A GPU-centric Genetic Algorithm for Polygonal Image Reconstruction

Anonymized

Abstract. Although Genetic Algorithms (GAs) significantly reduce the size of the solution space they explore, they remain computationally expensive for complex and high-dimensional optimization problems. This paper investigates a fully GPU-accelerated GA for triangle-based polygonal image reconstruction, formulated as a highly challenging optimization task. The proposed approach exploits GPU parallelism at multiple levels of the evolutionary process, with particular emphasis on an efficient GPU-based image rendering strategy, identified as the dominant computational bottleneck. Extensive experiments conducted on a diverse set of benchmark images demonstrate that the proposed GPU-based GA achieves significant performance gains. In particular, speedups exceeding two orders of magnitude are observed on large-scale problem instances when compared to a validated sequential baseline from the literature. These results highlight the effectiveness of GPU acceleration for computationally intensive evolutionary optimization tasks.

Keywords: Genetic Algorithms · GPU Programming · Parallel Metaheuristics · Image Reconstruction · High-Performance Computing

1 Introduction

Among metaheuristic optimization methods, Genetic Algorithms (GAs) are particularly well suited to tackle complex, nonlinear, and high-dimensional problems, especially when conventional deterministic solvers prove ineffective. As members of the broader class of evolutionary algorithms, GAs iteratively evolve populations of candidate solutions through biologically inspired operators [9]. Despite their flexibility and robustness, high computational cost often limits their practical applicability on large or complex problem instances.

A natural way to overcome this limitation is to exploit the massive parallelism offered by Graphics Processing Units (GPUs). While many existing approaches rely on hybrid CPU–GPU designs or partially offload computations, fully GPU-centric GAs—where all operators and data structures are executed on the device—remain underexplored. Such designs require careful mapping of the evolutionary process to the GPU, minimizing host–device communication and maximizing throughput.

Polygonal image reconstruction—and more specifically, triangle-based image reconstruction—constitutes a computationally demanding optimization problem

for which evolutionary approaches have demonstrated strong effectiveness. Beyond its visual interpretability, this application offers practical advantages over raster representations, such as compactness and inherent vectorization. Despite these characteristics, parallel and GPU-based implementations remain scarce. Previous work includes Wiransky [11] with a GA, Cavallaro *et al.* [2] with a hybrid machine learning approach, and Valois *et al.* [10] with a parallel island-model GA.

In this paper, we investigate a fully GPU-centric GA (GcGA) for triangle-based image reconstruction, designed to exploit parallelism at all stages of the evolutionary process directly on the GPU. Unlike hybrid approaches, our method keeps the entire evolutionary workflow on the device, enabling efficient execution and scalability to large problem instances. To the best of our knowledge, this is the first approach to this application that leverages GPU computation.

The main contributions of this work are as follows:

- We propose a fully GPU-centric GA, contributing to the still limited body of work on fully device-resident evolutionary optimization.
- We provide a rare GPU-based parallelization of the polygonal image reconstruction problem.
- We demonstrate the ability of the approach to efficiently handle large-scale instances, achieving speedups exceeding $\times 100$ compared to a validated sequential baseline.

The remainder of the paper is organized as follows. Section 2 formulates triangle-based image reconstruction as a complex optimization problem. Section 3 details the proposed GPU-centric GA. Section 4 presents experimental results, including validation of the baseline and performance analysis of the GPU-based implementation. Finally, Section 5 concludes the study and outlines directions for future research.

2 Problem Statement

The problem addressed in this work consists in reconstructing a target image using an ordered set of colored triangles, which may overlap and vary in shape, size, color, and opacity. The objective is to identify an optimal configuration of vertex coordinates and color attributes for N triangles such that their superposition approximates the target image while minimizing information loss. Images are represented in 8-bit RGB format. Following [10], triangle-based image reconstruction is formulated as the following optimization problem.

