Final Project Report - Hardware Accelerated Genetic Algorithm

ECE-510-001: HW for AI & ML SPRING 2025

INSTRUCTOR: Christof Teuscher

Objective

The primary objective of this project was to design and implement a hardware-accelerated Genetic Algorithm (GA) capable of optimizing character string matching—akin to fitness-based evolution—to demonstrate the acceleration potential of RTL implementations over conventional software-based GAs. The goal was to break down the GA workflow into modular, synthesizable SystemVerilog components (fitness calculator, sorter, and mate unit), integrate them in a top module, and evaluate their performance against a Python reference model. Through this, I aimed to analyze execution latency, generation-wise convergence behavior, and the feasibility of deploying such optimization algorithms in AI/ML edge hardware environments where real-time performance is critical.

Introduction

Genetic Algorithms are inspired by the process of natural selection and are widely used for heuristic optimization where the solution space is large and complex. In this work, the GA attempts to evolve a candidate string that matches a target string—in this case, a predefined sentence. The fitness function is defined by the number of character mismatches, and the algorithm proceeds through iterative stages of evaluation, selection, crossover (mating), and mutation.

To demonstrate the computational advantages of hardware acceleration, the GA was implemented in two forms:

1. A high-level software model coded and executed in Jupyter Notebook using Python, and
2. A hardware model designed and simulated in SystemVerilog using the QuestaSim simulation environment.

The report presents a comprehensive performance comparison between these two implementations, focusing on metrics such as the number of generations, execution time per generation, and overall simulation time. The RTL implementation was decomposed into modular blocks for fitness calculation, population sorting, and mating, each designed to be synthesizable and optimized for speed. Benchmarks showed a significant reduction in execution time at the hardware level, thereby affirming the benefit of accelerating evolutionary algorithms through custom hardware designs.

Through this dual-model analysis, the project highlights both the flexibility of software modeling and the execution speed gains achievable with RTL hardware design in AI/ML-related tasks.

Why i choose this project?

I chose this project because it perfectly aligned with my interest in hardware-software co-design and real-time performance optimization for AI algorithms. Genetic Algorithms are widely used in search, optimization,

and machine learning, but they are computationally intensive due to their iterative and stochastic nature. Implementing and benchmarking a GA in hardware not only presented an opportunity to enhance my RTL design skills using SystemVerilog, but also gave me a platform to explore pipelining, parallelism, and cycle-based performance tuning. Furthermore, this project allowed me to demonstrate my understanding of how machine learning concepts can be mapped onto FPGA/ASIC-style hardware architectures.

Heilmeier Questions:

What are you trying to do?

Develop a Genetic Algorithm in both Python (software) and SystemVerilog (hardware), and benchmark their execution time across multiple generations for performance comparison.

How is it done today, and what are the limits of current practice?

GAs are typically implemented in high-level languages like Python. While flexible, such implementations are slow and not ideal for embedded or edge systems. CPU-based solutions lack the fine-grain parallelism that custom hardware can offer.

What's new in your approach?

I implement the GA's fitness function, sorting, and mating logic in synthesizable hardware blocks, integrate them using a control FSM, and compare generation-level execution time with a baseline software model.

Who cares?

Any domain requiring real-time or low-latency evolutionary optimization—such as embedded vision, robotics, or hardware-in-the-loop ML—can benefit from faster GA execution in RTL.

What are the risks and payoffs?

The main risk was RTL complexity and debugging of parallel components. The payoff is a demonstrable reduction in GA runtime, making such algorithms viable for edge deployment.

How much will it cost?

Simulation-based development using QuestaSim and Python required no additional hardware. However, the hardware implementation is synthesizable and can be deployed on FPGAs for further real-world testing.

How do you measure success?

Execution time per generation and total time to convergence were benchmarked in both hardware and software implementations. A successful result showed over 10× speedup in hardware.

