CS326 High-Performance Computing
Semester Final Project
April 25, 2023

Shashank Thakur
U22CS060
NIT Surat

Accelerating prism refraction search, a novel physics-based metaheuristic algorithm, using CUDA for optimization

In this paper, we implement and accelerate the Prism Refraction Search (PRS) algorithm, a single-solution metaheuristic inspired by the physical behavior of light passing through a prism, to solve high-dimensional optimization problems. While PRS shows promising accuracy, its runtime on traditional CPU implementations becomes a bottleneck as the dimensionality and population size increase. To address this, we parallelized key components of the algorithm using NVIDIA's CUDA platform, enabling massive parallel execution on the GPU. Our CUDA-accelerated implementation achieves significant speedups, up to 30x, compared to the CPU version, with equal or improved accuracy across various benchmark instances of the Rastrigin function and other such functions. These results demonstrate the effectiveness of GPU acceleration for complex metaheuristics and highlight PRS as a viable candidate for real-time and large-scale optimization tasks.

1 Introduction

Prism Refraction Search (PRS) is a nature-inspired optimization algorithm that simulates the behavior of light rays refracting through a prism to explore and exploit the search space. Like other single-solution or population-based metaheuristics, PRS maintains a set of candidate solutions and iteratively updates them based on a physical analogy, in this case refraction laws. The algorithm has demonstrated competitive performance in solving complex and multimodal optimization problems, particularly in lower-dimensional spaces [2].

However, as the dimensionality of the problem increases, the computational cost of PRS also rises sharply. This is due to the algorithm's reliance on evaluating large populations over many iterations, which introduces significant overhead when executed sequentially on a CPU. To overcome this limitation, we explore GPU acceleration using NVIDIA's CUDA platform.

By identifying and parallelizing the most computationally expensive components of PRS—such as population initialization, fitness evaluation, and position updates—we achieve substantial runtime reductions without compromising solution quality. Our CUDA-accelerated PRS is benchmarked on the Rastrigin function in various dimensions, and the results show considerable speedup and robust performance.

This paper presents our implementation approach, design decisions, performance evaluation, and a comparison of the CPU and GPU versions of PRS.

2 Preliminaries

Prism Refraction Search (PRS) draws its inspiration from the physics of light refraction through a triangular prism. The key physical foundation is Snell's Law, which governs how light bends when transitioning between media with different refractive indices as follows.

$$\frac{\sin i}{\sin r} = \frac{\mu_2}{\mu_1} = \mu_{21}$$

In a triangular prism, the light ray undergoes two successive refractions: at the entry and exit surfaces. The angle of deviation δ depends on the incident angle i, emergence angle e, and the prism's apex angle A as follows.

$$r_1 + r_2 = A$$

$$\delta = (i - r_1) + (e - r_2) = i + e - A$$

Solving for i in terms of the prism parameters and emergence angle leads to the following.

$$i = \arcsin\left(\sin A\sqrt{\mu^2 - \sin^2 e} - \cos A \sin e\right)$$

This physical behavior is abstracted in PRS to define the movement of solutions (rays) through the search space (the prism). Each solution adjusts its position based on a refracted direction toward better candidates, balancing exploration and exploitation in a manner analogous to light finding the shortest optical path.

3 Algorithm

3.1 Initial Solution

The Prism Refraction Search algorithm begins by initializing a candidate solution as a vector of dimension N. This vector represents the incidence angle components for the problem, and its fitness is computed as the objective function value, denoted δ.

$$i_t = [i_t^1, i_t^2, i_t^3, \ldots, i_t^N] \qquad \delta_t = \mathcal{F}(i_t)$$

```
1: Initialize population $i_0$ denoted by the solution as incident angle, subject to the bounds
   defined by prism angle $A_0$, see Eq.8
2: for $iter = 1 : MaxIters$ do
3:     for $i = 1 : OneSolution$ do
4:         for $j = 1 : Dimensions$ do
5:             Get fitness calculating the angle of incidence $\delta_t$ using Eq.7
6:             Obtain BestScore
7:             if($\delta_t$<BestScore)
8:             BestScore = fitness
9:             end if
10:        end for
11:    end for
12:    Calculate the refractive index $\mu_m$ using the Eq.10
13:    for $i = 1 : OneSolution$ do
14:        for $j = 1 : Dimensions$ do
15:            Update emergent angle $E_t$ using Eq.9
16:            Defined a random number $r1$ into $[-1, 1]$
17:            Update incident angle $i_{t+1}$ by Eq.11
18:        end for
19:    end for
20:    Update prism angle $A_{t+1}$ by Eq.12
21:    Update the best solution and position
22: end for
```