Definition 1 (Image Reconstruction Problem). *Given a target image X, the goal is to determine an ordered set of N colored triangles, denoted $z \in \Omega^N$, such that the image $Y = g(z)$ obtained by their superposition minimizes the objective function $f(X, Y)$:*

$$\min_{z \in \Omega^N} f\big(X, g(z)\big). \tag{1}$$

Here, Ω denotes the set of all feasible configurations for a single triangle (i.e., vertex coordinates and RGBA color attributes), while g represents the rendering function that maps an ordered set of N triangles to a rasterized image. The function f quantifies the discrepancy between a candidate reconstruction Y and the target image. In this work, the Mean Squared Error (MSE) is adopted as the objective function due to its simplicity and satisfactory empirical performance [10]. The pixel-wise MSE between two images X and Y is defined as

$$\mathrm{MSE}(X, Y) = \frac{1}{3p} \sum_{c \in \{R,G,B\}} \sum_{i=1}^{p} \left(X_{i,c} - Y_{i,c}\right)^2, \tag{2}$$

where p denotes the total number of pixels in each image.

The space of feasible solutions associated with Problem (1) is exceedingly large. For instance, reconstructing a 1000×1000 pixel image using $N = 100$ colored triangles yields on the order of $\mathbf{10^{2762}}$ possible configurations. The resulting scale and dimensionality of this combinatorial search space make the image reconstruction task particularly challenging. As a result, traditional deterministic optimization methods are impractical, and exploration-oriented metaheuristics—most notably genetic algorithms—naturally emerge as well-suited approaches for tackling such a problem.

3 GPU-based Genetic Algorithm Design

Genetic algorithms operate by evolving a population of candidate solutions—referred to as individuals—over a fixed number of iterations known as generations. Due to their population-based structure and typically high computational demands, GAs are well suited to parallel execution on GPU architectures. Modern GPU devices provide massive data-parallel processing capabilities by enabling the concurrent execution of thousands of lightweight threads, hierarchically organized into blocks and grids.

A substantial body of literature has demonstrated the effectiveness of GPU-based implementations in accelerating GAs [4]. These contributions include CUDA-based approaches achieving notable speedups over CPU implementations [3], as well as GPU-based island-model GAs reporting significant improvements in both computational performance and solution quality [6, 7]. However, many existing GPU-accelerated GAs focus on offloading only selected operators—most commonly the fitness evaluation—to the device. Designing a fully GPU-centric GA, in which all genetic operators are executed on the device, therefore remains more challenging and requires careful algorithmic and architectural design.

In this section, we begin by briefly describing the genetic operators used to address the image reconstruction problem defined in (1). These operators are identical to those employed by Valois *et al.* [10], whose implementation serves in this paper as the sequential baseline for the subsequent performance evaluation. We then detail their parallelization on GPU architectures, which forms the core of the proposed fully GPU-accelerated GA.

3.1 Genetic Operators

The GA is initialized with a population $\mathcal{P}^0$ of P individuals, each encoding a configuration of N randomly generated triangles. The population is then evolved over a fixed number of generations G by sequentially applying, at each generation, the genetic operators described below. The output of the algorithm is the reconstructed image associated with the best fitness value—*i.e.*, the lowest MSE (2) with respect to the target image—obtained at the final generation.

Selection. A new population $\mathcal{P}^i$ is formed by selecting the most promising individuals from the previous generation $\mathcal{P}^{i-1}$. To introduce stochasticity, tournament selection is employed: for each selection event, the individual with the best fitness is chosen from a randomly sampled subset of Q candidates. This process is repeated until P individuals have been selected.

Crossover. Assuming an even population size P, individuals from $\mathcal{P}^i$ are randomly paired to form parent couples. Each pair generates two offspring by exchanging R randomly selected triangles.

Mutation. Each individual in $\mathcal{P}^i$ undergoes mutation with probability p_m. A mutation consists of either modifying one vertex of a randomly selected triangle or altering its color attributes. Both mutation types occur with equal probability, and at most one mutation is applied per individual.

Rendering. Each individual in $\mathcal{P}^i$, represented as an ordered set of triangles, is rendered into an image tensor, forming the set $\mathcal{Y}^i$. This process is performed by sequentially overlaying the triangles onto a blank canvas.

Evaluation. Each rendered image in $\mathcal{Y}^i$ is compared to the target image using the MSE objective function (2), yielding a fitness value for every individual and forming the set $\mathcal{F}^i$.

Sorting. After evaluation, the population $\mathcal{P}^i$ is sorted according to fitness, as both the elitism and selection operators require ranked individuals.

Elitism. The best-performing half of the current population $\mathcal{P}^i$ is combined with the best-performing half of the previous generation $\mathcal{P}^{i-1}$ to form the next population.