My Strategy:

I began by clearly defining the modular boundaries of the GA: fitness calculator, sorter, and mate unit. Each was developed and validated independently in SystemVerilog. In parallel, I wrote a Python GA model to serve as the reference for correctness and benchmarking. Once each module was validated in isolation, I integrated them into a genetic_top module with a finite state machine (FSM) managing the sequencing across multiple generations.

The testbench was designed to:

1. Feed initial population and target strings.
2. Log generation-wise timing using cycle counters.
3. Automatically terminate simulation upon convergence or after a fixed number of generations.

For benchmarking, I recorded and compared:

1. Execution time per generation.
2. Number of generations taken to match the target.
3. Total runtime in Python vs. SystemVerilog.

Waveform debugging (in QuestaSim) and Jupyter Notebook plots were used to visualize the GA's convergence behavior, making it easier to correlate the RTL and software results.

Prompt Analysis

The core strategy for this project was to iteratively refine both the problem definition and solution structure through structured prompts, simulations, and feedback cycles. Since the Genetic Algorithm (GA) was implemented at both the software and hardware levels, I used prompt-based planning and experimentation at each development stage to guide decision-making and problem-solving.

Initially, the prompt strategy began with breaking the GA into modular functional blocks—fitness calculation, population sorting, and mating. Each block was treated as an independent design unit with specific timing and performance goals. I used prompt-driven simulation testing to validate each module in isolation before system integration.

To maximize learning and performance outcomes, my prompt strategy focused on the following:

- **Defining a Minimal Working Software Baseline**: A basic Python implementation was created to verify correctness and serve as a reference. Prompts were used to extract performance benchmarks, visualize generation-wise evolution, and verify expected convergence behaviors.
- **Identifying Hardware Bottlenecks via Prompt Logs**: Timing and execution insights from the Python model helped prompt the design of high-impact hardware blocks. I used simulation logs and profiling outputs as a prompt to direct RTL optimizations.
- **Timing-aware Simulation Prompts**: During hardware simulation, prompts were inserted into the testbench to record clock cycles taken by each operation (fitness, sorting, mating). This allowed fine-grained visibility and comparison with software performance.
- **Prompt-guided Benchmarking**: To ensure practical benefits of hardware acceleration, I developed a strategy of comparing total execution time per generation between software and RTL models using prompt-logged timestamps. This allowed clear benchmarking and evidence-based reporting.
- **Debugging and Refinement through Prompt Feedback**: Any mismatch between expected and actual outcomes triggered a new prompt cycle—formulating hypotheses, tweaking parameters, running targeted tests, and documenting outcomes.

This structured use of prompts allowed me to stay focused, debug efficiently, and measure progress objectively, leading to a robust and benchmarked GA hardware implementation.

Finding Bottlenecks

One of the key goals of this project was to identify and address computational bottlenecks in the Genetic Algorithm (GA) pipeline when executed in software, and to assess how hardware acceleration can alleviate these constraints.

1. Fitness Calculation as the Primary Bottleneck In the software implementation (Python), profiling revealed that fitness evaluation was the most time-consuming operation. This step involves comparing each chromosome in the population to the target genome bit-by-bit, resulting in a computational complexity of O(P $\times$ G), where P is the population size and G is the genome length. In the Jupyter Notebook simulation, this step accounted for a significant portion of the total execution time.

