Firefly + BFGS algorithm with openMP and Cuda project for Advanced Methods for Scientific Computing

Javed Abdullah 275266 - Group 2

a.y 2024/2025

Indice

1 Introduction

This project is an experimental attempt to combine the Firefly Algorithm (FA) [2] with the BFGS method [1]. FA is good at exploring the global search space, while BFGS converges fast when near a local minimum. By combining them, the idea is to use FA to find promising areas and BFGS to refine the best solution.

The implementation is written in C++ and uses OpenMP for CPU parallelism and CUDA for GPU acceleration. The goal is to test if this hybrid approach leads to better results and faster convergence compared to using either method alone.

1.1 Firefly Algorithm

The Firefly Algorithm (FA) is a swarm-based metaheuristic inspired by the flashing behavior of fireflies. In this implementation, each firefly represents a potential solution in a D-dimensional space, encoded as a `std::vector<double>`. The brightness of a firefly is determined by the objective function value at its position.

The main steps of the algorithm are as follows:

1. **Initialization:** A population of fireflies is randomly initialized within the given bounds. Each firefly stores its position and brightness.
2. **Evaluation:** The brightness of each firefly is computed using the objective function.
3. **Movement:** For every pair of fireflies (i, j), if firefly j is brighter than i, then i is attracted toward j by the formula:

   $$x_i \leftarrow x_i + \beta_0 \cdot e^{-\gamma r_{ij}^2} \cdot (x_j - x_i) + \alpha \cdot \text{rand}(-1, 1)$$

   where:

   - r_{ij} is the Euclidean distance between fireflies i and j
   - β_0 is the attractiveness constant
   - γ is the light absorption coefficient
   - α controls the randomness
4. **Clipping:** After each move, the firefly's position is clipped to stay within the search bounds.
5. **Iteration:** Steps 2–4 are repeated for a fixed number of iterations.

Key C++ implementation: Firefly Algorithm

The following C++ snippet shows the core of the firefly update loop, where each firefly moves toward brighter ones:

```
for (int i = 0; i < numFireflies; ++i) {
    for (int j = 0; j < numFireflies; ++j) {
        if (fireflies[j].getBrightness() < fireflies[i].getBrightness()) {
            double r = euclideanDistance(fireflies[i], fireflies[j]);
            double beta_effective = beta * exp(-gamma * r * r);

            std::vector<double> newPosition = fireflies[i].getPosition();
            std::vector<double> targetPosition = fireflies[j].getPosition();

            for (int d = 0; d < dimensions; ++d) {
                newPosition[d] += beta_effective * (targetPosition[d] -
                    newPosition[d])
                               + alpha * randomNoise(gen);
            }

            // Clipping
            for (int d = 0; d < dimensions; ++d) {
                newPosition[d] = std::max(lower_bound, std::min(upper_bound,
                    newPosition[d]));
            }

            fireflies[i].setPosition(newPosition);
        }
    }
}
```

Listing 1: Firefly position update loop

1.2 BFGS Method

The BFGS algorithm (Broyden–Fletcher–Goldfarb–Shanno) is a quasi-Newton method for unconstrained optimization. It aims to find a local minimum of a differentiable scalar function by iteratively updating an estimate of the inverse Hessian matrix, avoiding the need to compute second derivatives directly.

In this implementation, the BFGS method is used to refine the best solution found by the Firefly Algorithm. The optimization proceeds as follows:

1. **Initialization:** The algorithm starts from an initial guess x_0 (a vector in R^D) and uses the identity matrix as the initial inverse Hessian approximation H.

2. **Numerical Gradient:** The gradient of the objective function is approximated using finite differences. Given a step size ϵ, the i-th component of the gradient is:
$$\frac{\partial f}{\partial x_i} \approx \frac{f(x+\epsilon e_i) - f(x)}{\epsilon}$$
where e_i is the i-th unit vector.

3. **Update Rule:** At each iteration, the search direction is computed as $p_k = -H_k \nabla f_k$, and the point is updated as $x_{k+1} = x_k + \alpha p_k$, with a fixed step size α (e.g., 0.01).

4. **Hessian Update:** The inverse Hessian approximation is updated using:

$$H_{k+1} = H_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{H_k s_k s_k^T H_k}{s_k^T H_k s_k}$$

   where $s_k = x_{k+1} - x_k$ and $y_k = \nabla f_{k+1} - \nabla f_k$.

5. **Stopping Criteria:** The iterations stop when the norm of the gradient is below a small threshold or the maximum number of iterations is reached.