3.2 GPU-centric Genetic Algorithm

The design of a fully GPU-centric GA requires carefully rethinking each stage of the evolutionary process to exploit the massive parallelism offered by modern GPU hardware. In this section, we describe how all major GA operators—rendering, fitness evaluation, sorting, and genetic operations—are implemented directly on the device, ensuring that no host–device data transfers are needed during execution.

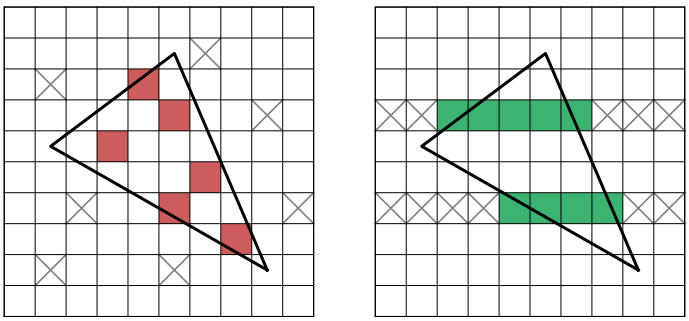
GPU-based Rendering Among all operators in the GA workflow, rendering is the most computationally expensive—particularly as the resolution of the reconstructed image increases—as confirmed by the experimental profiling presented later in this paper. A natural first parallelization strategy therefore consists in offloading this computation to the GPU.

To this end, our approach exploits two complementary levels of parallelism: (i) *iteration-level parallelism*, whereby the rendering process is performed concurrently for all individuals in the population, and (ii) *solution-level parallelism*, in which the rendering of a single individual is itself parallelized to fully exploit the device. The latter is especially effective, as image rendering is inherently data-parallel and well suited to GPU execution. For the solution-level parallelization, two GPU-based rendering strategies are considered. These approaches, illustrated in Fig. 1, enable the adjustment of the granularity of parallelism:

- **Pixel-wise rendering.** For each colored triangle, each GPU thread is responsible for updating a single pixel of the image. Pixels located inside the triangle are colored accordingly, while all others remain unchanged.
- **Row-wise rendering.** For each colored triangle, each GPU thread processes a single row of the image. Within rows intersecting the triangle, pixels that lie inside the shape are colored, whereas the remaining pixels are left unaffected.

Rendering produces fully rasterized images whose memory footprint becomes substantial as both image resolution and population size increase. Transferring these image tensors back to the host would therefore incur significant host–device communication overhead, severely impacting performance. To avoid this bottleneck, the subsequent fitness evaluation is also performed directly on the GPU, thereby eliminating any pixel data transfers throughout the GA execution.

GPU-based Evaluation In standard GAs, fitness evaluation is often the most computationally demanding operator. In the present application, it constitutes the second most expensive step after rendering and can likewise be efficiently parallelized on the GPU. As for rendering, two levels of parallelism are exploited: (i) *iteration-level parallelism*, with all individuals being evaluated concurrently,



(a) Pixel-wise (b) Row-wise

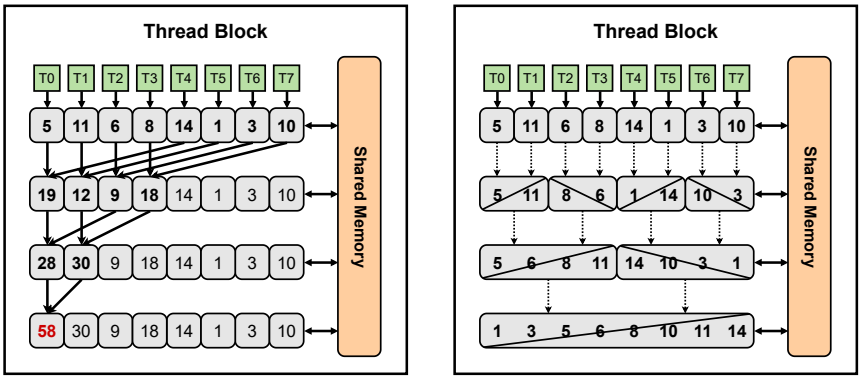
Fig. 1: Illustration of the two GPU-based rendering strategies considered for triangle rasterization: (a) pixel-wise rendering, where each thread processes a single pixel, and (b) row-wise rendering, where each thread processes an entire image row intersecting the triangle.

and (ii) *solution-level parallelism*, whereby the fitness computation for each individual is parallelized.