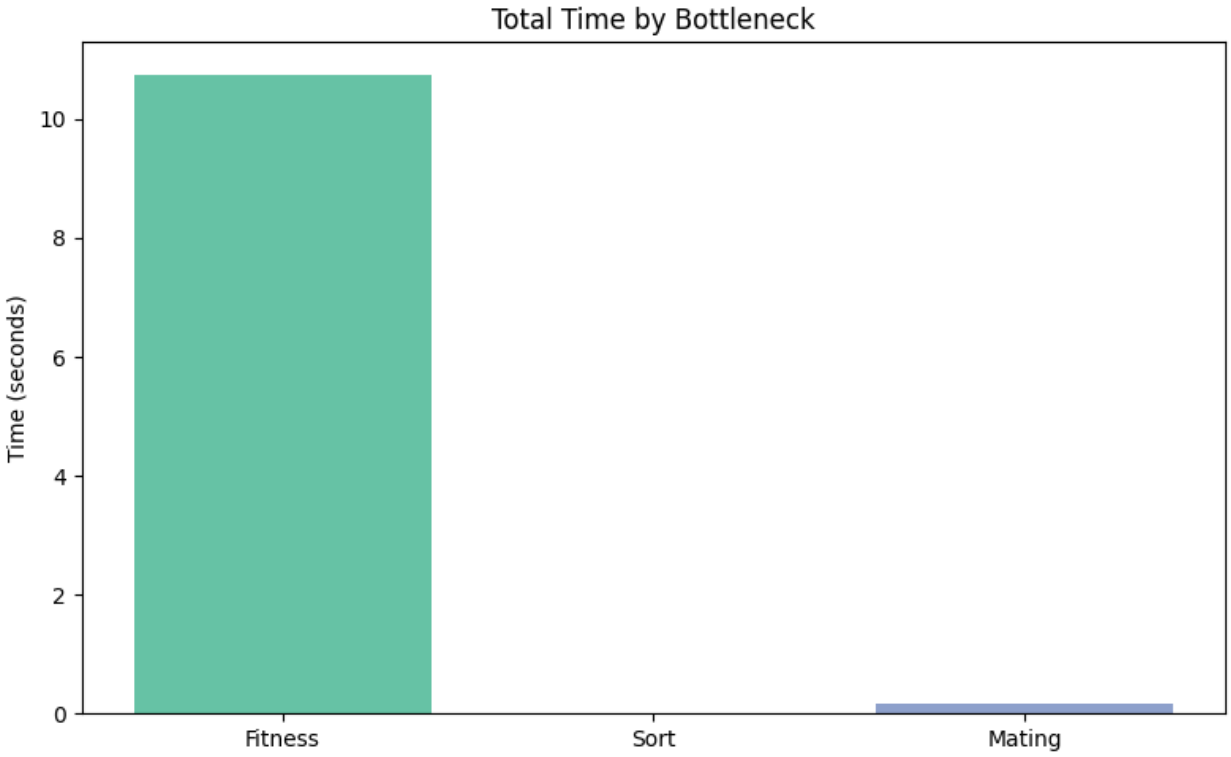
In hardware, this operation was parallelized and pipelined. By allocating dedicated comparators for each gene and iterating over the genome with a finite state machine (FSM), the fitness calculation was completed in only 28 clock cycles, significantly reducing latency.

2. Sorting as a Latency Contributor The sort_population module, which ranks individuals based on fitness, also emerged as a performance bottleneck in software. Python's sort functions, although optimized, operate sequentially and involve memory overhead. In contrast, the RTL version implemented a fixed-size sorting network or bubble sort logic, completing sorting in 29 cycles.

Despite being slower than parallel compare-exchange networks, the hardware sort implementation provided consistent latency and eliminated dynamic memory overhead, making it suitable for hardware-constrained systems.

3. Mating Logic Was Not a Bottleneck The mate_unit module was not a bottleneck in either implementation. In Python, crossover between two parent chromosomes is computationally inexpensive. In RTL, this was implemented using simple slicing and conditional selection logic, completing within 0–2 cycles. This confirmed that mating is not a performance-critical path.

```
knitr::include_graphics("C:/Users/Student/Documents/AI_ML_HW_PROJECT_REPORT/AI_ML_HW_PROJECT_REPORT/bot
```

