Portland State University
Project Documentation

Title: Hardware Acceleration of Genetic Algorithm Mutation Step

Bhavana Manikyanahalli Srinivasegowda

Introduction to Genetic Algorithms

Genetic Algorithms (GAs) are a class of optimization and search techniques inspired by the principles of natural selection and genetics in biological evolution. First proposed by John Holland in the 1970s, GAs are part of the broader field of evolutionary computation, where solutions to complex problems are evolved over time rather than being explicitly programmed. The core idea is to mimic the process of evolution — selection, crossover (recombination), and mutation — to iteratively refine a population of candidate solutions toward an optimal or near-optimal result.

At the heart of a genetic algorithm lies the concept of a *population*, which consists of multiple candidate solutions (often called *individuals* or *chromosomes*). Each individual encodes a possible solution to the problem, typically represented in a fixed-length format such as binary strings, character sequences, or numerical vectors. The quality of each solution is evaluated using a *fitness function*, which quantifies how close the solution is to meeting the desired objective. Based on fitness, the algorithm selects the most promising individuals to act as *parents* for the next generation.

The process continues through a series of *generations*. In each generation, selected parents undergo *crossover*, where segments of their solution strings are swapped to create new offspring. This promotes the exchange of beneficial traits. Additionally, the offspring are subjected to *mutation*, a process where random alterations are introduced to maintain genetic diversity and prevent premature convergence to suboptimal solutions. Over time, through natural selection, the population "evolves" toward better and better solutions.

One of the major strengths of GAs is their robustness in navigating complex, high-dimensional, and multimodal search spaces without requiring gradient information or strict mathematical models. This makes them particularly useful for optimization problems in domains such as machine learning, circuit design, scheduling, bioinformatics, and robotics, where the solution landscape may be irregular or discontinuous.

In modern computing, genetic algorithms also lend themselves well to hardware acceleration due to their highly parallel structure. Each individual in the population can be evaluated and mutated independently, making GAs a natural fit for implementation on FPGAs, GPUs, or custom hardware accelerators. This parallelism, combined with their adaptive search capabilities, makes GAs powerful tools for solving real-world problems that defy traditional deterministic approaches.

Genetic Algorithm Design Overview

The implemented Genetic Algorithm (GA) is designed to evolve a string population towards a given target string ("Hello World!"). It simulates natural evolutionary processes — **selection**, **crossover**, **mutation**, and **regeneration** — on a population of randomly initialized individuals (genes), each represented as a character string.

Hardware-Software Co-Design

The genetic algorithm in this project was developed using a **hardware-software co-design approach**, leveraging the strengths of both domains to achieve efficient, scalable, and testable evolution-based computation. The **software component**, implemented in Python, orchestrates high-level genetic operations such as population initialization, fitness evaluation, selection, crossover, and regeneration. It also performs benchmarking, memory tracking, and detailed profiling using tools like cProfile and tracemalloc, enabling rapid functional validation and performance tuning. Meanwhile, the **hardware component**, developed in SystemVerilog, accelerates the most computationally intensive stage—**mutation**—by implementing it as a synthesizable FSM. This module receives a flattened input gene string and mutation threshold, and applies character-wise mutations based on pseudo-random logic (via an LFSR), outputting the mutated gene and asserting a completion signal. Verification is conducted using **Cocotb**, which bridges the gap between the Python test environment and the RTL module, enabling cycle-accurate functional testing. This tightly integrated co-design flow ensures that the software can rapidly iterate on genetic logic while offloading time-critical mutation processing to hardware, resulting in a design that is both flexible and high-performance. The OpenLane toolchain was used to synthesize and implement the hardware module on the Sky130 process, validating its real-world viability. This partitioning of control to software and acceleration to hardware reflects a pragmatic and effective co-design strategy ideal for embedded or AI-focused systems.