The latter consists in computing the MSE objective function (2) in parallel, which ultimately reduces to a large summation on the device. To efficiently perform this operation, we adopt the parallel reduction with sequential addressing strategy presented in [5] and illustrated in Fig. 2a. Image data are stored using an interleaved memory layout $(R, G, B, \ldots, R, G, B)$, which improves spatial locality and enables more coalesced global memory accesses during evaluation. In addition, shared memory is exploited within each thread block to further reduce memory access latency. Together, these design choices aim at maximizing evaluation throughput on the GPU.

GPU-based Sorting The population ordering step following fitness evaluation also requires careful consideration, as designing an efficient GPU-compatible sorting algorithm is non-trivial. To address this challenge, we employ the *bitonic sort* algorithm [1], which is well suited to parallel execution on GPUs [8]. Our implementation leverages shared memory, and its operating principle is illustrated in Fig. 2b.

To further improve performance, we avoid physically reordering the population during sorting, as this would involve numerous intra-device memory exchanges of triangle vertices and color attributes. Instead, only the indices of individuals are sorted according to their MSE values and subsequently used to reference the corresponding solutions. It should be noted that bitonic sort requires the population size P to be a power of two, which introduces an additional design constraint.



(a) Parallel sum (b) Bitonic sort

Fig. 2: GPU-based primitives used in the proposed implementation: (a) parallel sum reduction within a thread block using shared memory and sequential addressing, and (b) bitonic sort executed within a thread block, illustrating the successive compare-and-swap stages leading to a fully sorted sequence.

GPU-based Selection, Crossover, Mutation, and Elitism The remaining genetic operators admit a comparatively straightforward GPU parallelization. Tournament selections, triangle exchanges during crossover, and potential mutations across individuals are all performed concurrently. The elitism step is implemented by concurrently assembling the best-performing half of the current population with the best-performing half of the previous generation.

Although these operators are less computationally intensive than rendering and evaluation, their GPU-based execution provides additional speedups and, crucially, enables the entire evolutionary loop to be executed on the device without host–device data transfers. Together, they complete the design of a fully GPU-accelerated GA framework.

Figure 3 illustrates the resulting GPU-centric workflow for polygonal image reconstruction. All GA operators—including population initialization—are executed on the GPU through dedicated kernel functions invoked by the CPU host across G generations. As shown in Fig. 3, all intermediate data—namely population individuals, rendered images, and fitness values—reside exclusively in device memory throughout the execution. The role of the CPU is therefore limited to transferring the target image X to the device and retrieving the final reconstructed image with the best fitness.

4 Experimental Results

4.1 Experimental Setup

The algorithms presented in this study are implemented in `C++` and `CUDA 12.2`, with `gcc 10.4.0` as the host compiler. All experiments are conducted on a high-performance computing cluster equipped with `AMD EPYC 7513` CPUs and `NVIDIA A100` GPUs featuring `40 GiB` of device memory.

In the remainder of this paper, the following GA implementations are evaluated:

- **SGA.** The standard CPU-based GA introduced by Valois *et al.* [10]. After validation against a representative approach from the literature, this implementation serves as the sequential baseline for assessing the performance of the proposed GPU-based variant.
- **GcGA.** The GPU-centric GA presented in Section 3.2, designed to deliver substantial performance improvements over the SGA baseline.

The use of a single sequential baseline is motivated by prior work on polygonal image reconstruction. In particular, [2] has demonstrated the effectiveness of genetic algorithms for this complex optimization task when compared to classical approaches such as Tabu Search, Iterated Local Search, and Artificial Immune Systems. Furthermore, the parallel island-model GA proposed by Valois *et al.* [10], while improving solution quality, incurs higher computational cost than its sequential counterpart. In this context, the SGA implementation—sharing the same genetic operators as GcGA—provides a relevant and fair reference for a strictly performance-oriented comparison, which constitutes the primary focus of this work.