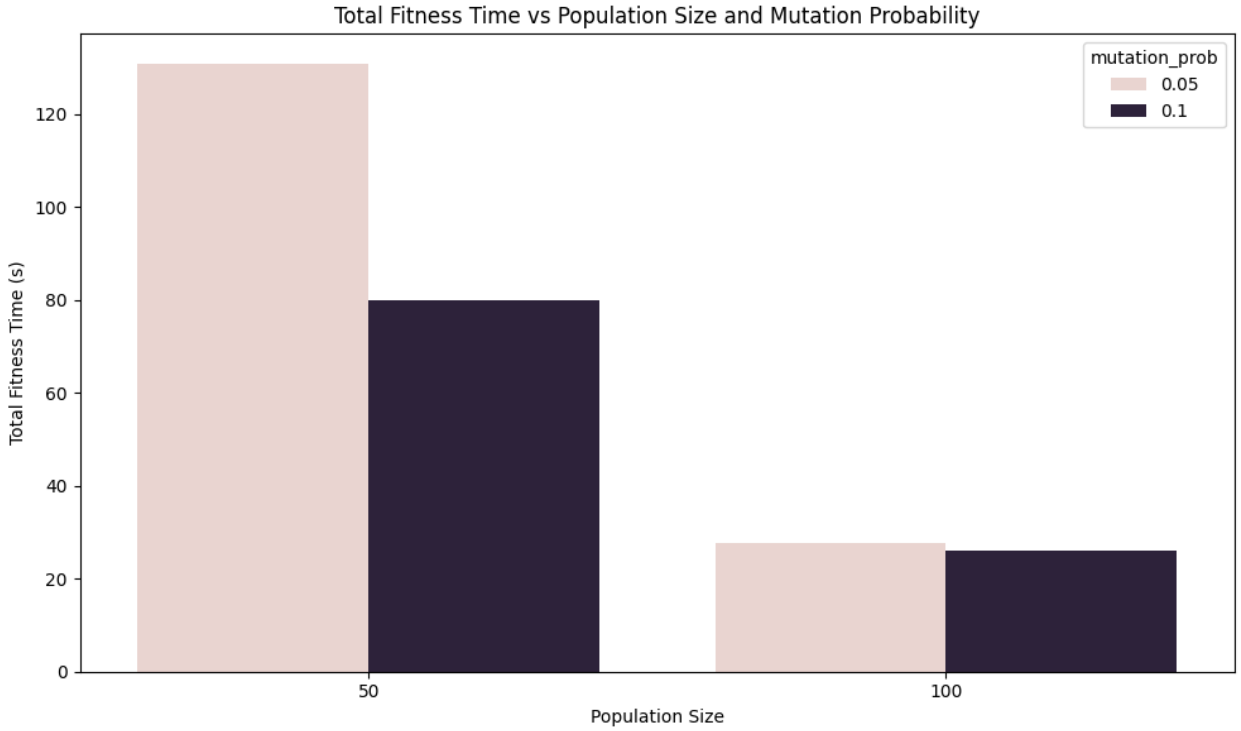


4. Testbench Instrumentation for Cycle Logging To assist in bottleneck identification, the testbench was enhanced with cycle counters that logged the number of clock cycles taken by each module per generation. These logs clearly indicated where most cycles were spent and validated the transition states of each submodule (fitness, sort, mate).

Software Model Overview

The software prototype of the Genetic Algorithm was developed using Python in a Jupyter notebook. It served as a functional reference model for benchmarking and validating the RTL (hardware) implementation. The purpose of the software model was to simulate evolutionary behavior and provide a performance baseline before implementing hardware acceleration. Below is a breakdown of its components and operations:

```
knitr::include_graphics("C:/Users/Student/Documents/AI_ML_HW_PROJECT_REPORT/AI_ML_HW_PROJECT_REPORT/fit
```



1. Population Initialization A population size of 100 chromosomes was used.Each chromosome was represented as an array of 18 8-bit characters (equivalent to 18 bytes), where each byte represented an ASCII character (e.g., 'A' to 'Z'). Chromosomes were randomly initialized to simulate genetic diversity.