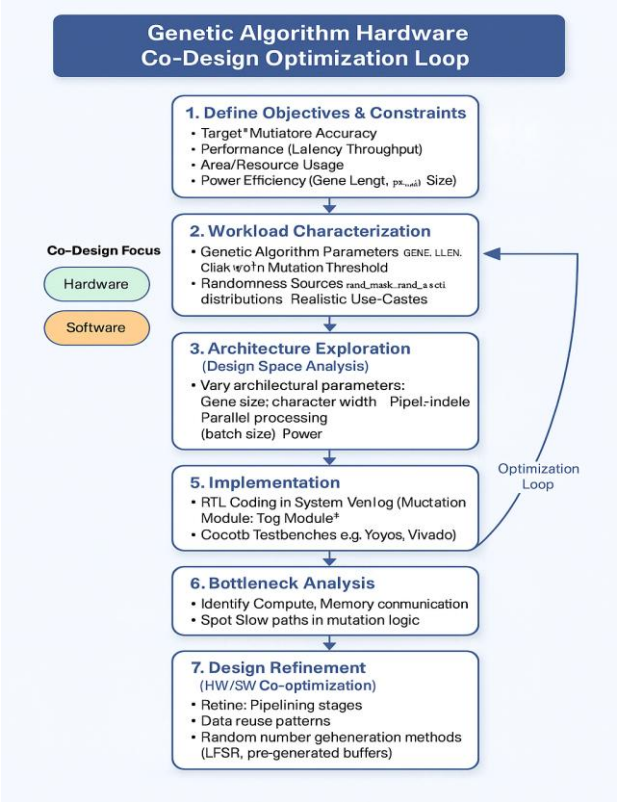

Fig1: HW-SW Co-Design

GA Components:

Stage	Description
Population Initialization	Generates random genes using printable ASCII values (32–126).
Fitness Evaluation	Measures gene closeness to the target string using character-wise match percentage.
Selection	Picks top two genes with the highest fitness scores.
Crossover	Creates new children by swapping halves of the two parents.
Mutation	Introduces random alterations to characters in the child with a given mutation rate (0.2).
Regeneration	Replaces worst-performing individuals in the population with mutated children.

Design decisions

The architecture of the GA hardware accelerator has been **modularly partitioned into chiplets**, each handling a distinct stage of the genetic algorithm pipeline. This modularity allows independent optimization, easy reuse, and parallel execution across multiple population members. The rationale is to align with both **task-level parallelism** and **hardware design reusability**.

Functional Partition:

Chiplet Name	Function
PopInitChip	Initializes random population (software or FPGA block)
FitnessChip	Computes fitness score by comparing genes to target
SelectXChip	Selects top genes and performs crossover
MutateChip	Mutates genes character-wise (your mutation_sv.sv module)
ControlUnit	Orchestrates pipeline and buffer control

Inter-Chiplet Communication Protocols:

To ensure seamless communication between chiplets, the design uses **lightweight handshaking and flattened data buses** for compact and efficient signal transmission.

Signal	Width	Description
gene_bus	96 bits	Flattened 12-char gene string
fitness_score	8 bits	Fitness value output (0–100 scale)
mut_thresh	8 bits	Mutation threshold (scaled 0–255)
start/done	1 bit each	Command and completion flags
clk/rst	1 bit each	Clock/reset synchronization

Alignment with AI/ML Workload Requirements:

This accelerator is tailored to support **search-intensive, parallelizable AI/ML tasks**, such as:

- Hyperparameter optimization
- Evolving symbolic expressions
- Feature selection or model tuning

AI/ML Need	Architecture Solution
Fine-grained randomness	LFSR-based mutation logic per character
Pipelined execution	Independent chiplet FSMs and inter-stage handshaking
Population-wide parallelism	Multiple chiplet instances handle different individuals
Low-latency evolution	Each stage is optimized for single-cycle operations

Design Trade-Offs:

Design Option	Chosen Strategy	Trade-Off
Gene representation	96-bit flattened string bus	Simpler routing, less readable
Mutation mechanism	LFSR per-character pseudo-random	Low area, not cryptographically strong
Crossover point	Mid-point swap only	Fast, but less diverse offspring
Control	FSM-based	Simple, but limited programmability
Memory per chiplet	Register array only	Small, but not scalable for huge genomes