The method does not require the exact Hessian matrix, which makes it efficient in practice, especially when the function is expensive to evaluate or when second derivatives are not available. The use of ‘std::vector¡double¿‘ and ‘Eigen::MatrixXd‘ allows for easy and efficient numerical operations in C++.

Key C++ implementation: BFGS Optimizer

This code shows the main update step in the BFGS method:

```cpp
std::vector<double> BFGSOptimizer::optimize(const std::vector<double>& x0, int
     maxIterations) {
    Eigen::VectorXd x = Eigen::Map<const Eigen::VectorXd>(x0.data(), x0.size());
    //hessian matrix aproximation
    Eigen::MatrixXd H = Eigen::MatrixXd::Identity(dim, dim);

    for (int iter = 0; iter < maxIterations; ++iter) {
        Eigen::VectorXd grad = numericalGradient(x);
        if (grad.norm() < 1e-6) break;

        Eigen::VectorXd p = -H * grad; //direction for (discesa)
        double step = 1e-1; // step a = 0.1 (basic step)

        Eigen::VectorXd x_new = x + step * p; // move to the p direction
        Eigen::VectorXd grad_new = numericalGradient(x_new);

        Eigen::VectorXd s = x_new - x;
        Eigen::VectorXd y = grad_new - grad;

        double sy = s.dot(y);

        //for avoid huge division and calculate new H
        if (sy > 1e-10) {
            Eigen::MatrixXd I = Eigen::MatrixXd::Identity(dim, dim);
            H = (I - s * y.transpose() / sy) * H * (I - y * s.transpose() / sy) +
                 (s * s.transpose()) / sy;
        }

        x = x_new;
    }

    std::vector<double> result(x.data(), x.data() + x.size());
    return result;
}
```

Listing 2: BFGS update and direction computation

1.3 Test Functions

In this work, we evaluate the hybrid optimization algorithm on three standard benchmark functions:

- **Sphere Function:** A simple convex function used to test convergence.
- **Rastrigin Function:** A non-convex function with many local minima.
- **Rosenbrock Function:** A classical test problem with a curved, narrow valley.

Below are the plots of these functions in 3D:

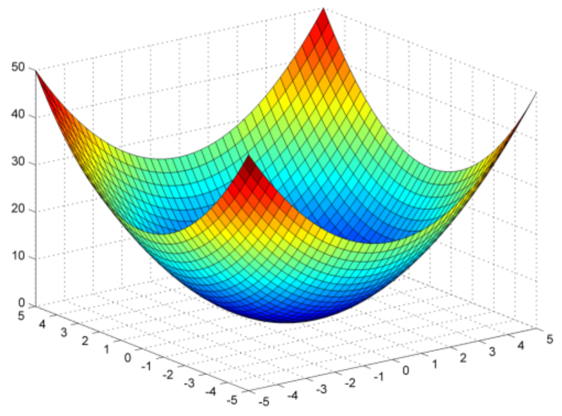

Figura 1: Sphere function. Source: `http://www.sfu.ca/~ssurjano/spheref.html`

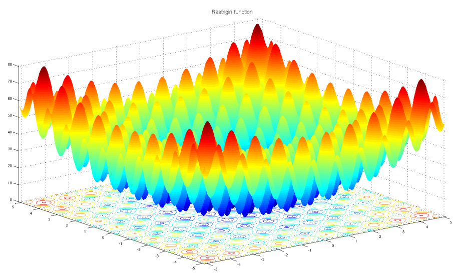

Figura 2: Rastrigin function. Source: `https://en.wikipedia.org/wiki/Rastrigin_function`

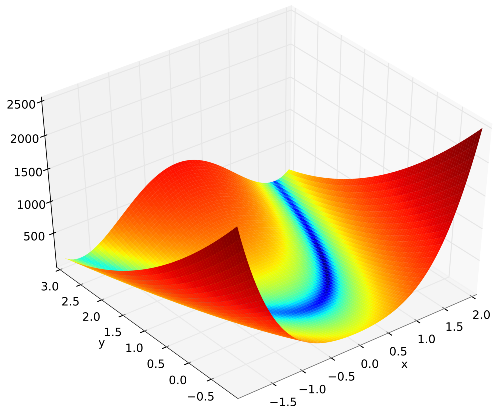

Figura 3: Rosenbrock function. Source: `https://it.m.wikipedia.org/wiki/File:Rosenbrock_function.svg`