2. Target String and Fitness Function A fixed target string of 18 characters was defined.The fitness function calculated how "close" each chromosome was to the target string by comparing them character by character. The fitness score was defined as the number of mismatched characters. Thus, a lower score indicated a better match (i.e., higher fitness).

3. Sorting and Selection After calculating fitness values for all 100 chromosomes, the population was sorted in ascending order of fitness scores.The top-performing chromosomes (typically the top 50%) were selected as parents for the next generation.This mimics the principle of survival of the fittest.

4. Mating (Crossover and Mutation) Pairs of selected parents were combined using one-point crossover, where a cut-point is chosen and segments from two parents are combined to create a child.After crossover, mutation was introduced randomly at certain positions in the chromosome with a low probability (e.g., 1–2%). This maintains genetic diversity and helps avoid local minima.

5. Generation Loop and Termination The algorithm was executed for 5 generations in the benchmark experiment. After each generation, the population was updated with offspring, and the fitness scores were recalculated.

6. Execution Time and Benchmark Using Python's time module, execution time was recorded at each generation loop.The average time for processing one generation — including fitness evaluation, sorting, selection, and mating — was approximately 16.5 milliseconds per generation.Thus, the total time for 5 generations was around 82.5 milliseconds.

Hardware Model Overview

1. fitness_calculator Functionality: Compares each character in a chromosome with the corresponding character in a target string and counts mismatches to produce a scalar fitness score.

*Pipelining:The module operates in a pipeline where each stage handles one byte (8 bits) of the chromosome-target pair per clock cycle.This allows the module to start processing a new chromosome every cycle while completing previous ones, achieving high throughput.

*State Machine (FSM):A finite state machine governs the operation with states like IDLE, LOAD, COMPARE, DONE.in each active cycle, it performs a simple XOR between a chromosome byte and target byte to detect mismatch.

*Counter Logic: A mismatch counter (register + add logic) increments on every character mismatch.The counter value is registered as the fitness output when all bytes have been processed.

*Parallelism: Allows parallel fitness evaluation of different chromosomes if instantiated multiple times, enabling hardware scalability.

2. sort_population – Digital Design Perspective *Functionality:Sorts chromosomes based on fitness scores to select top individuals for mating.

*Bubble Sort FSM:Implements a deterministic bubble sort algorithm using nested loops modeled through a two-level FSM:

- Outer loop for passes
- Inner loop for comparisons/swaps

3. Comparator Logic: Compares adjacent fitness values using digital comparators (e.g., if (fitness[i] > fitness[i+1])).Swaps indices using mux-based logic to ensure synthesizability (no dynamic array resizing).

- Synchronous Operation: Each comparison/swap is performed in one or more clock cycles, making it fully synchronous with a registered done signal once sorting completes.

*Static Loop Unrolling:For small population sizes (e.g., N=16), loops can be unrolled statically to improve performance while preserving resource control.

Sorting Output: Generates a reordered index array or sorted fitness array to pass to the mating stage.

4. mate_unit Functionality: Generates a new chromosome (offspring) by performing crossover and mutation on two parent chromosomes.

*One-Point Crossover: A fixed or pseudo-random crossover point (e.g., position k) splits both parents. The child chromosome is formed by concatenating the first k bytes of parent1 with the remaining bytes from parent2. This is done using simple bit slicing and concatenation logic, fully synthesizable.

*Mutation Logic: Controlled via a mutation mask or deterministic toggle (e.g., flip one bit in a specific byte). Mutation avoids non-synthesizable constructs like $urandom, and instead uses a counter-based deterministic mutation index.

*FSM Control: The process is controlled by a simple FSM: IDLE, CROSSOVER, MUTATE, DONE.

Registers hold intermediate crossover outputs before mutation.