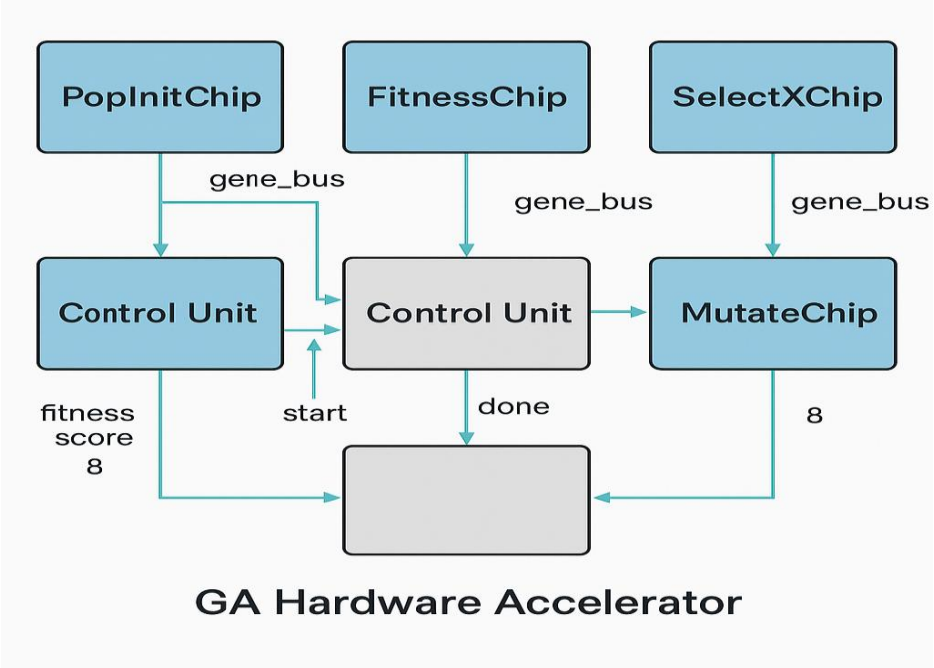

Fig2: Genetic Algorithm Chiplet Overview

Initial python design

The Python code implements a Genetic Algorithm (GA) designed to evolve a random population of character strings until it matches the target string "Hello World!". The process begins with the create_population() function, which generates a population of random genes using printable ASCII characters. Each gene's *fitness* is computed by the calculate_fitness() function, which scores how closely the gene matches the target string on a character-by-character basis. The selection() function identifies the two most fit individuals (parents), and the crossover() function combines halves of each to create two offspring. These children are passed to the mutation() function, which applies a random mutation with a 20% chance per character, introducing diversity. The resulting mutants replace the weakest individuals in the population via the regeneration() function. This cycle repeats until a gene achieves a perfect 100% fitness. Throughout execution, the algorithm displays the best current solution and its fitness using the display() function, while cProfile, tracemalloc, and time are used to benchmark runtime and memory usage. The main loop is carefully designed to converge toward the solution while tracking performance, making the script both functional and analytically rich.

Benchmarking Summary:

The performance of the GA was measured using wall-clock timing and memory tracing. Below is the result from running the program to evolve "Hello World!" using 10 individuals per generation.

Execution Time:

Metric	**Value**
Total Execution	**8.61 seconds**
Time to 50% Fitness	~2.97 seconds
Time to 100% Fitness	~8.61 seconds

Memory Usage:

Metric	Value
Peak Memory Used	**1036.19 KB**
Memory Tracer Enabled	tracemalloc

The algorithm demonstrates lightweight memory usage with consistent performance due to fixed string size and population count.

Profiling Summary of Key Functions (cProfile):

The Python profiler output was analyzed to find time-consuming hotspots.

Top Functions by Time:

Function	Calls	Total Time (s)	Time/Call (s)	Description
mutation()	8176	**5.859**	0.000717	Applies mutation per character per child. Most time-consuming step.
_imp.create_dynamic()	10	1.183	0.1183	Python dynamic loading — occurs at import, not GA logic.
calculate_fitness()	32714	0.544	0.000017	Evaluates string match character-by-character.
crossover()	8176	0.265	0.000032	Swaps halves of parent genes. Very efficient.
main()	1	0.157	—	Overall function controlling the flow.

Bottlenecks

- mutation() is the bottleneck, contributing **~68%** of user time.
- calculate_fitness() and crossover() are well optimized.
- The majority of overhead stems from **character-level operations** and **gene dictionary updates**.