2 Experiments and Results

2.1 Experimental Setup

All tests were performed on a laptop running Ubuntu Linux with the following specifications:

- CPU: Intel Core i5-8250U (4 cores / 8 threads)
- RAM: 8 GB DDR4 @ 1.6 GHz
- OS: Ubuntu 24.10 LTS

The algorithm was compiled with and executed using the following configuration:

```
const int trials = 30;
const int dimensions = 2;
const int numFireflies = 40;
const int maxIterations = 100;
const double alpha = 0.5, beta = 0.2, gamma = 1.0;
```

Listing 3: Firefly Algorithm configuration parameters

2.2 Firefly Algorithm Parameters

In the Firefly Algorithm, the parameters α, β, and γ play distinct roles:

- α **(randomness factor):** Controls the strength of the random walk. Higher values mean more exploration, especially useful in early iterations.

- β **(attractiveness):** Determines how strongly a firefly is attracted to brighter ones. A higher β leads to faster convergence.
- γ **(absorption coefficient):** Regulates how quickly the attractiveness decreases with distance. A small γ means fireflies are affected even at longer distances.

These parameters were chosen empirically to balance exploration and exploitation.

2.3 Benchmark Functions

The Firefly Algorithm was tested on three standard benchmark functions:

- Sphere
- Rastrigin
- Rosenbrock

The version tested here is a basic "raw" Firefly Algorithm implementation, without hybridization or parallelism.

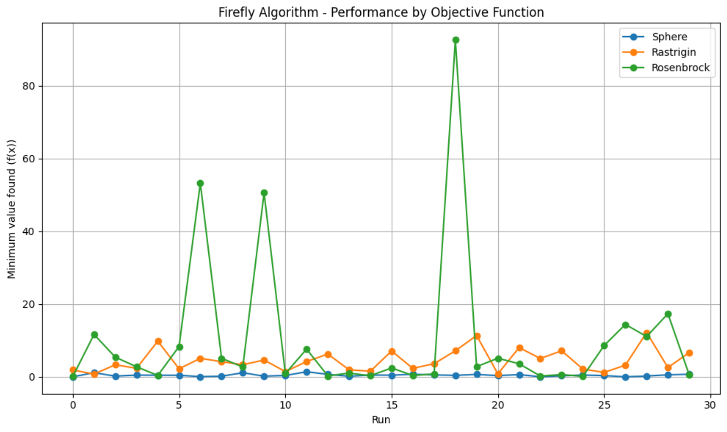

Figura 4: Performance of the Firefly Algorithm on benchmark functions (averaged over 30 trials).

2.4 Result Analysis

Axes Description:

- **X-axis (Run):** Each point represents one of the 30 independent executions of the algorithm.

- **Y-axis ($f(x)$):** Best objective value found in each run (lower is better).

Function-wise analysis:

- **Sphere:**
  - All values are close to zero across runs.
  - No significant outliers.
  - High stability and consistent convergence.
  - *Conclusion:* Firefly performs excellently on simple convex landscapes.
- **Rastrigin:**
  - Some variability, but within a controlled range.
  - A few worse runs, yet no extreme outliers.
  - *Conclusion:* Firefly manages to escape many local minima, though it doesn't consistently approach zero.
  - *Observation:* Adding BFGS could improve in later stages.
- **Rosenbrock:**
  - Very high variance
  - Some runs are near-optimal, others are far off.
  - General instability in performance.
  - *Conclusion:* The algorithm occasionally finds the narrow valley of the Rosenbrock function, but often fails to converge.
  - *Suggestion:* This motivates the use of local optimization (e.g., BFGS), parameter tuning, or restart strategies.

3 OpenMP Optimization

To accelerate the execution on CPU, two main parts of the Firefly Algorithm were parallelized using OpenMP: the initialization of the fireflies and the evaluation of their brightness during each iteration.

1. Initialization of Fireflies

The function `initializeFireflies()` generates initial random positions for each firefly in the population. Since each firefly is independent, we parallelized the loop with `#pragma omp parallel for`.

```cpp
void FireflyAlgorithm::initializeFireflies() {
    fireflies.clear();
    fireflies.resize(numFireflies, Firefly(dimensions));

    std::uniform_real_distribution<double> dist(lower_bound, upper_bound);

#pragma omp parallel for
    for (int i = 0; i < numFireflies; ++i) {
        std::mt19937 thread_rng(seed + i); // unique seed per thread
        std::vector<double> pos(dimensions);
        for (int d = 0; d < dimensions; ++d) {
            pos[d] = dist(thread_rng);
        }
        fireflies[i] = Firefly(dimensions);
        fireflies[i].setPosition(pos);
        fireflies[i].setBrightness(std::numeric_limits<double>::max());
    }
}
```

Listing 4: Parallelized firefly initialization

Each thread generates a unique random number stream using a different seed (`seed + i`), which ensures reproducibility and avoids race conditions.