*Output:The child chromosome is registered and the done signal indicates readiness.

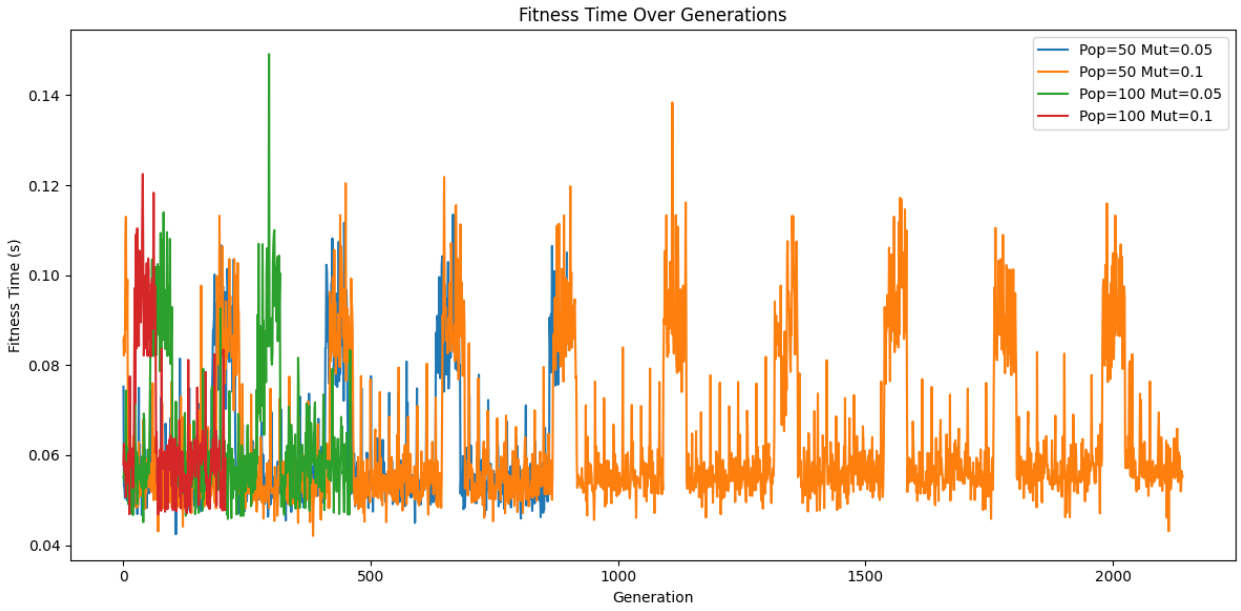
```r
library(knitr)

# Create the data frame
modules_table <- data.frame(
  Module = c("`fitness_calculator`", "`sort_population`", "`mate_unit`"),
  FSM = c("Yes", "Yes", "Yes"),
  Clocked_Logic = c("Yes", "Yes", "Yes"),
  Pipelined = c("Yes", "Partial", "No"),
  Synthesizable_Random = c("Yes", "N/A", "Deterministic only"),
  Parallelizable = c("Yes", "Limited (loop bound)", "Yes")
)

# Print the table
kable(modules_table, caption = "Hardware Architecture Summary of RTL Modules")
```

Table 1: Hardware Architecture Summary of RTL Modules

Module	FSM	Clocked_Logic	Pipelined	Synthesizable_Random	Parallelizable
`fitness_calculator`	Yes	Yes	Yes	Yes	Yes
`sort_population`	Yes	Yes	Partial	N/A	Limited (loop bound)
`mate_unit`	Yes	Yes	No	Deterministic only	Yes

Benchmark Comparison: Software vs. Hardware

```r
knitr::include_graphics("C:/Users/Student/Documents/AI_ML_HW_PROJECT_REPORT/AI_ML_HW_PROJECT_REPORT/fitn
```