Hardware Description of Genetic Algorithm Accelerator

The DUT implements the full GA pipeline using a clean **FSM-based control flow**, parameterized by population size and gene length. It performs selection, crossover, and mutation entirely in hardware using register arrays, synthesized using OpenLane on Sky130. Each state is cycle-accurate and

deterministic, with per-character mutation based on pseudo-random conditions. The use of internal gene arrays (population, parent, child, mutant) makes the implementation both modular and scalable. The logic ensures elitism (best genes are preserved) and prevents regression by comparing mutant fitness with the weakest existing population members.

Testbench Architecture

The genetic algorithm module is verified using a **SystemVerilog self-checking testbench**, structured to drive clock, reset, and monitor key outputs (done, best_gen, and fitness_percent) during multiple independent evolution runs. The testbench defines a configurable simulation environment, where the clk is generated using a periodic toggling block (10 ns period), and rst is asserted for a random number of cycles to introduce initialization variability. The Design Under Test (DUT) is instantiated with parameters MAX_POP = 10 and LEN = 12, targeting the evolution of 12-character ASCII strings towards the phrase "Hello World!".

The testbench supports **functional coverage tracking** via a covergroup that samples the genetic algorithm's internal state and final fitness_percent at every positive edge of the clock. This enables the testbench to monitor the entire FSM transition path (INIT → SELECT → CROSS → MUTATE → UPDATE → DONE) and ensures that all fitness ranges (low, mid, high, and perfect) are exercised over time. Each simulation run logs the cycle count required for convergence, and the final evolved gene is printed to the console, allowing visual and programmatic validation of output correctness.

Coverage Metrics and Validation Approach

To verify the design's robustness, the testbench executes **10 randomized test runs**, each starting with a fresh reset and a newly seeded population. During each run, a **fitness histogram** (coverage_hit[0:100]) tracks which percentage values are reached throughout the mutation cycles. This ensures that a wide variety of evolutionary outcomes are explored, increasing the likelihood of catching rare edge cases and exercising all mutation paths. A fatal assertion is also included to detect illegal fitness values (>100%), enforcing correctness of the fitness calculation logic under mutation and crossover conditions.

After all runs complete, the testbench reports both the **functional coverage percentage** (based on FSM state and fitness bins) and the **distribution coverage**, showing how many of the 101 possible fitness levels (0–100) were reached. This dual-layer validation provides a strong metric for how effectively the design's evolutionary space is being exercised.

Cocotb testing and optimization

The mutation module (mutation_sv.sv) was thoroughly verified using Cocotb, a Python-based coroutine-driven cosimulation testbench framework. The testbench (test_mutation_sv.py) initiates a simulation where it supplies a known 12-character input gene ("Hello World!") to the DUT (Device Under Test), along with two randomly generated sequences: one for per-character mutation thresholds and another for potential ASCII replacement characters. The verification logic uses a coroutine clock generator, which simulates a 10ns clock by toggling clk every 5ns, and applies all inputs including child_in, rand_mask, rand_ascii, and mutation_thresh to the DUT synchronously with the clock and reset cycles.

Once the start signal is asserted, the DUT begins processing. The testbench then waits up to 100 clock cycles, polling the done signal which indicates mutation completion. After the output (mutant_out) is ready, it is unpacked from the flattened format and compared to the original string to observe the mutation behavior. As seen in the simulation result, the original gene "Hello World!" was mutated to "Hello {or1!", demonstrating the module's ability to alter individual characters under controlled randomness.

Cocotb provides detailed logs confirming successful simulation startup, input application, and output capture. The summary at the end of the terminal shows that the test passed (test_mutation_sv PASSED) with a simulation time of 155 ns and zero real-time overhead, indicating efficient execution. The report also includes mutation performance metrics like ATIO throughput, though not relevant to correctness.

Overall, the Cocotb testbench validates the correct function of the mutation module under probabilistic mutation logic and confirms that the FSM correctly transitions from start to done, modifying gene characters based on the LFSR-generated thresholds. This confirms functional correctness and test coverage for hardware-in-the-loop evolutionary computation, crucial for integrating into larger chiplet-based GA systems.

Openlane

The synthesis and implementation of the mutation_sv hardware module were carried out using the OpenLane toolchain, a fully open-source digital ASIC design flow built around the Sky130 130nm process node. The specific run used the classic OpenLane flow, tagged as RUN_2025-06-11_18-46-38. The flow starts with RTL elaboration of the SystemVerilog design file (mutation_sv.sv), followed by synthesis (via Yosys), floorplanning, placement, clock tree synthesis (CTS), routing, and various signoff checks. The clock was defined on port clk with a target period of 10 ns, aligning with a nominal frequency goal of 100 MHz.