Figure 1: CPU-based PRS pseudocode

Each component of the solution vector is initialized randomly within the problem bounds, using a uniform distribution. Additionally, the prism's apex angle A_0 is initialized using the bounds of the search space.

$$i_0 = LB + (UB - LB) \times \mathcal{U}[0, 90]$$

$$A_0 = \max_{1\le j\le N}(LB) + \left(\min_{1\le j\le N}(UB) - \max_{1\le j\le N}(LB)\right) \times \mathcal{U}[15, 90]$$

Here, $\mathcal{U}[a, b]$ represents a value sampled uniformly from the interval $[a, b]$.

3.2 Solution Update

In each iteration, the algorithm updates the current solution vector based on its fitness and a physically inspired model of refraction. This process guides the population toward better solutions over time.
The refractive index μ_t at time step t is computed using:

$$\mu_t = \frac{\sin\left(\frac{A_t+\delta_t}{2}\right)}{\sin\left(\frac{A_t}{2}\right)}$$

Then, the **emergence angle** is derived from the deviation and the current incidence angle:

$$E_t = \delta_t - i_t + A_t$$

The incidence angle for the next iteration is updated as follows:

$$i_{t+1} = \sin^{-1}\left(-\sin E_t \cos A_t + r_1 \sin A_t \sqrt{\mu_t^2 - \sin^2 E_t}\right)$$

where r_1 is a uniformly distributed random number sampled from $[-1, 1]$.
To encourage convergence, the apex angle decays non-linearly over iterations, controlled by a decay factor α:

$$A_{t+1} = A_t \times \exp\left(\frac{-\alpha \cdot t}{\text{MaxIter}}\right)$$

This decay progressively reduces the search space, focusing the algorithm's exploration as it approaches potential optima.

4 CUDA Acceleration

4.1 Motivation for CUDA

Modern GPUs offer massive parallelism with thousands of lightweight threads operating concurrently, making them highly effective for data-parallel algorithms. The Prism Refraction Search (PRS) algorithm, which evaluates and evolves a population of candidate solutions, fits naturally into this paradigm. CUDA (Compute Unified Device Architecture), developed by NVIDIA, provides fine-grained control over GPU computation and memory, enabling efficient implementation of population-based metaheuristics like PRS [1].

The GPU memory model comprises global, shared, local, and constant memory spaces, each optimized for specific access patterns. In our implementation, we leverage global memory for solution vectors and results, and shared memory for intermediate reductions and communication within thread blocks.

4.2 Parallelization Strategy

To accelerate PRS on the GPU, we decompose the algorithm into a series of CUDA kernels, each targeting a specific parallel operation:

- `prs_init_incident_angles_kernel`: Initializes the incident angle vector; each thread handles one solution component using uniform sampling in (0°, 90°).
- `prs_init_emergent_angles_kernel`: Sets emergent angles to zero at the start; each thread initializes one component.

- `prs_calculate_fitness_kernel`: Computes the fitness (objective function value δ) for each individual; each thread evaluates one solution.
- `prs_reduce_fitness_sum_block_kernel`: Performs block-level reduction of the fitness array to compute partial sums.
- `prs_reduce_fitness_sum_final_kernel`: Final reduction across blocks to obtain the global fitness sum or average.
- `prs_reduce_fitness_min_block_kernel`: Performs block-level reduction to find local minima in the fitness values and their indices.
- `prs_reduce_fitness_min_final_kernel`: Aggregates block-level minima to find the global best solution and its fitness.
- `prs_copy_best_solution_kernel`: Copies the best solution vector across device memory; one thread per dimension.
- `prs_update_emergent_prs_update_incident_kernel`: Each thread handles an (individual, dimension) pair to compute emergent angles and update incident angles using the core PRS equations.

4.3 CUDA Implementation Details

Each CUDA kernel launches with a thread configuration optimized for occupancy and memory throughput. A block size of 1024 threads is used, which matches the warp-based execution model of NVIDIA GPUs.

