FPGA-based A* Pathfinding Accelerator

Changwe Musonda
Electrical and Computer Engineering at California State Polytechnic University, Pomona
cbmusonda@cpp.edu

Kevin Wang
Electrical and Computer Engineering at California State Polytechnic University, Pomona
kevinwang@cpp.edu

Emily Morales
Electrical and Computer Engineering at California State Polytechnic University, Pomona
emorales11303@gmail.com

Abstract*—Pathfinding algorithms are widely used in robotics, navigation systems, and artificial intelligence applications. With efficiency as our primary focus, our project seeks to compare the performance of the A* search algorithm implemented on the Nexys A7 100T FPGA and in Python. The Python implementation provides a flexible and accessible baseline, while the FPGA design makes use of hardware parallelism to accelerate computation. For a fair comparison, both implementations use the octile heuristic to guide the search process in a 16x16 grid with varying obstacles. Metrics such as execution time and cycle counts were used to analyze the performance of each platform. Experimental results demonstrate that the FPGA-based solution performed faster and more efficiently compared to the Python implementation, but it also requires greater development effort and hardware resources. The findings from our experiment highlights the trade-offs between running a program solely through software and executing it with the addition of hardware. This information will provide insight on efficient solutions in real-time pathfinding applications.

Index terms - A* search, FPGA, octile heuristic, real-time pathfinding

I . INTRODUCTION

Pathfinding is a common issue we encounter in robotics, navigation, and artificial intelligence. It is vital to find the shortest path efficiently for real-time decision making. Although there are multiple algorithms that can be used for pathfinding, we decided to adopt the A* search algorithm due to its balance in optimality and efficiency when computing. We'll also be applying an octile heuristic formula to model 8-directional movements in a grid-based environment. This heuristic allows the algorithm to provide us with better distance estimations in a two-dimensional grid compared to a simpler heuristic such as the Manhattan distance heuristic. With the addition of diagonal movement, the octile heuristic removes unnecessary expansions and improves efficiency. This makes it better suited for our project.

Software implementations of A* are commonly developed in high-level languages such as Python. Although high-level languages offer flexibility, readability, and easy testing, these implementations of A* often suffer from performance limitations when applied to larger grids or time-sensitive applications. In contrast, hardware acceleration using Field-Programmable Gate Arrays (FPGAs) allows multiple operations to run simultaneously while ensuring consistent results. This will enable FPGAs to potentially execute faster and reduce latency. The Nexys A7-100T FPGA that we are using in this project provides substantial logic resources, block RAM, and DSP slices. This makes it suitable for implementing computationally intensive algorithms such as A*. Our project presents a comparative study of A* search implemented on the Nexys A7-100T FPGA and in Python. By analyzing metrics such as execution time and cycle counts, we're able to identify and determine which platform is a more efficient solution for real-time pathfinding applications.

II . RELATED WORKS

Study on hardware acceleration has become a common topic as researchers look for means to improve the efficiency of computationally intensive algorithms. GeeksforGeeks [1] explains the benefits of specialized hardware such as GPUs, FPGAs, and ASICs in computer vision applications. Additionally, they also noted that CPUs often struggle to meet real-time requirements. With the use of FPGAs, there is an increase in reconfigurability and energy efficiency, making them suitable for tasks that demand deterministic performance in robotics and navigation.

The distinction between CPUs and FPGAs has been further analyzed by Reflex CES [2], who highlight the advantages of FPGA technology in terms of parallelism, reduced latency, and scalability. Unlike CPUs, which execute instructions sequentially, FPGAs can be tailored to specific workloads, enabling simultaneous processing and optimized resource usage. This adaptability allows engineers to design hardware that evolves with application requirements, which is particularly valuable in mission-critical systems.

Intel [3] provides a broader comparison of CPUs, GPUs, and FPGAs within the context of heterogeneous computing. Their analysis shows that CPUs excel at general-purpose tasks, GPUs at high-throughput parallel workloads, and FPGAs at customizable, low-latency operations. This comparison underscores the unique positioning of FPGAs as accelerators that balance energy efficiency with deterministic execution, making them attractive for real-time applications such as pathfinding. While prior work has focused on hardware acceleration in domains like computer vision and signal processing, fewer studies have directly compared FPGA-based implementations of A* search with software baselines in Python. This paper addresses that gap by presenting a systematic evaluation of both approaches, highlighting the trade-offs between rapid prototyping in software and high-performance execution in hardware.