2. Parallel Evaluation of Brightness

Inside the main optimization loop (`optimize()`), we parallelized the evaluation of brightness using a parallel `for` loop, since each firefly is evaluated independently.

```cpp
for (int iter = 0; iter < maxIterations; ++iter) {

#pragma omp parallel for
    for (int i = 0; i < fireflies.size(); ++i) {
        fireflies[i].setBrightness(objectiveFunction(fireflies[i].getPosition()))
            ;
    }

    updateFireflies(); // not parallelized due to shared state
}
```

Listing 5: Parallel fitness evaluation loop

This allows evaluating the fitness of all fireflies in parallel, significantly speeding up execution when the objective function is expensive to compute.

Parallelization Notes

- The movement phase (`updateFireflies()`) is not parallelized here to avoid race conditions, since fireflies influence each other's positions.
- Parallelization is effective because each firefly is updated independently during initialization and fitness evaluation.

- The usage of OpenMP makes the CPU version much faster on average, especially for large populations or expensive functions.

Execution Time Comparison with and without OpenMP

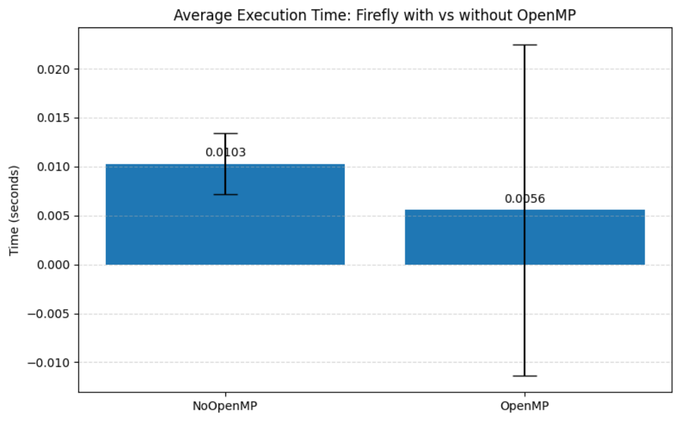

Figura 5: Average execution time of Firefly Algorithm with and without OpenMP.

As shown in Figure 5, the average execution time using OpenMP is lower than the serial version. However, the standard deviation is very high, indicating potential overhead due to thread contention and scheduling. This was observed when OpenMP was run without limiting the number of threads.

Optimizing Thread Count

To address the instability caused by excessive threading, the algorithm was executed with a fixed number of threads:

```
OMP_NUM_THREADS=4 ./firefly
```

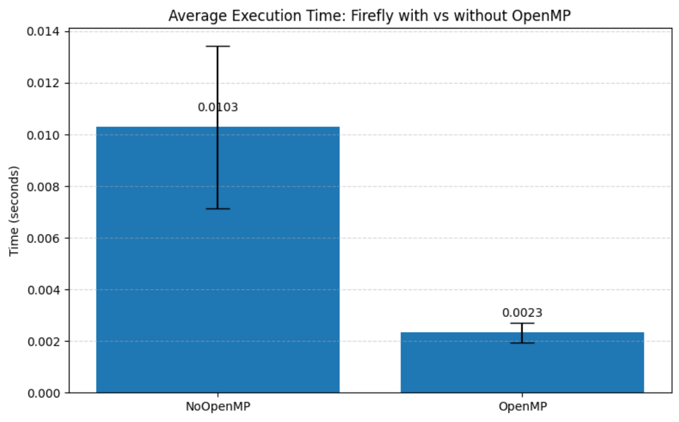

Figura 6: Execution time comparison using 4 OpenMP threads.

As shown in Figure 6, limiting OpenMP to 4 threads resulted in both lower execution time and significantly reduced variance. This demonstrates that choosing an appropriate number of threads leads to better CPU utilization and more stable performance. In this case, OpenMP with 4 threads achieved a speed-up of more than 4× compared to the serial version, with minimal overhead.