Data	Location	Notes
incident_angles (pop × dim)	Global GPU	Updated every iteration
emergent_angles (pop × dim)	Global GPU	Device-only; used to compute next iteration
solution (dim)	Thread-local	Used per-thread to evaluate solutions
best_solution (dim)	Host + GPU	Synced between host and device
best_score	Host + GPU	Tracks fitness of best solution
lowerbound, *upperbound*	Global GPU	Constants per dimension for solution space mapping
delta_t (scalar)	Global GPU	Reduced average deviation value
prism_angle	Host + GPU	Updated on host and passed to device every iteration
params	Constant memory	Immutable meta-parameters (e.g., α, bounds)

Table 1: Memory layout and usage of major data structures

4.4 Pseudocode of CUDA-Accelerated PRS

Algorithm 1 CUDA-Accelerated Prism Refraction Search

```
1: Launch prs_init_incident_angles_kernel to initialize $i_0$
2: Launch prs_init_emergent_angles_kernel to set $E_0 = 0$
3: Initialize apex angle $A_0$
4: for $t = 1$ to MaxIter do
5:     Launch prs_calculate_fitness_kernel to evaluate all individuals
6:     Launch reduction kernels to compute average $\delta_t$
7:     Compute refractive index:
           $$\mu_t = \frac{\sin\left(\frac{A_t+\delta_t}{2}\right)}{\sin\left(\frac{A_t}{2}\right)}$$
8:     Launch prs_update_emergent_prs_update_incident_kernel:
        Update each $i_{t+1}$ using:
           $$i_{t+1} = \sin^{-1}\left(-\sin E_t \cos A_t + r_1 \sin A_t \sqrt{\mu_t^2 - \sin^2 E_t}\right)$$
9:     Update apex angle using exponential decay:
           $$A_{t+1} = A_t \cdot \exp\left(\frac{-\alpha \cdot t}{\text{MaxIter}}\right)$$
10:    Launch reduction kernels to identify best fitness and its index
11:    Launch prs_copy_best_solution_kernel if new best is found
12: end for
13: Return best_solution, best_score
```

5 Experimental Setup

All experiments were conducted on a laptop with an NVIDIA *GeForce MX250* GPU (compute capability 6.1, 3 multiprocessors, 4GB memory) and an Intel Core i7-1065G7 CPU running Arch Linux. The CUDA toolkit version was 6.1, using `nvcc`. Tests were performed on the Rastrigin function with dimensions 1 to 4. For the CUDA version, the population size was fixed at 1024 with iterations ranging from 10 to 12,000. For the CPU implementation, population sizes ranged from 18 to 1000 and iterations from 1,000 to 5,000.

6 Results and Comparative Analysis

6.1 Runtime Comparison

The following table presents the average runtime and best score obtained for the Rastrigin function of varying dimensions, evaluated over 100 runs for both the CPU and CUDA implementations. Each configuration was tuned independently for optimal and similar scores within the constraints of hardware and memory.

Table 2: Runtime and Score Comparison: Rastrigin Function (avg over 100 runs)

Dimension	**Platform**	**Runtime (s)**	**Score**	**MaxIter**	**PopSize**	**Speedup**
1	CPU	0.055207	0.000282	1000	18	29.61×
	CUDA	0.001864	0.000137	10	1024	
2	CPU	3.853343	0.099895	2000	400	264.88×
	CUDA	0.014552	0.059208	100	2048	
3	CPU	19.482192	0.151782	3000	800	28.56×
	CUDA	0.682174	0.104553	8000	1024	
4	CPU	57.037649	0.202980	5000	1000	30.15×
	CUDA	1.892460	0.249476	12000	1024	

Table 3: Runtime and score comparison: Sphere function (avg over 100 runs)