Ⅲ. SYSTEM ARCHITECTURE

A. Python Implementation

The software baseline was implemented in Python to perform A* search on a 16×16 grid with eight-connected movements. The algorithm initially used a Manhattan heuristic, but we transitioned to the octile heuristic for more efficient pathing. To accurately model traversal, we assigned a cost of 10 for cardinal moves and 14 for diagonals. Obstacles were represented in binary maps, and the search maintained open and closed sets with a priority queue for node selection. Parent references were used to reconstruct the path once the goal was reached. In addition to computing metrics such as nodes expanded, path length, path cost, and computation time, the Python implementation generated grid visualizations of the path, obstacles, start, and goal positions, which were used to qualitatively verify correctness across multiple test scenarios.

B. FPGA Implementation

The hardware design was implemented in Verilog on the Nexys A7-100T FPGA. A finite state machine controlled the A* process, with states corresponding to initialization, node selection, neighbor expansion, cost updates, path reconstruction, and completion. Node costs, parent references, and open and closed set membership were stored in hardware arrays, enabling deterministic updates and parallel access. Neighbor expansion considered all eight directions, with costs of 10 and 14, and updated scores and parents when shorter paths were found. Once the goal was reached, the path was reconstructed in hardware and marked in a grid map. Performance metrics including nodes expanded, path length, path cost, and cycle counts were recorded, with a timeout mechanism ensuring deterministic execution.

Nexys A7-100T	
Artix-7 Part	XC7A100T-1CSG324C
Logic Slices	15,850 (4 6-input LUTs & 8 flip flops each)
Block RAM	4,860 Kbits
Clock Tiles	6 (each with PLL)
DSP Slices	240
Internal Clock	450 MHz+
DDR2 Memory	128 MiB

Figure 1. Table containing the total specifications of the Nexys A7-100T FPGA.

C. Experimental Setup

Seven structured test cases were constructed to evaluate performance under different conditions. These included a simple diagonal path, a spiral maze, dense random obstacles, a snake-like pattern, rooms and corridors, a long diagonal corridor, and a scenario with no valid path due to a complete wall. Both implementations used identical obstacle maps and start and goal positions to ensure fairness. The Python version measured computation time in milliseconds, while the FPGA design reported cycle counts directly from hardware registers. Since our FPGA's internal clock was set to 100MHz, we'll also be able to calculate the total time taken for execution. Metrics were collected across all test cases, and Python visualizations provided qualitative verification of correctness alongside quantitative results. This setup enabled a direct comparison of

software flexibility against hardware acceleration under consistent conditions.

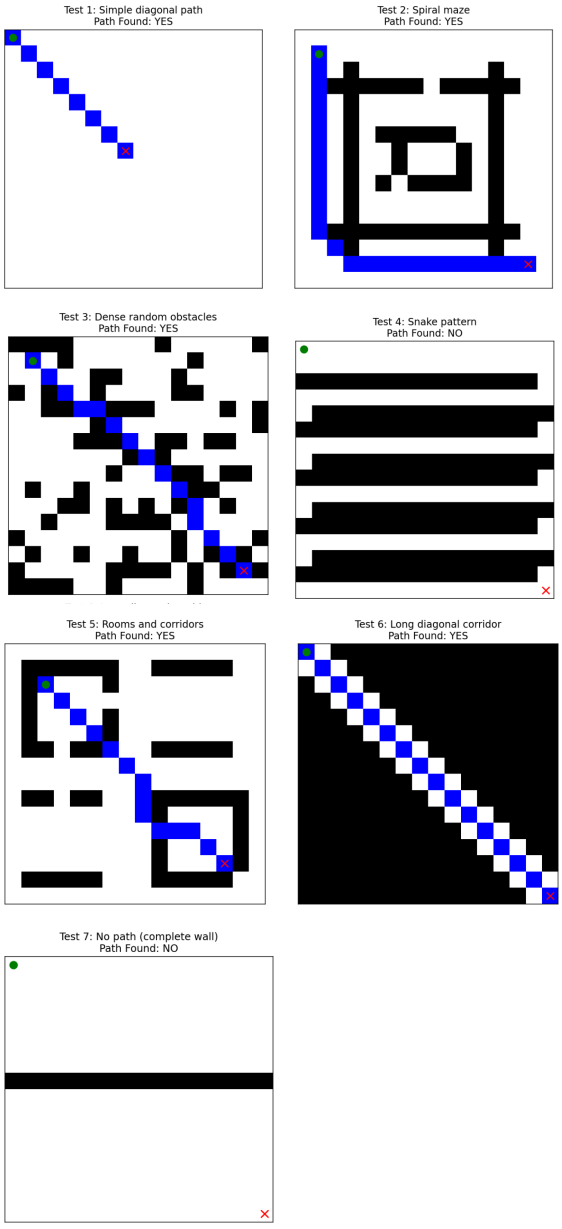

Figure 2. Visualization of each of the seven test cases generated in Python. A green dot represents the starting point, while a red X represents the goal. Obstacles are in black, and the final path is expressed as blue tiles.