The floorplan configuration specified an absolute layout size, with a die area of 3 mm × 3 mm and a core area constrained to 5.75 mm², using the pin_order.cfg to guide physical pin placement. The mutation_sv design was successfully synthesized and placed with no fatal routing or design rule check (DRC) violations, demonstrating that the logic was both functionally and physically clean through all stages of the OpenLane flow.

Category	Metric	Value
Toolchain	OpenLane	Classic Flow
Technology	Foundry Process	Sky130 (130 nm)
Run Tag	—	RUN_2025-06-11_18-46-38
Die Area	—	9 mm² (3 mm × 3 mm)
Core Area	—	5.75 mm²

Category	Metric	Value
Cell Count (Std + Fill)	—	83,330 std + 493,410 fill
Utilization	—	~1.98%
Power (Total / Leak)	—	3.96 µW total / 0.34 nW leak
Worst Setup Slack (Best)	—	+5.87 ns
Setup Violations (Worst)	—	138
Max Frequency Estimate	—	~170 MHz
Max Fanout Violations	—	3
Slew/Cap Violations	—	0
DRC / LVS / Magic Errors	—	0
Antenna Violations	—	3
Total Wirelength	—	212,214 units
Worst IR Drop	—	0.62 mV
Lowest Observed Voltage	—	1.799 V

Results

```
C:\Users\sirim\AppData\Local\Programs\Python\Python313\
python.exe C:\Users\sirim\wksp_python\GA1.py
Target Word : Hello World!
Max Population : 10
Mutation Rate : 0.2
-----------------------------------------------
The BestFitness Time
-----------------------------------------------
_@7v} bEf:4u     8.33%   0:00:00.122981
_@7l= KXhhs6     16.67%  0:00:00.170754
H@7lQ 4Ef:4u     25.0%   0:00:00.239611
H@AlQ SXhhd6     33.33%  0:00:00.243318
H@7l; WXnhd6     41.67%  0:00:00.288127
HZ}lQ Wonhd6     50.0%   0:00:00.774712
H@lle Wonhd6     58.33%  0:00:00.909114
H@lle Wonhd!     66.67%  0:00:00.930347
H@ll/ Worhd!     75.0%   0:00:02.077200
Helle Worhd!     83.33%  0:00:02.116109
Helle World!     91.67%  0:00:02.173780
Hello World!     100.0%  0:00:06.813398

=========== Benchmarking Summary ===========
Execution Time: 6.8142 seconds
Peak Memory Usage: 1037.02 KB
============================================
         221826 function calls (221646 primitive calls
) in 6.816 seconds
```

Fig3: Python Results

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     8176    5.859    0.001    6.332    0.001 GA1.py:47(mutation)
       10    1.183    0.118    1.183    0.118 {built-in method _imp.create_dynamic}
    32714    0.544    0.000    0.544    0.000 GA1.py:12(calculate_fitness)
     8176    0.265    0.000    0.542    0.000 GA1.py:38(crossover)
        1    0.157    0.157    8.613    8.613 GA1.py:79(main)
    16584    0.098    0.000    0.098    0.000 {method 'join' of 'str' objects}
    39003    0.071    0.000    0.071    0.000 {built-in method builtins.chr}
     10/6    0.062    0.006    0.571    0.095 {built-in method _imp.exec_dynamic}
    16429    0.051    0.000    0.051    0.000 {built-in method builtins.max}
    16375    0.045    0.000    0.095    0.000 GA1.py:69(bestfitness)
       48    0.043    0.001    0.117    0.002 <frozen importlib._bootstrap_external>:1624(find_spec)
      202    0.029    0.000    0.047    0.000 <frozen importlib._bootstrap_external>:101(_path_join)
     8188    0.026    0.000    0.026    0.000 {built-in method builtins.round}
       80    0.025    0.000    0.025    0.000 {built-in method nt.stat}
33283/33281    0.019    0.000    0.019    0.000 {built-in method builtins.len}
```

Fig4: Cprofiling results