4 Hybrid Optimization: Firefly + BFGS

To enhance solution accuracy, especially for functions with smooth, well-conditioned landscapes, a hybrid approach was implemented by combining the global exploration of the Firefly Algorithm with the local refinement of the BFGS optimizer.

The idea is simple: use the best solution found by Firefly as the initial guess for the BFGS optimizer.

```cpp
FireflyAlgorithm optimizerCPU(numFireflies, dimensions, alpha, beta, gamma);
optimizerCPU.setObjectiveFunction(finfo.func);

std::vector<double> bestCPU = optimizerCPU.optimize(maxIterations);
double valueCPU = finfo.func(bestCPU);

BFGSOptimizer bfgsCPU(dimensions);
bfgsCPU.setObjective(finfo.func);

std::vector<double> refinedCPU = bfgsCPU.optimize(bestCPU, maxIterations);
double refinedValueCPU = finfo.func(refinedCPU);
```

Listing 6: Hybrid optimization: Firefly followed by BFGS

Results on Sphere Function

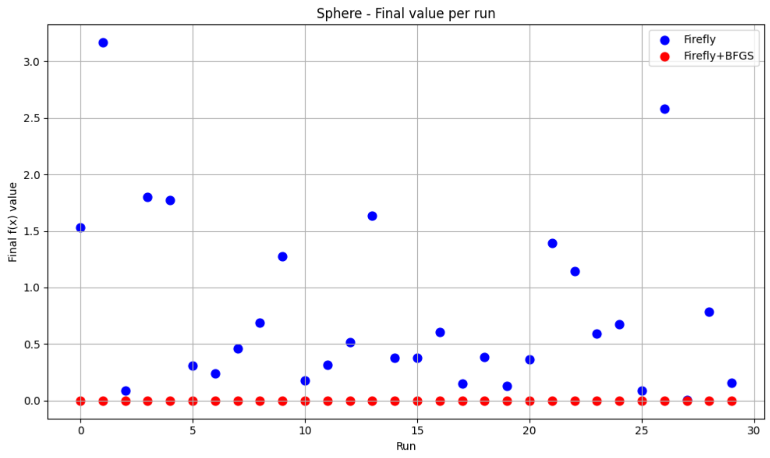

Figura 7: Final $f(x)$ values per run on Sphere function: Firefly vs Firefly+BFGS.

Figure 7 shows a consistent improvement when BFGS is applied after Firefly. The red dots (Firefly+BFGS) are almost all at the theoretical minimum (≈ 0), while the blue ones (Firefly alone) show a wider spread.

Interpretation: On the Sphere function, BFGS effectively refines the Firefly result, especially when Firefly gets close but not exactly to the minimum.

Results on Rosenbrock Function

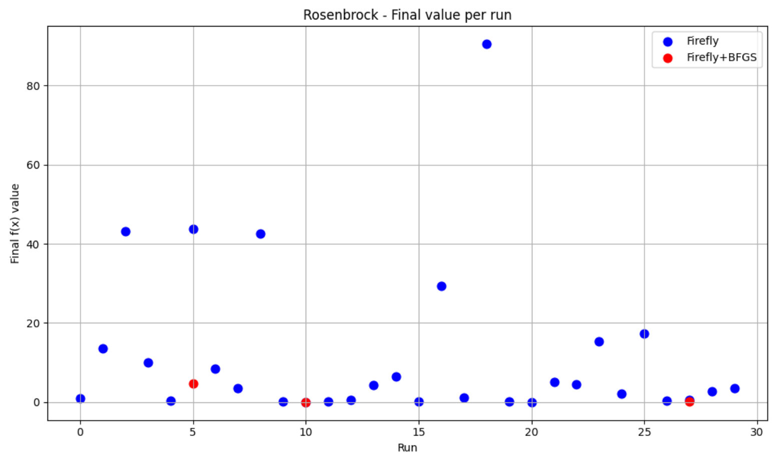

Figura 8: Final $f(x)$ values per run on Rosenbrock function: Firefly vs Firefly+BFGS.

Figure 8 illustrates a more complex scenario. In some runs, the BFGS optimizer (red) significantly improves the result over Firefly alone. In others, the Firefly result was already very good, or BFGS did not help much.

Interpretation: Due to the narrow curved valley of Rosenbrock, local refinement via BFGS can be beneficial — but only if the Firefly solution is already close enough to the valley.

When is the Hybrid Strategy Useful?