IV. RESULTS AND COMPARISONS

The performance of the Python and FPGA implementations was evaluated across six structured test cases, each designed to highlight different pathfinding challenges. In all cases where a solution existed, both platforms produced paths of identical cost, confirming algorithmic correctness. The Python implementation provided visual confirmation of path validity, while the FPGA design reported equivalent metrics directly from hardware registers.

Test Case	Path Found	Path Cost	Nodes Expanded Python	Nodes Expanded FPGA	Python Time (ms)	FPGA Cycles
Simple Diagonal	YES	98	8	8	0.112	2457
Spiral Maze	YES	248	116	54	1.136	15172
Snake Pattern	NO	-	50	50	0.376	14059
Rooms and Corridors	YES	166	38	19	0.36	5499
Long Diagonal Corridor	YES	210	16	32	0.357	9089
No Path (complete wall)	NO	-	112	107	1.566	0

Figure 3. Table of data for this project. The dense random obstacles test case was excluded from this table as it did not represent a static benchmark and produced non-deterministic results across runs.

Execution speed differed significantly between platforms. The Python implementation measured computation time in milliseconds, whereas the FPGA reported cycle counts, which were converted to time using a 100 MHz clock (1 cycle = 10 ns). Across all test cases, the FPGA consistently achieved lower execution time. For example, in the spiral maze test, Python required 1.136 ms while the FPGA completed in 0.152 ms. In the rooms and corridors test, Python took 0.36 ms compared to the FPGA's 0.055 ms. Even in the snake pattern and no-path scenarios, where no solution was found, the FPGA terminated faster due to its deterministic timeout mechanism.

Minor differences were observed in the number of nodes expanded. In the spiral maze, Python expanded 116 nodes while the FPGA expanded only 54. Conversely, in the long diagonal corridor, the FPGA expanded more nodes (32 vs. 16). These variations are attributed to differences in heuristic formulation and tie-breaking behavior. The Python version used the octile heuristic, while the FPGA design employed a scaled Manhattan heuristic. Both are admissible, ensuring optimal path cost, but they guide the search differently. Additionally, the FPGA counter includes the goal node in its expansion tally, which may account for slight discrepancies.

Overall, the results demonstrate the trade-off between flexibility and performance. Python offers rapid prototyping and visualization, making it ideal for development and debugging. The FPGA design delivers deterministic execution and significantly lower latency, showcasing the benefits of hardware acceleration for real-time applications. Despite minor differences in search behavior, both implementations consistently produced correct and comparable results, validating the integrity of the comparison.

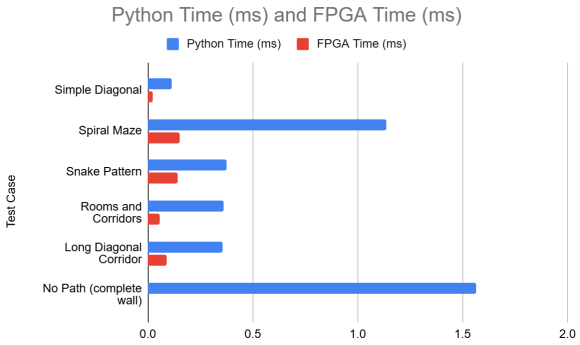

Figure 4. Chart comparing the time taken for the A* algorithm in each test case.

V. CONCLUSION

This work presented a comparative study of A* search implemented in Python and on an FPGA across a set of structured benchmark test cases. Both platforms consistently produced correct paths with identical costs, confirming algorithmic validity despite minor differences in node expansion counts. These discrepancies were attributed to differences in heuristic formulation and tie-breaking behavior, yet they did not affect the optimality of the solutions.

Performance analysis demonstrated the clear advantage of hardware acceleration. Operating at a 100 MHz clock, the FPGA achieved execution times significantly lower than those of the Python implementation, with improvements most pronounced in complex scenarios such as spiral mazes and corridor environments. The deterministic nature of FPGA execution further ensured predictable timing, while Python offered flexibility, rapid prototyping, and visualization capabilities that facilitated development and debugging.

Together, these findings illustrate the trade-off between accessibility and efficiency: Python provides an adaptable environment for algorithm design and verification, whereas FPGA deployment delivers superior speed and determinism for real-time applications. Future work may extend this comparison to larger grid sizes, alternative heuristics, or hybrid CPU–FPGA systems, enabling deeper exploration of how algorithmic design interacts with platform constraints.

REFERENCES

[1] GeeksforGeeks, “Hardware acceleration for computer vision algorithms,” *GeeksforGeeks*, Jul. 23, 2025.

[2] C. Lavaine, “Why use FPGA instead of CPU ? - reflex ces,” *Reflex Ces*, Jun. 12, 2024.

[3] Intel, “Compare benefits of CPUs, GPUs, and FPGAs for different OneAPI compute workloads,” Nov. 09, 2022.