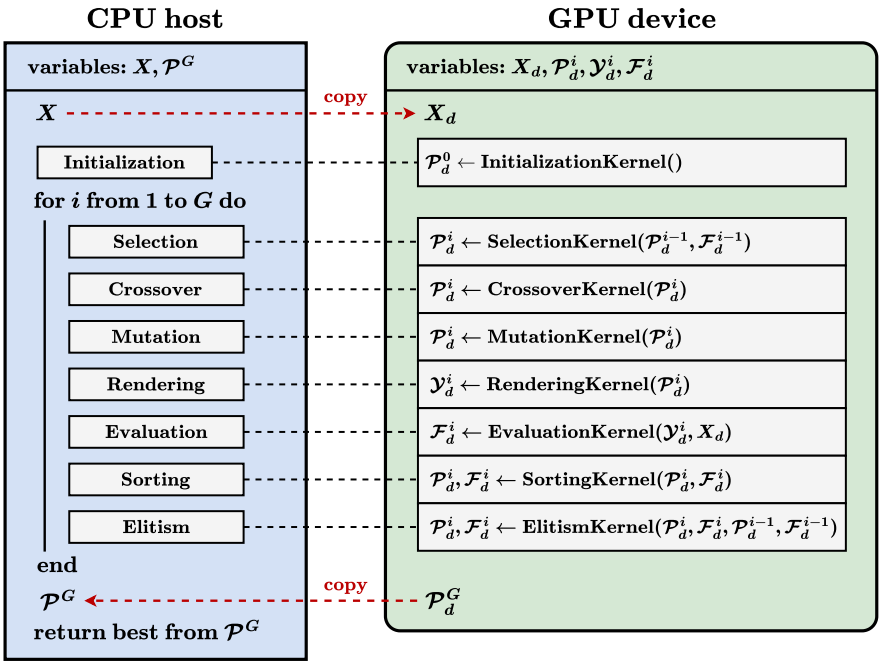


Fig. 3: GPU-centric GA workflow illustrating the interaction between the CPU host and the GPU device. All genetic operators are executed on the device *via* dedicated kernel functions, while data remain resident in GPU memory throughout the evolutionary process. Host–device transfers are limited to the input target image and the final reconstructed solution.

4.2 Validation of the Sequential Baseline

We begin the experimental evaluation by validating the performance and accuracy of the considered sequential **SGA** baseline, which corresponds to the standard GA implementation introduced in [10]. Specifically, we compare this baseline against the related study by Cavallaro *et al.* [2] in order to ensure methodological consistency before proceeding to the performance assessment of the proposed GPU-based extension. In [2], the authors propose a hybrid approach, denoted **GAML**, which incorporates machine learning techniques based on linear regression for adaptive parameter tuning.

For a direct comparison, we replicate the experimental setup described in [2], which considers the reconstruction of the 165×190 pixel *Mona Lisa* portrait shown in Fig. 4a. In the experimental setting reported in [2], GAML is executed with a population of 197 individuals for 10 000 generations, using $N = 100$ triangles and the MSE—consistent with our formulation—as the optimization criterion. To ensure a fair comparison, our SGA experiments adopt a closely matching parametric configuration, summarized in Table 1. Fig. 4 shows a visual comparison of reconstructions produced by GAML and SGA, while Table 2 re-

Table 1: Parameter configuration used to replicate the experimental setup of GAML [2] for validating the sequential GA baseline (SGA [10]).

symbol	**description**	**value**
G	number of generations	10 000
P	population size	196
Q	tournament selection size	40
N	number of triangles	100
R	triangle swaps for crossover	50
p_m	mutation probability	0.7

(a) Target (b) GAML (c) SGA

Fig. 4: Mona Lisa portrait reconstruction: (a) target image, (b) GAML output [2], and (c) baseline SGA output [10].

Table 2: Quantitative comparison between GAML and SGA on the Mona Lisa reconstruction in terms of final MSE fitness and execution time (in seconds).

	GAML	**SGA**
MSE	1377.7	397.3
time (s)	7142.5	5506.2

ports the corresponding quantitative results in terms of final MSE and execution time. Execution times are provided for indicative purposes only, as the two implementations were evaluated in different software and hardware environments. For consistency, SGA results are averaged over ten independent runs, with the reconstruction in Fig. 4c corresponding to the best-performing run. As shown in Table 2, SGA achieves substantially lower MSE than GAML on the considered instance, yielding higher reconstruction quality. This is consistent with Valois *et al.* [10] and positions SGA as a state-of-the-art sequential implementation and a robust CPU-based baseline for evaluating the proposed GPU-centric approach.