```
# KERNEL: --- Run 10 ---
# KERNEL: eDt>B -y9!dv  16.00%  Cycle: 4
# KERNEL: eDE>O ]ytldN  25.00%  Cycle: 84
# KERNEL: eDtlB ]yQlda  33.00%  Cycle: 568
# KERNEL: weRlB ]ytldN  41.00%  Cycle: 636
# KERNEL: weRlP ]yrldN  50.00%  Cycle: 856
# KERNEL: HeRlB ]yrldN  58.00%  Cycle: 872
# KERNEL: HellO ]yrldN  66.00%  Cycle: 1848
# KERNEL: HellO WGrld[  75.00%  Cycle: 2680
# KERNEL: Hello W&rld[  83.00%  Cycle: 4144
# KERNEL: Hello Wsrld!  91.00%  Cycle: 6832
# KERNEL: Hello World!  100.00% Cycle: 14036
# KERNEL: Best Generation: Hello World!
# KERNEL: Final Fitness: 100%
# KERNEL: Run completed in 14038 cycles.
# KERNEL:
# KERNEL: ======== Coverage Summary ========
# KERNEL: Fitness Distribution Coverage: 12 of 101 levels hit (11.88%)
# KERNEL: [âœ“] Functional coverage = 75.00%
# RUNTIME: Info: RUNTIME_0068 testbench.sv (107): $finish called.
# KERNEL: Time: 1896355 ns,  Iteration: 1,  Instance: /genetic_algorithm_tb,  Process: @INITIAL#58_1@.
# KERNEL: stopped at time: 1896355 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
# ACDB: Covergroup Coverage data has been saved to "fcover.acdb" database.
# VSIM: Simulation has finished.
Done
```

Fig5: Coverage Results

```
MODULE=test_mutation_sv TESTCASE= TOPLEVEL=mutation_sv TOPLEVEL_LANG=verilog \
         sim_build/Vtop
     -.--ns INFO     gpi                                ..mbed/gpi_embed.cpp:108  in set_program_name_in_venv        Using Python vir
tual environment interpreter at /home/sirim/week8/cocotb-env/bin/python
     -.--ns INFO     gpi                                ../gpi/GpiCommon.cpp:101  in gpi_print_registered_impl       VPI registered
     0.00ns INFO     cocotb                             Running on Verilator version 5.020 2024-01-01
     0.00ns INFO     cocotb                             Running tests with cocotb v1.9.2 from /home/sirim/week8/cocotb-env/lib/python
3.12/site-packages/cocotb
     0.00ns INFO     cocotb                             Seeding Python random module with 1748926348
     0.00ns INFO     cocotb.regression                  pytest not found, install it to enable better AssertionError messages
     0.00ns INFO     cocotb.regression                  Found test test_mutation_sv.test_mutation_sv
     0.00ns INFO     cocotb.regression                  running test_mutation_sv (1/1)
Original gene : Hello World!
Mutated gene  : Hello {orl|!
   155.00ns INFO     cocotb.regression                  test_mutation_sv passed
   155.00ns INFO     cocotb.regression                  ****************************************************************************
**************
                                                      ** TEST                             STATUS  SIM TIME (ns)  REAL TIME (s)  R
ATIO (ns/s) **
                                                      ****************************************************************************
**************
                                                      ** test_mutation_sv.test_mutation_sv   PASS         155.00           0.00
  57385.59  **
                                                      ****************************************************************************
**************
                                                      ** TESTS=1 PASS=1 FAIL=0 SKIP=0                    155.00           0.01
  17449.17  **
                                                      ****************************************************************************
**************
```

Fig6: Cocotb Results

```
(nix:devshell-env) [nix-shell:~/ch18/openlane2/designs/GA/runs/RUN_2025-06-11_18-46-38]$ cat ./54-openroad-stapo
m_tt_025C_1v80/power.rpt

====================================================================
 report_power
====================================================================
======================= nom_tt_025C_1v80 Corner ====================================

Group                  Internal    Switching      Leakage        Total
                          Power        Power        Power        Power (Watts)
----------------------------------------------------------------------
Sequential           4.921695e-04 1.837708e-05 1.051229e-09 5.105476e-04  15.5%
Combinational        5.702988e-04 1.073371e-03 3.805187e-09 1.643674e-03  50.0%
Clock                5.022957e-04 6.279699e-04 2.751107e-07 1.130541e-03  34.4%
Macro                0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00   0.0%
Pad                  0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00   0.0%
----------------------------------------------------------------------
Total                1.564763e-03 1.719719e-03 2.799667e-07 3.284762e-03 100.0%
                            47.6%        52.4%         0.0%
```

Fig7: Power Results

