently from other hives. Their independence is largely due to the impossibility for blocks to exchange information between each other in a synchronized manner.

2.3.5 Crowd wisdom

It's the idea that aggregating the results from many independent sources will result in a consensus closer to the truth [4], this is theorized to be due to the fact that independent actors are susceptible to errors, and such error can be eliminated through the correct aggregation of the multitude of data. This idea only holds given that there's a large, mutually independent population from which to obtain the results from.

2.4 Code implementation

2.4.1 Obtaining random numbers

The library used for obtaining random numbers is cuRAND, which requires its initialization and a seed to instantiate a state, which will then be used to get a stochastic number.

2.4.2 Error detection

A library already present in CUDA's examples was used (`cuda_helper.h`), which checks the return values of various CUDA standard API.

2.5 Aggregating the results

After the hives return their final fitnesses they are aggregated in the following way:

$$sol_{weighted_i} = \sum_{j=0}^{HN} (sol_{ij} \cdot \frac{fit_j}{fit_{total_j}}) \tag{3}$$

Where HN is the number of hives that were instantiated.

2.5.1 Managing potential overflow

There exists a latent possibility for overflow to occur during the aggregation step of the fitnesses, for this reason, an overflow-aware algorithm was implemented in advance, which consists in maintaining the relative ratio information by dividing every fitness by the maximum value among them.

3 Methodology

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

3.1 Benchmark functions

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

3.1.1 Rastrigin

$$f(\mathbf{x}) = 10n + \sum_{i=1}^{n} (x_i^2 - 10cos(2\pi x_i)) \tag{4}$$

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.

3.1.2 Sphere

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

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.

3.1.3 Rosenbrock

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} (100(x_{i+1} - x_i^2)^2 + (1 - x_i^2)) \tag{6}$$

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.

3.2 Mapping the functions

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

$$fitness_{max} = \frac{1}{fitness_{min}} \tag{7}$$

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

3.3 Error measurement

Measured in mean squared error.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (x_i - \hat{x_i}) \tag{8}$$

Where n is the dimension of the solution.

4 Results

The baseline implementation has the following parameters:

- 8 blocks.
- 128 threads.
- 3 maximum patience.
- -3.0 minimum float.
- 3.0 maximum float.
- 0.5 onlooker to employed ratio.
- 0 seed for initializing the random states.
- 2 solution dimension.

4.1 Performance - timing

Table 1: Amount of iterations vs the main kernel execution time.

Iterations	Rastrigin	Spheric	Rosenbrock
8	3.2 ms	3 ms	3.03 ms
64	24.2 ms	22 ms	23.74 ms
256	96.83 ms	87 ms	94.30 ms
1024	387.38 ms	351.45 ms	377.02 ms

4.2 Performance - error

Table 2: Amount of iterations versus the obtained result.

Iterations	Rastrigin	Spheric	Rosenbrock
8	$(0.106, 8.13 \cdot 10^{-2})$	$(1.3 \cdot 10^{-2}, -5.87 \cdot 10^{-3})$	(0.939,0.906)
64	$(2.24 \cdot 10^{-2}, 1.86 \cdot 10^{-3})$	$(1.97 \cdot 10^{-3}, 5.63 \cdot 10^{-3})$	(1.011,1.022)
256	$(-1.82 \cdot 10^{-3}, 2.88 \cdot 10^{-3})$	$(1.52 \cdot 10^{-3}, -8.16 \cdot 10^{-4})$	(1.001018,1.002216)
1024	$(-3.3 \cdot 10^{-4}, 5.83 \cdot 10^{-3})$	$(9.76 \cdot 10^{-4}, 4.19 \cdot 10^{-4})$	(1.000574,1.000933)

Table 3: Amount of iterations versus error.