4.3 GPU-based Genetic Algorithm Performance Evaluation

We now turn to the performance evaluation of the proposed GPU-centric implementation, denoted **GcGA**. Since GcGA employs the same genetic operators

(a) Mona Lisa (b) Starry Night (c) Bibi & Lala (d) Saturn (e) Basic-Fit

Fig. 5: Benchmark images used for evaluating the performance of GcGA.

Table 3: Parameter configuration used for the performance evaluation of SGA and GcGA across all benchmark images.

symbol	**description**	**value**
G	number of generations	5000
P	population size	128
Q	tournament selection size	24
N	number of triangles	100
R	triangle swaps for crossover	50
p_m	mutation probability	0.7

as the previously validated sequential implementation from [10], the following analysis focuses exclusively on computational performance.

We consider the diverse set of benchmark images shown in Fig. 5, similar to the one employed in [10]. These images serve as targets for polygonal reconstruction and exhibit a wide range of color variations and visual detail. Their resolution is varied across experiments to control the difficulty of Problem (1), while all other GA parameters remain fixed throughout this section and are listed in Table 3. For each parameter configuration, the reported results are averaged over ten independent GA runs. These values are then averaged across the five benchmark images, yielding a total of 50 independent runs per reported value, thereby ensuring statistical robustness and consistency.

As a preliminary step, we perform an experimental profiling of the sequential SGA baseline on the benchmark set (*cf.* Fig. 5). Table 4 reports the distribution of execution time among genetic operators for increasing image resolutions: 64×64, 128×128, 256×256, 512×512, and 1024×1024 pixels. The results in Table 4 clearly indicate the dominance of the rendering operator—and, to a lesser extent, the evaluation operator—in the overall computational cost. This effect becomes increasingly pronounced as resolution increases, thereby strongly motivating the GPU-based acceleration strategies introduced in Section 3.2.

We now compare the computational efficiency of the two GPU-based rendering methods introduced in Section 3.2 to identify the strategy best suited to the GPU-centric framework, as rendering is by far the dominant cost in the sequential baseline (*cf.* Table 4). Figure 6 reports, for both methods and for

Table 4: Distribution of execution time (%) among genetic operators for SGA at increasing image resolutions.

resolution	**64 px**	**128 px**	**256 px**	**512 px**	**1024 px**
selection	5.0	1.8	0.5	0.1	< 0.1
crossover	0.8	0.3	0.1	< 0.1	< 0.1
mutation	2.4	0.8	0.2	0.1	< 0.1
rendering	77.0	90.4	87.5	89.8	90.4
evaluation	1.4	1.9	10.4	9.7	9.5
sorting	8.3	3.0	0.8	0.2	< 0.1
elitism	5.1	1.8	0.5	0.1	< 0.1

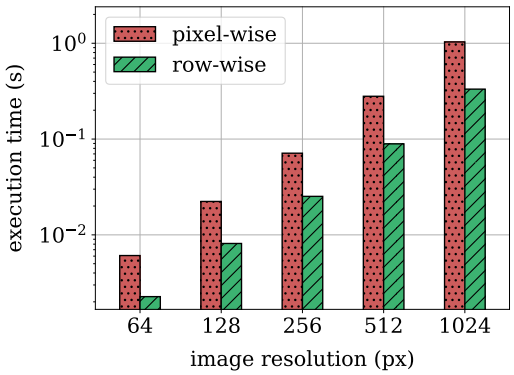


Fig. 6: Average rendering time (log scale) for the two rendering strategies across increasing image resolutions.

increasing image resolutions, the average rendering time (log scale) over 10 runs for a randomly generated population of size $P = 512$, with $N = 100$ triangles per individual. The row-wise strategy (*cf.* Fig. 1b) consistently outperforms the pixel-wise approach (*cf.* Fig. 1a) by approximately a factor of three, indicating that it provides sufficient parallelism to achieve high GPU utilization while being more computationally efficient. Accordingly, the row-wise rendering method is retained for the final performance evaluation.