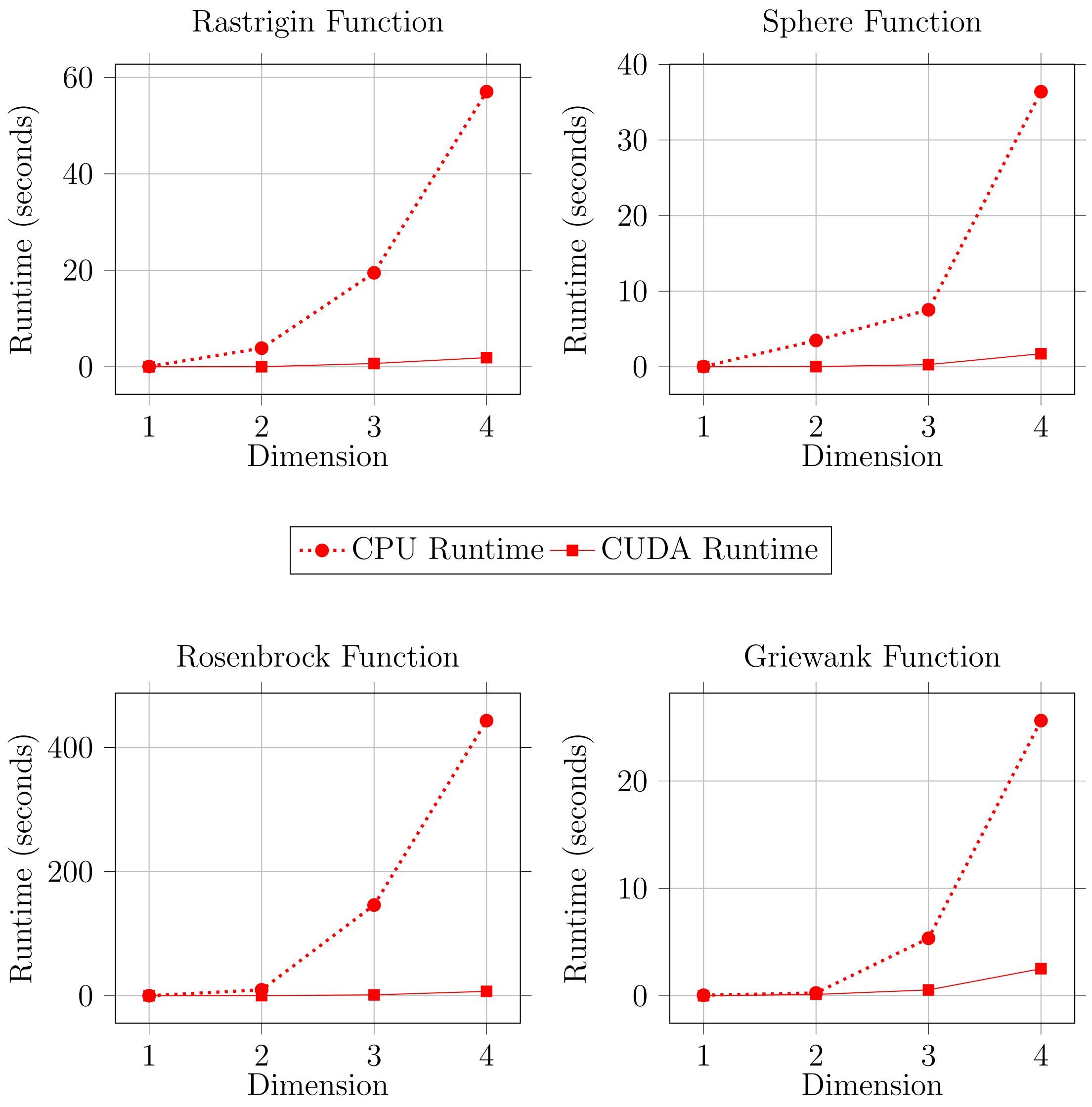
Dimension	**Platform**	**Runtime (s)**	**Score**	**MaxIter**	**PopSize**	**Speedup**
1	CPU	0.034360	0.000004	1000	20	18.59×
	CUDA	0.001848	0.000001	10	2048	
2	CPU	3.482226	0.080217	2000	200	142.48×
	CUDA	0.024434	0.081354	200	2048	
3	CPU	7.533911	0.279868	2000	500	26.36×
	CUDA	0.285832	0.199241	2000	2048	
4	CPU	36.393386	0.012489	5000	1000	21.05×
	CUDA	1.729266	0.023390	15000	1024	

Table 4: Runtime and score comparison: Rosenbrock function (avg over 100 runs)

Dimension	**Platform**	**Runtime (s)**	**Score**	**MaxIter**	**PopSize**	**Speedup**
1	CPU	0.053262	0.000000	1000	100	18.73×
	CUDA	0.002843	0.000000	10	1024	
2	CPU	9.520549	0.271784	1000	2000	57.06×
	CUDA	0.166852	0.271784	1000	2048	
3	CPU	146.079431	0.016273	10000	2000	106.07×
	CUDA	1.377080	0.022650	20000	2048	
4	CPU	443.198342	0.010350	15000	4000	63.15×
	CUDA	7.018276	0.015784	50000	2048	