Iterations	Rastrigin	Spheric	Rosenbrock
8	$9 \cdot 10^{-3}$	$1.1 \cdot 10^{-4}$	$6.15 \cdot 10^{-3}$
64	$2 \cdot 10^{-4}$	$1.77 \cdot 10^{-5}$	$3.24 \cdot 10^{-4}$

Iterations	Rastrigin	Spheric	Rosenbrock
256	$5.81 \cdot 10^{-5}$	$1.48 \cdot 10^{-6}$	$2.97 \cdot 10^{-6}$
1024	$1.70 \cdot 10^{-5}$	$5.64 \cdot 10^{-7}$	$5.99 \cdot 10^{-7}$

Table 4: Amount of threads vs error.

Threads	Rastrigin	Spheric	Rosenbrock
32	$7.4 \cdot 10^{-3}$	$1.1 \cdot 10^{-3}$	$1.5 \cdot 10^{-5}$
64	$2.32 \cdot 10^{-6}$	$4.01 \cdot 10^{-4}$	$7.6 \cdot 10^{-6}$
128	$5.93 \cdot 10^{-6}$	$2.19 \cdot 10^{-6}$	$1.03 \cdot 10^{-6}$

5 Discussion

5.1 Execution time

The main factor that affects the amount of time the kernel is running is the amount of iterations, this effect can be seen in tbl. 1.

5.2 Correctness of the algorithm

The bees in a hive are correctly gathering around optimum points, as can be seen in fig. 7.

Given the algorithm is running for long enough the solution will converge in the global minimum, as can be seen between fig. 7 and fig. 8.

The algorithm succesfully managed to find the global optimum as can be seen in fig. 8, fig. 10 and fig. 11.

In summary, for obtaining the best results there needs to be a combination of:

- Large enough amount of hives (More than 4 blocks).
- Large enough amount of bees (More than 32 threads).
- Large enough amount of iterations (More than 32 iterations).

6 Ablation

6.1 Storing the passed struct in constant memory

There was no noticeable difference in performance.

6.2 Using shared memory for storing fitness information

It consistently performs 1.2 times longer than its global memory counterpart, the reason for this is still unknown, considering that the shared memory is being

accessed in a sequential mode, which entails that there will not be any bank conflict.

7 Conclusion

Through the usage of GPUs lots of computations can be performed in parallel, effectively accelerating the computational time. One of the potential applications are relayed in this study.

The Artificial Bee Colony algorithm is found to be a prime candidate for exploiting the Single Instruction Multiple Data model due to the need of large amount of computations to be performed.

8 Future work

It's recommended to study further optimizations on the algorithm's workflow.