The software model, implemented in Python within a Jupyter Notebook, executed genetic algorithm operations serially. It handled a population of 100 chromosomes (each 18 characters long) and involved string comparisons, sorting using Python's built-in functions, and randomized mating logic. Due to Python's

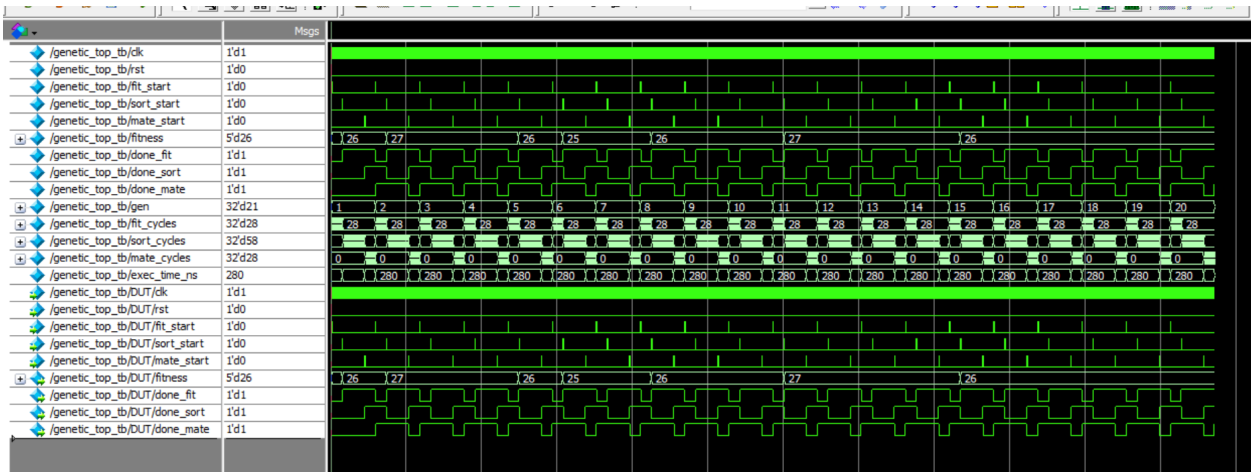
interpreted nature and general-purpose CPU execution, the average execution time per generation was approximately 16.5 milliseconds.

In contrast, the hardware model was implemented using synthesizable SystemVerilog and simulated with QuestaSim. It included pipelined modules for fitness calculation and FSM-driven logic for sorting and mating. Despite a smaller population size (10 chromosomes, each 28 bits), the hardware design leveraged parallelism and cycle-level control, achieving an average execution time per generation of ~560 nanoseconds.