Table 5: Runtime and score comparison: Griewank function (avg over 100 runs)

Dimension	**Platform**	**Runtime (s)**	**Score**	**MaxIter**	**PopSize**	**Speedup**
1	CPU	0.050755	0.007969	1000	20	17.06x
	CUDA	0.002974	0.009929	10	2048	
2	CPU	0.259725	0.040733	1000	100	2.05x
	CUDA	0.126386	0.028577	1000	1024	
3	CPU	5.352715	0.017556	5000	500	9.86x
	CUDA	0.542837	0.018956	5000	2048	
4	CPU	25.624583	0.015332	10000	1000	10.16x
	CUDA	2.521094	0.019204	10000	2048	



6.2 Accuracy Comparison

While both implementations achieved low scores, the CPU-based version occasionally produced slightly better results. This was primarily due to the more uniform distribution of the initial random population. The CPU's random number generator provided higher-quality randomness, resulting in better early-stage exploration of the search space. The CUDA implementation, constrained by a simpler RNG setup, sometimes lacked sufficient diversity in its initial population, slightly limiting its exploratory power.

6.3 Scalability

The CPU implementation scaled effectively with higher dimensionality, though at a significant cost in runtime. Conversely, the CUDA implementation exhibited significantly faster runtimes across all dimensions, though performance was limited by GPU memory constraints and memory access inefficiencies. Specifically, the flattened 2D representation of arrays led to poor memory coalescing, slowing down access patterns within warps.

Additionally, runtime for the CPU version increased more with population size, while the CUDA version's runtime was more sensitive to iteration count. This highlights a key trade-off in tuning hyperparameters for different hardware architectures.

6.4 Insights

The experimental results highlight the strengths and weaknesses of both approaches. CUDA excels in scenarios where rapid evaluations of large populations are needed, even at the cost of slightly reduced accuracy. The CPU variant, on the other hand, offers better control over exploration quality, especially for small populations or lower iteration budgets.

These observations suggest a potential for hybrid strategies, where initial exploration could be performed on the CPU, followed by accelerated exploitation using the GPU. Additionally, improvements to the CUDA RNG or adoption of parallelized high-quality random number generation could further enhance CUDA's search efficiency.

7 Conclusion

The Prism Refraction Search (PRS) algorithm is primarily an exploitation-based approach, with limited exploratory capabilities during the initial phase when the angles are initialized with random numbers. It is best utilized as a refinement tool to perform a finer search once a broader search space has been narrowed down by a more efficient exploration algorithm.
In this study, we implemented the PRS algorithm on both CPU and CUDA, observing significant performance improvements with the CUDA implementation. However, despite its faster execution, the CUDA-based algorithm faced scalability challenges, particularly in high-dimensional problems. This may be attributed to the random number generator used in the CUDA kernels, which potentially limits the exploration of the solution space. To achieve higher exploration, the maximum number of iterations would need to be considerably increased.
Several improvements could enhance the performance of the CUDA implementation:

- **Improved memory allocation:** Better memory coalescence could be achieved by optimizing how data is accessed by threads within a warp. Currently, the 2D array of solutions is flattened to fit into GPU memory, and while a 3D allocation exists, padding requirements may reduce efficiency. Exploring other memory allocation techniques could improve performance.

- **More efficient kernel design:** Leveraging advanced techniques such as parallel reduction, as discussed in NVIDIA's documentation and forums, could enhance the efficiency of summing and finding minimum values within arrays, thus optimizing the kernels further.

The CUDA-accelerated implementation achieved a $40\times$ speedup compared to the CPU version, while also handling a larger population size and more iterations. Furthermore, it demonstrated better solution accuracy. These results highlight the potential of parallelization to not only improve the runtime of population-based metaheuristic algorithms but also enhance their solution quality, underlining the significant benefits of utilizing GPU computing for such tasks.

References

[1] NVIDIA Corporation. *CUDA Toolkit Documentation 12.8*. 2025. URL: `https://docs.nvidia.com/cuda/`.

[2] R. Kundu, S. Chattopadhyay, S. Nag, et al. "Prism refraction search: a novel physics-based metaheuristic algorithm". In: *Journal of Supercomputing* 80 (2024), pp. 10746–10795. DOI: `10.1007/s11227-023-05790-3`. URL: `https://doi.org/10.1007/s11227-023-05790-3`.