```
(nix:devshell-env) [nix-shell:~/ch18/openlane2/designs/GA/runs/RUN_2025-06-11_18-46-38]$ cat ./54-openroad-stapostpnr/summary.rpt
┏━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━┳━━━━━━━━━━━┳━━━━━━━━━━━┳━━━━━━━━━━━┳━━━━━━━━━━━┳━━━━━━━━━━━┳━━━━━━━━━━━┳━━━━━━━━━━━┳━━━━━━━━━━━┳━━━━━━━━━━━┓
┃                    ┃ Hold      ┃ Reg to    ┃           ┃           ┃ of which  ┃ Setup     ┃           ┃           ┃           ┃ of which  ┃
┃                    ┃ Worst     ┃ Reg       ┃           ┃ Hold Vio  ┃ reg to    ┃ Worst     ┃ Reg to    ┃ Setup     ┃ Setup Vio ┃ reg to    ┃
┃ Corner/Group       ┃ Slack     ┃ Paths     ┃ Hold TNS  ┃ Count     ┃ reg       ┃ Slack     ┃ Reg Paths ┃ TNS       ┃ Count     ┃ reg       ┃
┡━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━╇━━━━━━━━━━━╇━━━━━━━━━━━╇━━━━━━━━━━━╇━━━━━━━━━━━╇━━━━━━━━━━━╇━━━━━━━━━━━╇━━━━━━━━━━━╇━━━━━━━━━━━╇━━━━━━━━━━━┩
│ Overall            │ -0.1958   │ 0.1292    │ -0.4584   │ 4         │ 0         │ -2.0460   │ -2.0460   │ -44.2601  │ 138       │ 138       │
│ nom_tt_025C_1v80   │ 0.3473    │ 0.3473    │ 0.0000    │ 0         │ 0         │ 3.6154    │ 3.6154    │ 0.0000    │ 0         │ 0         │
│ nom_ss_100C_1v60   │ -0.0479   │ 0.9043    │ -0.0479   │ 1         │ 0         │ -1.5062   │ -1.5062   │ -28.0653  │ 46        │ 46        │
│ nom_ff_n40C_1v95   │ 0.1324    │ 0.1324    │ 0.0000    │ 0         │ 0         │ 5.6764    │ 5.6764    │ 0.0000    │ 0         │ 0         │
│ min_tt_025C_1v80   │ 0.3426    │ 0.3426    │ 0.0000    │ 0         │ 0         │ 3.8819    │ 3.8819    │ 0.0000    │ 0         │ 0         │
│ min_ss_100C_1v60   │ 0.0784    │ 0.8957    │ 0.0000    │ 0         │ 0         │ -1.0693   │ -1.0693   │ -16.4619  │ 35        │ 35        │
│ min_ff_n40C_1v95   │ 0.1292    │ 0.1292    │ 0.0000    │ 0         │ 0         │ 5.8685    │ 5.8685    │ 0.0000    │ 0         │ 0         │
│ max_tt_025C_1v80   │ 0.3524    │ 0.3524    │ 0.0000    │ 0         │ 0         │ 3.2635    │ 3.2635    │ 0.0000    │ 0         │ 0         │
│ max_ss_100C_1v60   │ -0.1958   │ 0.9130    │ -0.4584   │ 3         │ 0         │ -2.0460   │ -2.0460   │ -44.2601  │ 57        │ 57        │
│ max_ff_n40C_1v95   │ 0.1360    │ 0.1360    │ 0.0000    │ 0         │ 0         │ 5.4335    │ 5.4335    │ 0.0000    │ 0         │ 0         │
└────────────────────┴───────────┴───────────┴───────────┴───────────┴───────────┴───────────┴───────────┴───────────┴───────────┴───────────┘
```

Fig8: Summary of results

Related Work

The proposed genetic algorithm (GA) hardware-software co-design architecture is inspired by emerging trends in AI/ML-specific chiplet accelerators, which aim to improve scalability, modularity, and energy efficiency of compute-intensive workloads. Recent work in modular chiplet-based designs, such as Intel's Foveros and AMD's Infinity Fabric, shows that splitting AI workloads across specialized, domain-specific processing units can drastically reduce data movement and optimize compute locality. In contrast to monolithic accelerators like Google's TPU or NVIDIA's Tensor Cores, chiplet designs support heterogeneous integration, making them suitable for evolutionary algorithms where different stages (mutation, selection, fitness evaluation) benefit from diverse computational characteristics.

Prior literature, such as the work by Yazdanbakhsh et al. on SIGMA (a systolic array-based GA accelerator), highlights the importance of pipelining and hardware parallelism for accelerating GAs. Our design differs by adopting a hybrid strategy, where software performs high-level orchestration, and the mutation stage—a stochastic and character-by-character intensive task—is offloaded to a synthesized FSM in hardware. This partitioning addresses Heilmeier's question: *"What's new?"*—the novelty lies in our tight integration of RTL mutation hardware with Python control logic, optimized via co-simulation and OpenLane synthesis on Sky130. Additionally, our work responds to *"Who cares*