9 Figures

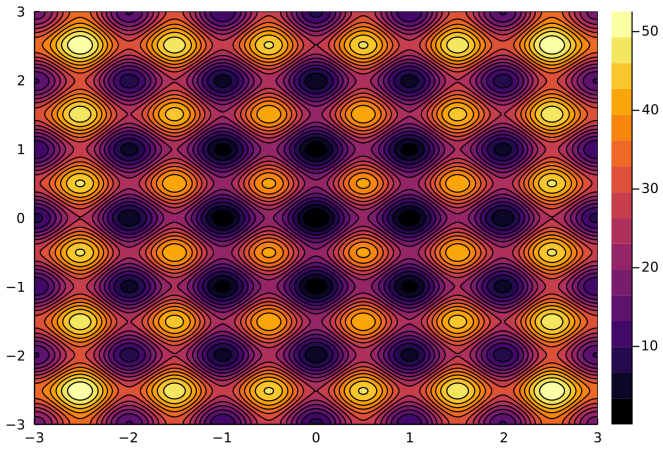

Figure 1: Contour of the Rastrigin function

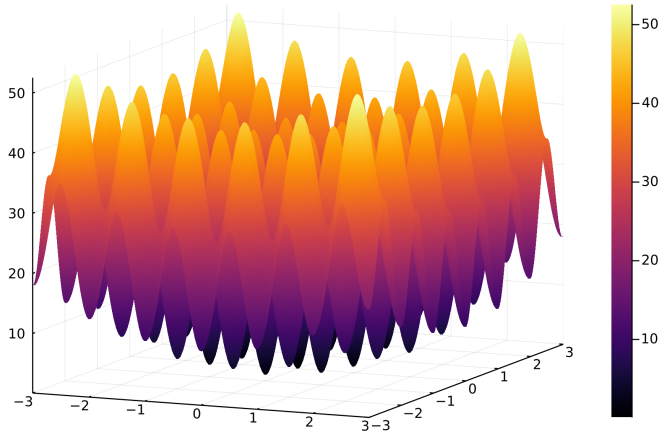

Figure 2: Surface of the Rastrigin function

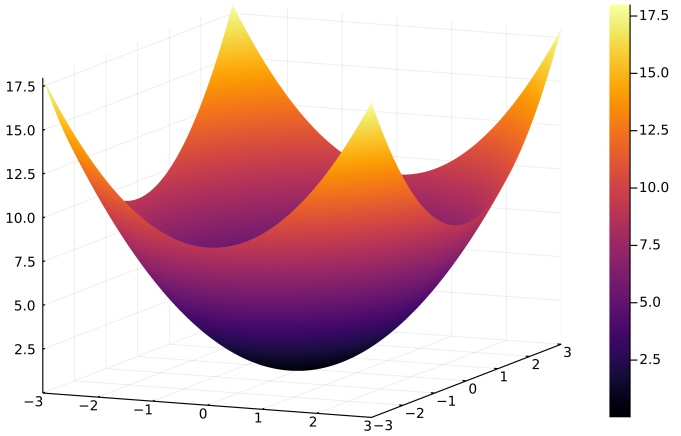

Figure 3: Surface of the spheric function

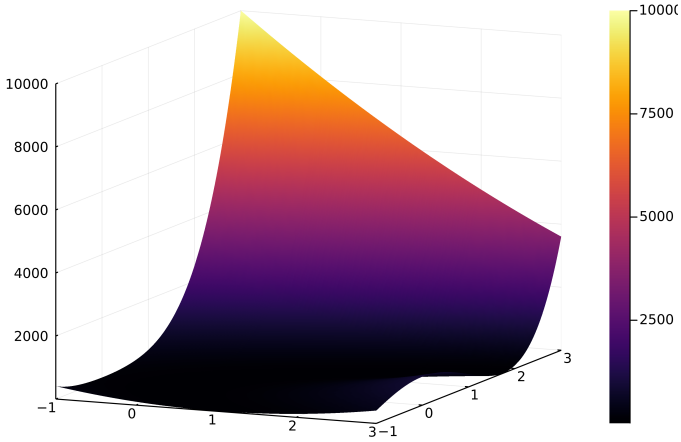

Figure 4: Surface of the Rosenbrock function

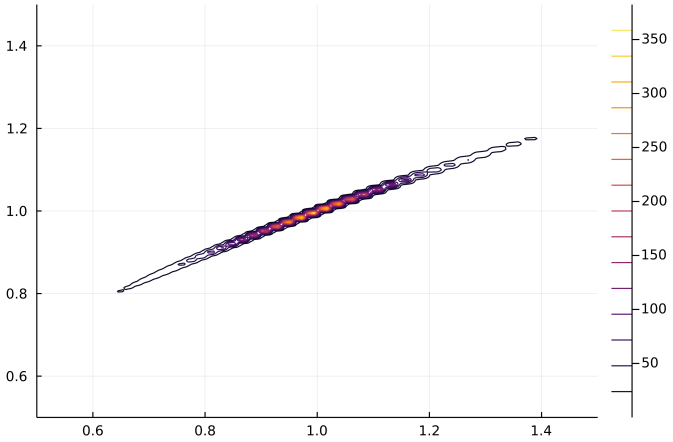

Figure 5: Contour of the inverse Rosenbrock function

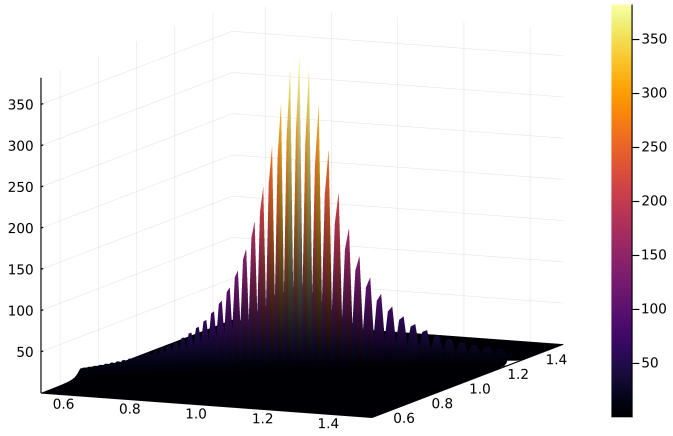

Figure 6: Surface of the inverse Rosenbrock function

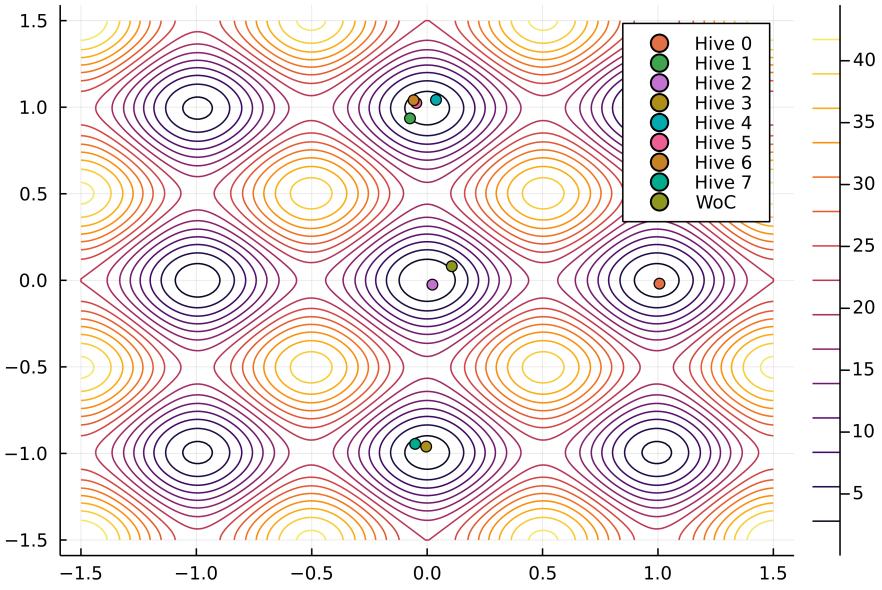

Figure 7: Solutions of Rastrigin after 8 iterations

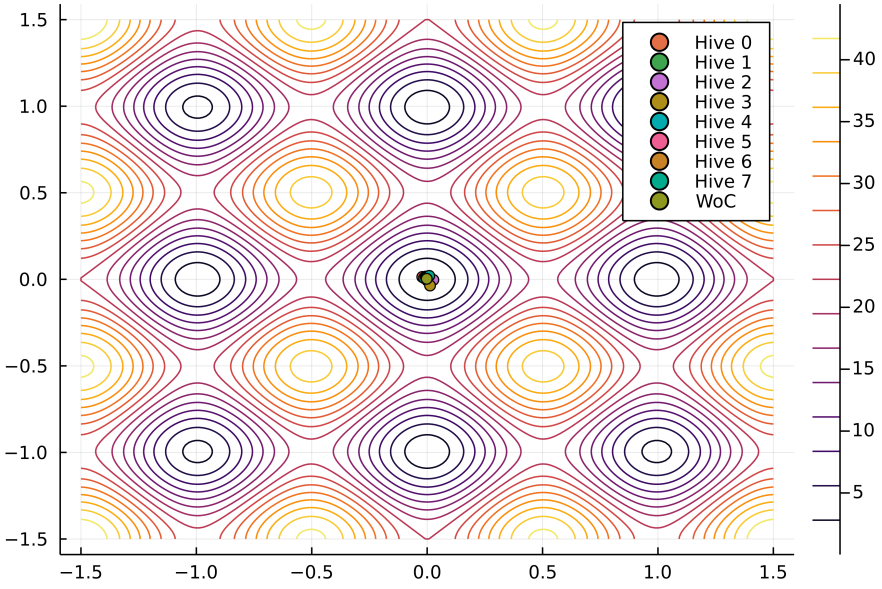