- On convex, smooth functions like Sphere, BFGS significantly improves solution accuracy.
- On difficult functions with many local minima (e.g., Rastrigin, not shown here), the benefit is limited unless Firefly gets close to a global basin.
- The hybrid approach combines robustness (Firefly) with precision (BFGS), with minimal computational overhead.

5 GPU Optimization with CUDA

To exploit GPU parallelism, a CUDA version of the Firefly Algorithm was developed. The class `FireflyAlgorithm_Cuda` inherits from `FireflyAlgorithm` and overrides key computational steps using CUDA kernels.

Class Design

The constructor simply forwards parameters to the base class:

```
FireflyAlgorithm_Cuda::FireflyAlgorithm_Cuda(
    int nFireflies, int dim, double a, double b, double g,
    double lower, double upper, size_t s
) : FireflyAlgorithm(nFireflies, dim, a, b, g, lower, upper, s) {
    // nothing else needed here
}
```

Listing 7: CUDA Firefly class constructor

The two main steps ported to GPU are:

- Evaluation of the objective function (brightness).
- Update of firefly positions.

These are launched via two wrapper functions:

```
extern "C" void launchUpdateFirefliesCUDA(
    double* d_positions, double* d_brightness,
    int numFireflies, int dimensions,
    double alpha, double beta, double gamma,
    unsigned int seed,
    double lower_bound, double upper_bound,
    int threadsPerBlock
);

extern "C" void launchEvaluateBrightnessCUDA(
    double* d_positions, double* d_brightness,
    int numFireflies, int dimensions, int objectiveType,
    int threadsPerBlock
);
```

Listing 8: CUDA wrappers for evaluation and update

CUDA Kernels

1. Brightness Evaluation Kernel

Each thread computes the objective function for a single firefly:

```
__global__ void evaluateBrightnessCUDA(
    double* positions, double* brightness,
    int numFireflies, int dimensions, int objectiveType
) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= numFireflies) return;
    brightness[i] = evalObjective(&positions[i * dimensions], dimensions,
         objectiveType);
}
```

Listing 9: CUDA kernel for brightness evaluation

2. Firefly Position Update (with Shared Memory)

The most computationally intensive part, updating each firefly's position, was parallelized on the GPU using a CUDA kernel that loads other fireflies into shared memory in tiles.

Rather than comparing every firefly pair globally from global memory (which is slow), we copy blocks of fireflies into shared memory and iterate on them locally. The firefly at thread i is updated using the standard attraction formula, but now computed using the faster shared memory.

High-level steps:

- Each thread i represents a firefly and copies its own position to the output buffer.
- A block of j fireflies is loaded into shared memory (positions and brightness).
- Thread i compares itself to each j in the block to determine attraction.
- If j is brighter, i is attracted and updates its position accordingly.
- After all updates, a clipping operation ensures bounds are respected.

Note: Shared memory size is limited per block. To avoid out-of-bounds errors, we added a check on the host side.

Host-side wrapper: memory allocation and shared memory size control

Before launching the kernel, the host code computes the needed shared memory size and verifies it does not exceed the hardware limit:

```
int blocksPerGrid = (numFireflies + threadsPerBlock - 1) / threadsPerBlock;
size_t sharedMemSize = threadsPerBlock * (dimensions * sizeof(double) + sizeof(
     double));

// Check device limit
int device;
cudaGetDevice(&device);
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, device);

if (sharedMemSize > prop.sharedMemPerBlock) {
    // Optional: raise error or adjust
    sharedMemSize = prop.sharedMemPerBlock;
    // Warning: May lead to incorrect behavior if too little memory is available
}

// Allocate output buffer
double* d_new_positions;
cudaMalloc(&d_new_positions, numFireflies * dimensions * sizeof(double));

// Launch kernel
```

```
updateFirefliesCUDA_shared<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(
    d_positions, d_brightness,
    numFireflies, dimensions,
    alpha, beta, gamma,
    seed,
    lower_bound, upper_bound,
    d_new_positions
);
```

Listing 10: Host-callable wrapper: memory and safety check

This logic prevents kernel crashes due to excessive shared memory usage. If the memory requested exceeds the per-block device limit, it is reduced to the maximum allowed. However, the kernel may behave incorrectly if shared memory is not sufficient for the tile size.

6 Performance Comparison: CPU vs GPU

Although the GPU implementation is significantly more parallelized, the performance benefit is not always visible on small-scale problems.