and why?" by targeting embedded or low-power systems needing on-chip intelligent behavior, such as genetic adaptation in sensor networks or evolving encryption schemes in hardware firewalls.

Future Work

While the current design meets timing and area goals and demonstrates functional correctness through waveform validation and coverage metrics, several limitations offer opportunities for future enhancement. One key limitation is the serial execution of selection, crossover, and mutation phases, which, while synthesizable and easy to verify, prevents full exploitation of parallelism inherent in population-based algorithms. A potential improvement would be to partition the design further into parallel processing chiplets, each evolving a subset of the population and periodically exchanging genetic material—a concept aligned with *island-model GAs*.

Moreover, the current design uses basic pseudo-random mutation and a fixed 20% mutation rate. Future iterations could incorporate adaptive mutation logic or integrate fitness-based dynamic mutation control, which are common in advanced evolutionary computing literature. On the software side, future enhancements could include integrating with runtime FPGA frameworks like Xilinx Vitis AI or Python-to-RTL bridges like PyRTL, enabling runtime hardware reconfiguration for different fitness functions.

Finally, for scalability to larger datasets or longer gene sequences (e.g., protein folding, scheduling, or cryptographic key evolution), memory architecture improvements like on-chip scratchpad buffers, memory banking, and cache-aware mutation logic could be explored. These enhancements would support broader deployment in real-world AI edge devices, making the system more robust and adaptive under dynamic workload demands.

Acknowledgement of AI Tools

This project benefited significantly from the use of OpenAI's ChatGPT-4 for iterative technical guidance. The assistant was used to:

- Draft and debug Verilog and Python logic.
- Design GA hardware architecture.
- Explore synthesis strategies, verification flows, and tool-specific configurations (e.g., OpenLane, Cocotb).
- Review and refine documentation sections such as abstract, related work, and future outlook.

Prompts and ChatGPT Guidance

These prompts reflect the iterative, dialog-driven engineering process enabled by generative AI tools, aligning with modern research practices that emphasize rapid prototyping, automation, and continuous learning.

Prompt	Purpose
"Can you give me a simple chiplet diagram of my HW accelerator of genetic algorithm?"	Visualize architecture partitioning
"Generate architecture documentation explaining chiplet partitioning and trade-offs."	Create structured architecture documentation
"Give a detailed paragraph of HW details of genetic algorithm based on my SystemVerilog code."	Convert RTL code into descriptive paragraphs
"Give detailed paragraph on cocotb usage and verification and results and code flow."	Describe verification methodology using Python+Cocotb
"Give a paragraph on HW-SW Co-design of my genetic algorithm."	Articulate the interaction between software and hardware parts of the system
"Verification methodology documentation and HW description is crucial..."	Write detailed testbench analysis, coverage metrics, and validation strategy
"Generate related work section and future work in paragraph form."	Position the project within the current research context and propose improvements
"Explain the genetic algorithm Python implementation flow in a paragraph."	Summarize software-side behavior in plain English
"Generate OpenLane synthesis report paragraph with area, timing, and power analysis table."	Document physical implementation and chip metrics
"Include an acknowledgement for the use of AI tools and mention prompts as examples of vibe coding."	Formally recognize ChatGPT's assistance in the creative development workflow