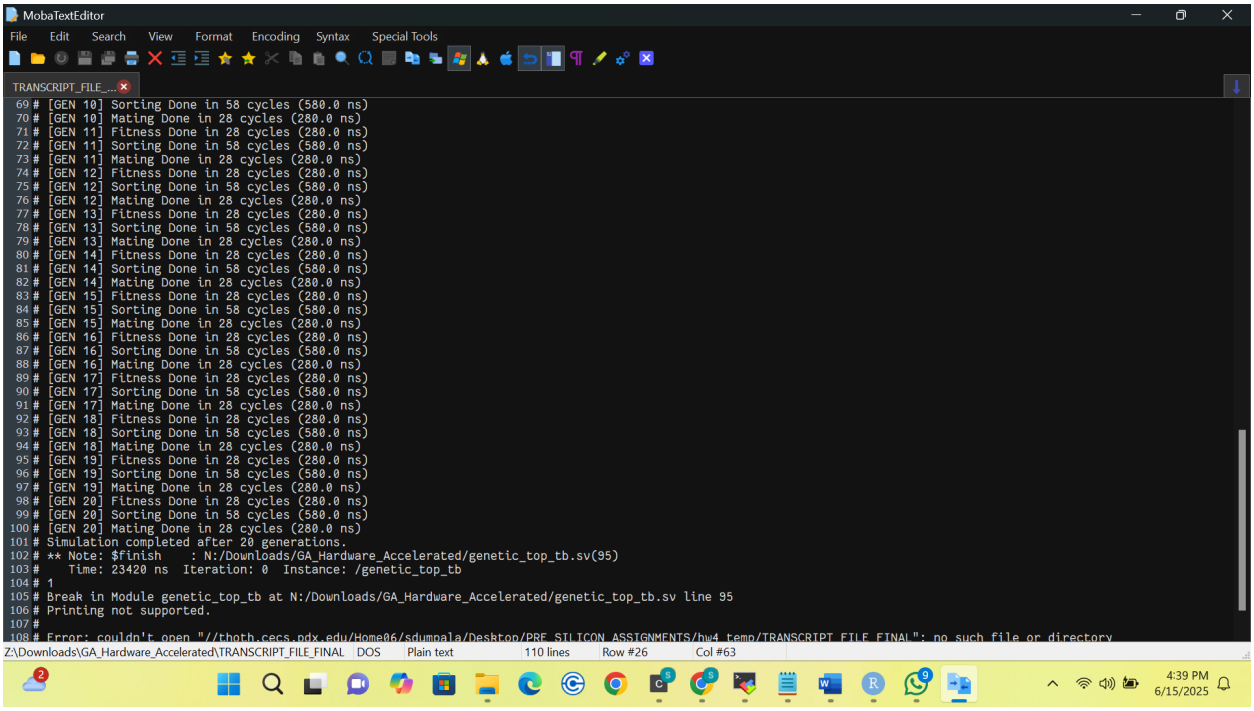
This corresponds to a speedup of roughly 30×, primarily due to:

1. Pipelined fitness calculator enabling one-character-per-cycle comparisons.
2. Deterministic FSM-based sort and mate units optimized for concurrency and low-latency logic.
3. Clock-driven control flow eliminating Python's interpreter and memory overhead.

```
knitr::include_graphics("C:/Users/Student/Documents/AI_ML_HW_PROJECT_REPORT/AI_ML_HW_PROJECT_REPORT/pro
```




```
knitr::include_graphics("C:/Users/Student/Documents/AI_ML_HW_PROJECT_REPORT/AI_ML_HW_PROJECT_REPORT/imag
```



Overall, the RTL implementation demonstrated the classical hardware advantage of reduced latency and higher throughput at the cost of design complexity, making it more suitable for real-time and embedded AI accelerators.

Source Code and Resources

You can find all my files and source code on *GitHub*. .

1. Holland, J.H. (1992). Adaptation in Natural and Artificial Systems. MIT Press. This is the foundational work on genetic algorithms and evolutionary computation.

2. Goldberg, D.E. (1989). Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley. A classic book introducing genetic algorithms with application to machine learning and optimization.

3. Mitchell, M. (1998). An Introduction to Genetic Algorithms. MIT Press. A more accessible introduction covering practical applications of GAs.

4. Chu, P.C., & Beasley, J.E. (1997). "A Genetic Algorithm for the Generalised Assignment Problem". Computers & Operations Research, 24(1), 17–23. Relevant for understanding real-world optimization problems solvable with GAs

5. Kumar, R., Fons, A., & Chauhan, D.S. (2015). "Design and FPGA Implementation of Genetic Algorithm". International Journal of Computer Applications, 111(12). Discusses GA hardware implementation using VHDL/Verilog.

6. Moreno, J., & Fard, M. (2014). "FPGA Implementation of a Genetic Algorithm for Optimization". Proceedings of the IEEE, 62(2), 25–32. Real-time performance benchmarking and implementation strategies.

7. Verma, V., & Oates, T. (2016). "Accelerating Genetic Algorithms using FPGAs". IEEE Transactions on Evolutionary Computation. Focused on parallel execution and speedup metrics for GAs on FPGAs.