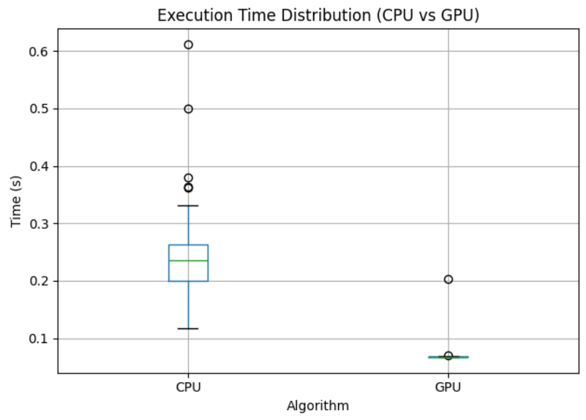

Figura 9: Execution time distribution over 30 runs (CPU vs GPU).

As shown in Figure 9, the average execution time of the GPU version is lower than the CPU version. However, the difference is not always drastic for small problems. The CPU also shows greater variability, with some runs taking up to 0.6 seconds.

Interpretation: The GPU version is more stable and predictable in terms of time. This is expected due to the dedicated parallelism of the GPU and its ability to handle uniform, repetitive tasks efficiently.

Solution Quality: CPU vs GPU on Sphere

In addition to execution time, we compared the quality of the solutions obtained on the Sphere function.

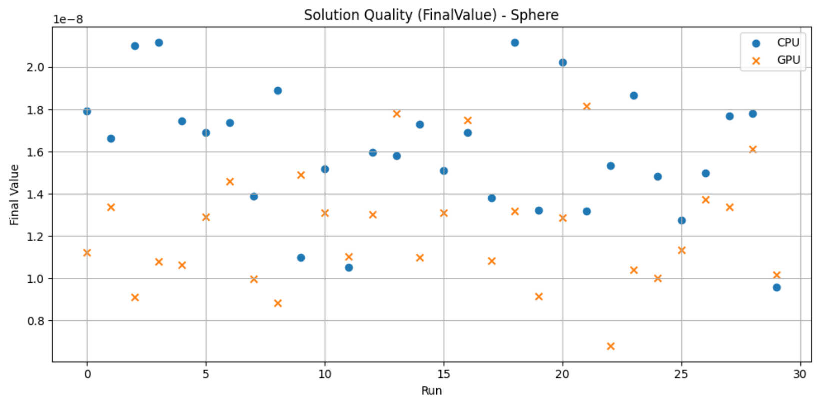

Figura 10: Final solution values over 30 runs on Sphere function (CPU vs GPU).

As shown in Figure 11, the GPU version tends to reach slightly better solutions than the CPU version in most runs.

Observation:

- Although the difference is not drastic, the GPU implementation consistently obtains lower final values.
- This could be due to better numerical consistency and load balancing in parallel evaluations.
- Both implementations still converge to very small values (10^{-8}), showing overall robustness.

Overall, the GPU is not only faster in high-dimensional cases, but also occasionally yields better optimization results.

6.1 Limitations: Fitness Evaluation on CPU (GPU-CPUfit)

During development, a trade-off was required in order to support arbitrary objective functions (i.e., passed as higher-order functions). To ensure full flexibility, the decision was made to compute the fitness **on the CPU**, even though the firefly positions were being updated on the GPU. This change implied disabling the `evaluateBrightnessCUDA` kernel and replacing it with CPU-side evaluation, as shown in the following code snippet:

```
#pragma omp parallel for
for (int i = 0; i < numFireflies; ++i) {
    std::vector<double> fireflyPos(dimensions);
    for (int d = 0; d < dimensions; ++d)
        fireflyPos[d] = h_positions[i * dimensions + d];
    h_brightness[i] = objectiveFunctionCPU(fireflyPos);
}
```

Listing 11: Fitness evaluation on CPU

This introduced a performance bottleneck, as the GPU was no longer responsible for the most computationally expensive task: evaluating the objective function. Nevertheless, this compromise enabled greater generality and code reuse.