Figure 8: Solutions of Rastrigin after 256 iterations

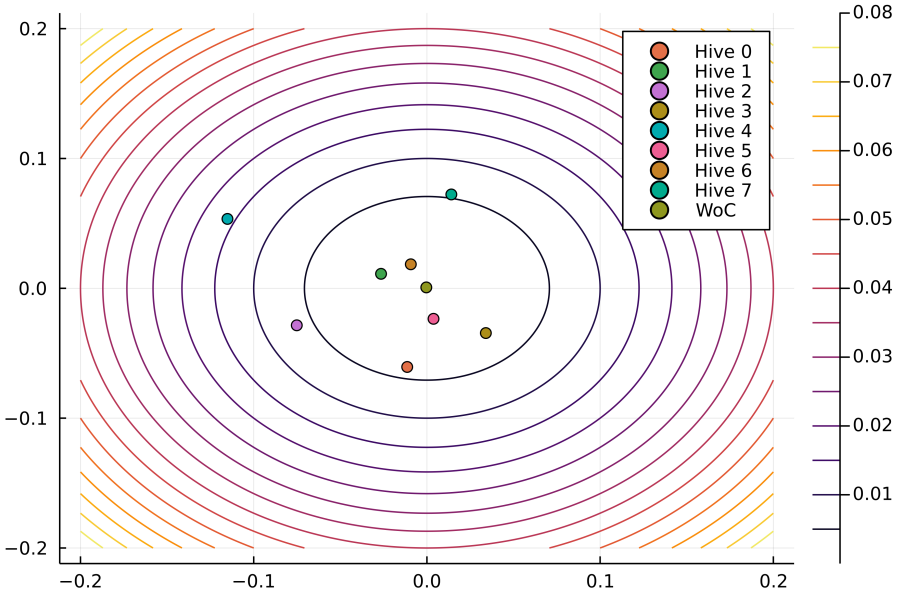

Figure 9: Solutions of the spheric function after 8 iterations

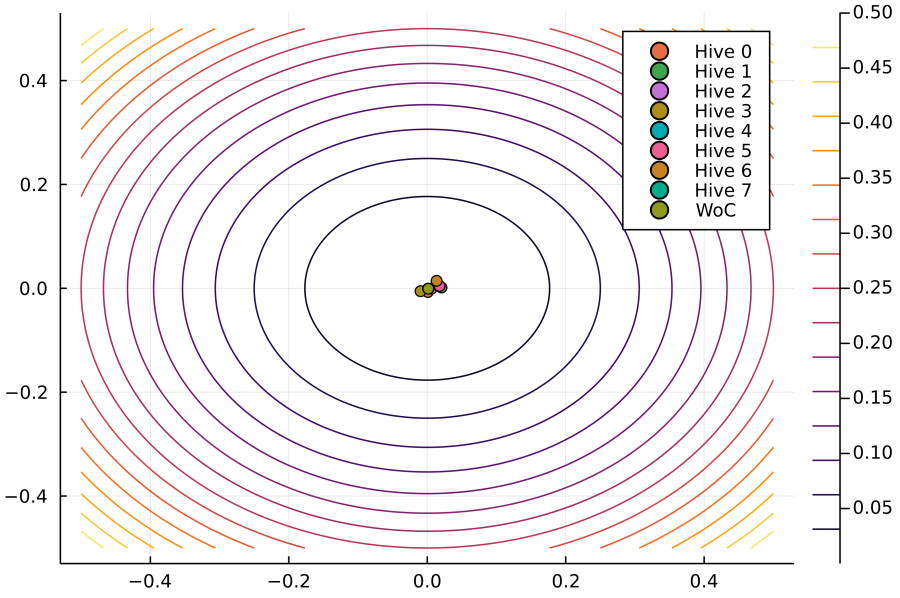

Figure 10: Solutions of the spheric function after 256 iterations

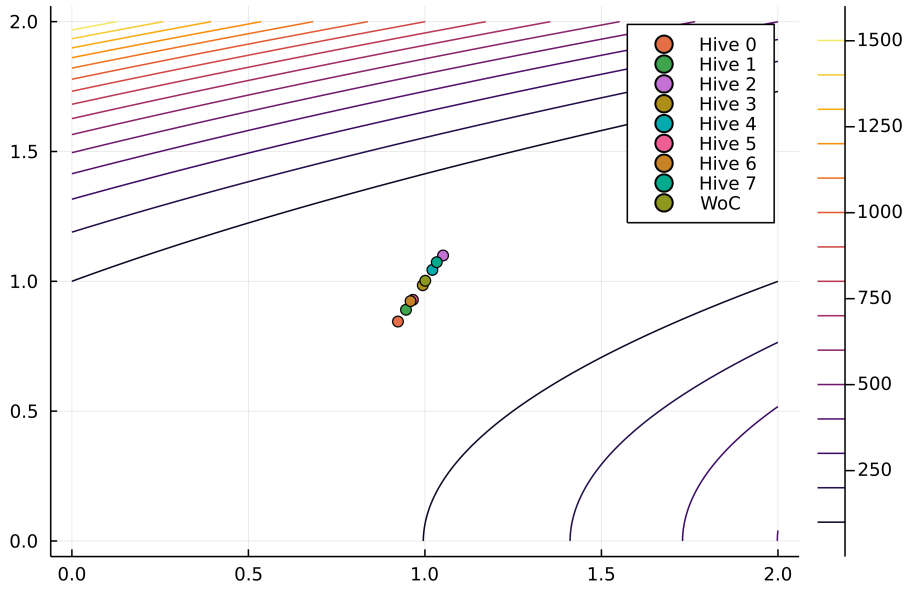

Figure 11: Solutions of Rosenbrock function after 256 iterations

10 Further comments

10.1 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 then be linked in a future step once all of the object files are generated.

10.2 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 succesfully doing so the flags -g, -G and -keep have to be set both during the generation of object files and the linking stage.

Another factor that affected the correct debugging of the program is the fact that, by default, the created user has no permissions to access the debugging intermediary device (Such device can be found in `/dev/nvhost-dbg-gpu`).

10.3 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 it to obtain and extract the statistics at strategic points. Many online resources pointed that the solution lied in disabling the Graphical User Interface (Uninstalling X and setting the execution level to multiuser), however this would not solve the issue. 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.

10.4 Issues with nvprof

Whenever the profiling had to be performed on a program that featured shared fitness memory, the nvprof utility would not be able to execute or gather metrics/events, citing a `cudaErrorLaunchFailure` as its root cause, despite the fact that the program ran succesfully without the usage of nvprof. Neither cuda-gdb nor cuda-memcheck were useful for pinpointing the issue. statistics correctly. No solution to this issue was found.

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.

[3] R. S. Parpinelli, C. M. V. Benitez, and H. S. Lopes, "Parallel approaches for the artificial bee colony algorithm," in *Adaptation, learning, and optimization*, Springer Berlin Heidelberg, 2011, pp. 329–345.

[4] Aristoteles and V. G. Manuela, "Libro III - la soberania de los pueblos," in *Politica*, Gredos, 1988, p. 180.