Finally, we turn to the overall performance evaluation of the proposed GPU-centric implementation, GcGA. We assess the computational speedups achieved by GcGA over its sequential counterpart, SGA, for the polygonal reconstruction of the benchmark images (*cf.* Fig. 5). Table 5 reports the average execution times and corresponding speedups for increasing image resolutions, highlighting the substantial benefits of GPU acceleration enabled by the complete on-device execution. While execution times grow rapidly with problem size for both approaches, the GPU-based implementation consistently mitigates this growth, yielding progressively larger speedups at higher resolutions. In particular, for 1024×1024 pixel images, GcGA achieves an average speedup of up to $\mathbf{\times 114.7}$

Table 5: Average execution times (in seconds) of SGA and GcGA together with the corresponding speedups for increasing image resolutions

resolution	time (s)		speedup
	SGA	GcGA	
64 px	365.2	6.37	×57.3
128 px	1022.3	12.1	×84.5
256 px	3889.7	42.4	×91.7
512 px	14 913.2	137.8	×108.2
1024 px	58 667.8	511.4	×114.7

over the sequential GA implementation from [10]. These results should nonetheless be interpreted with caution, as the reference implementation is sequential. However, as discussed in the introduction, to the best of our knowledge GcGA is not only the first GPU-based approach to this problem, but also the first to be explicitly designed to reduce execution time. In this context, the reported speedups provide meaningful evidence of the effectiveness of GPU-accelerated GAs and further motivate their application to computationally demanding optimization problems related to image processing.

5 Conclusions and Future Work

This work has demonstrated the substantial computational benefits of GPU acceleration for improving the efficiency of genetic algorithms when addressing complex optimization problems. Polygonal image reconstruction served as a meaningful and visually interpretable benchmark to evaluate the proposed GPU-centric GA, denoted GcGA. By leveraging modern GPU architectures and identifying an effective GPU-based rendering strategy—the most computationally demanding operator within the GA workflow—extensive experimental results show that GcGA achieves triple-digit speedups on large-scale problem instances when compared to a representative CPU-based approach from the literature.

A promising direction for future research consists in the dynamic tuning of algorithmic parameters, for instance through the integration of machine learning techniques within the GA framework [2], with the aim of further improving effectiveness and solution quality. Another avenue involves extending the proposed GPU-based implementation to an island-model GA [6, 7, 10]. Such a multi-population approach is currently under investigation and is already yielding encouraging results by exploiting multiple GPU devices in parallel.

Acknowledgments

Anonymized

References

1. Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference. pp. 307–314. ACM (1968). https://doi.org/10.1145/1468075.1468121
2. Cavallaro, C., Cutello, V., Pavone, M., Zito, F.: Machine learning and genetic algorithms: a case study on image reconstruction. Knowledge-Based Systems **284** (2024). https://doi.org/10.1016/j.knosys.2023.111194
3. Cheng, J.R., Gen, M.: Accelerating genetic algorithms with GPU computing: a selective overview. Computers & Industrial Engineering **128**, 514–525 (2019). https://doi.org/10.1016/j.cie.2018.12.067
4. Harada, T., Alba, E.: Parallel genetic algorithms: a useful survey. ACM Computing Surveys **53**(4), 86:1–86:39 (2021). https://doi.org/10.1145/3400031
5. Harris, M.: Optimizing parallel reduction in CUDA. Tech. rep., NVIDIA (2007), CUDA webinar tutorial slides
6. Luong, T.V., Melab, N., Talbi, E.G.: GPU-based island model for evolutionary algorithms. In: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO). pp. 1089–1096. ACM (2010). https://doi.org/10.1145/1830483.1830685
7. Pospichal, P., Jaros, J., Schwarz, J.: Parallel genetic algorithm on the CUDA architecture. In: Applications of Evolutionary Computation. pp. 442–451. Springer (2010). https://doi.org/10.1007/978-3-642-12239-2_46
8. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS). pp. 1–10 (2009). https://doi.org/10.1109/IPDPS.2009.5161005
9. Talbi, E.G.: Metaheuristics: from design to implementation. Wiley (2009)
10. Valois, J.P., Firmin, T., Melab, N.: A parallel island genetic algorithm for triangle-based image reconstruction. In: Proceedings of the IEEE/ACS 22nd International Conference on Computer Systems and Applications (AICCSA). pp. 1–8 (2025). https://doi.org/10.1109/AICCSA66935.2025.11315373
11. Wirsansky, E.: Hands-on genetic algorithms with Python: apply genetic algorithms to solve real-world AI and machine learning problems. Sciendo (2024)