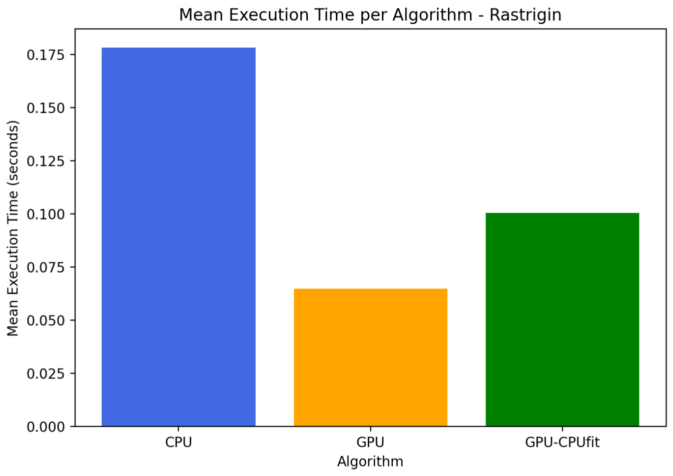

Figura 11: Mean execution time — Rastrigin function.

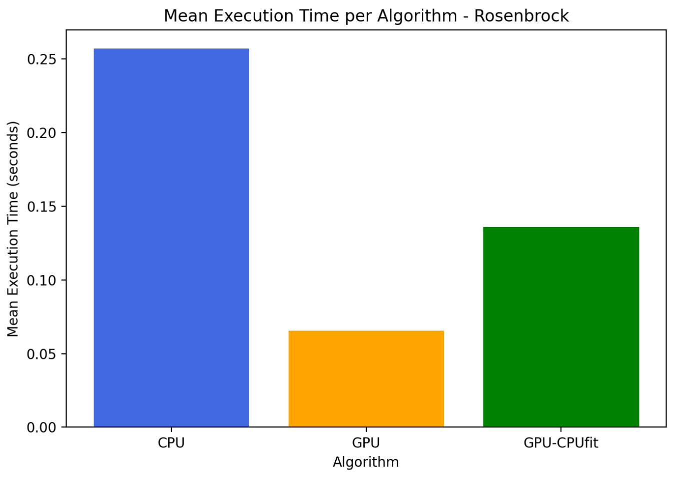

Figura 12: Mean execution time — Rosenbrock function.

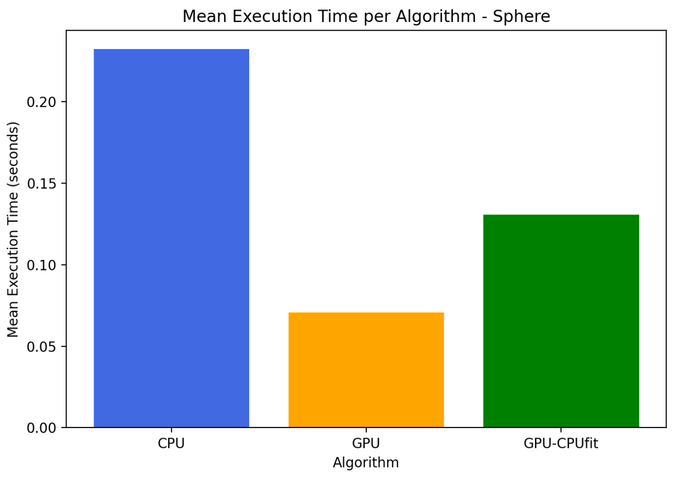

Figura 13: Mean execution time — Sphere function.

As shown above, the **pure GPU** version is generally the fastest. However, the `GPU-CPUfit` version shows a performance degradation due to CPU-side fitness evaluation — though still outperforming the pure CPU implementation. This compromise highlights the importance of aligning the computation model (GPU vs CPU) with the function evaluation pipeline to fully benefit from parallelism.

7 Conclusions

This project was an attempt to assess whether a hybrid and parallel implementation of the Firefly Algorithm—enhanced with local search (BFGS) and GPU acceleration—could provide practical benefits in terms of execution time and solution quality.

The codebase was developed following the structure of the existing `swarm-02` library, where this algorithm is integrated alongside other metaheuristics. The integration of multiple paradigms (OpenMP, CUDA, and local optimizers) required careful modularization to ensure compatibility and maintainability.

The most challenging part of the project was undoubtedly the implementation with CUDA, especially the management and correct usage of **shared memory**. However, through several iterations and incremental improvements, a working and extensible structure was achieved. The final CUDA version successfully parallelized the position updates of fireflies, with fitness evaluation optionally performed on the GPU or CPU.

To replicate the results or test the code, please refer to the `README` file in the project repository, which includes detailed build instructions, usage examples, and benchmarks.

Riferimenti bibliografici

[1] Jorge Nocedal and Stephen Wright. *Numerical optimization.* Springer Science & Business Media, 2006.

[2] Xin-She Yang. Firefly algorithms for multimodal optimization. *International symposium on stochastic algorithms*, 2009.