Hindawi
International Journal of Reconfigurable Computing
Volume 2020, Article ID 8896386, 11 pages
https://doi.org/10.1155/2020/8896386

Research Article

FPGA Implementation of A* Algorithm for Real-Time Path Planning

Yuzhi Zhou, Xi Jin, and Tianqi Wang

University of Science and Technology of China, Hefei, China

Correspondence should be addressed to Xi Jin; jinxi@ustc.edu.cn

Received 26 March 2020; Revised 1 August 2020; Accepted 7 August 2020; Published 17 August 2020

Academic Editor: Jose A. Boluda

The traditional A* algorithm is time-consuming due to a large number of iteration operations to calculate the evaluation function and sort the OPEN list. To achieve real-time path-planning performance, a hardware accelerator's architecture called A* accelerator has been designed and implemented in field programmable gate array (FPGA). The specially designed 8-port cache and OPEN list array are introduced to tackle the calculation bottleneck. The system-on-a-chip (SOC) design is implemented in Xilinx Kintex-7 FPGA to evaluate A* accelerator. Experiments show that the hardware accelerator achieves 37–75 times performance enhancement relative to software implementation. It is suitable for real-time path-planning applications.

1. Introduction

Path planning on grid maps is still an important problem in many modern domains, such as robotics [1], vessel navigation [2], and commercial computer games [3]. In some applications, the path-planning algorithm needs to run in real-time performance. For example, mobile robotics and real-time strategy (RTS) games operate in a highly dynamic map where obstacles and roads can change suddenly. In such cases, the maps cannot be loaded in advance to generate initial paths. What is more, the paths generally must be solved in milliseconds due to a large amount of path planning or replanning requests. Therefore, real-time path planning is needed.

A-star (or A*) search algorithm [4] is one of the most widely used heuristic path-planning algorithms on grid maps. It generates the global optimal paths dynamically and can theoretically guarantee the convergence of the global optimal solution [5]. Such characteristic makes it suitable for dynamically changed maps such as real-time path planning in robotics or RTS games.

However, the software A* algorithm's performance is not real-time due to a large number of iteration operations. Lots of previous works attempted to overcome this by lowering the number of cells to be expanded [6]. But the performance still cannot reach real-time performance. For example, Yao et al. [7] made the searching steps reduced from 200 to 80 (reduced to 40%) but the searching time only reduced from 4.359 s to 2.823 s (reduced to 65%).

This paper introduces a hardware framework to accelerate the performance of the A* algorithm by parallelizing the iteration operations. The scientific code could benefit from executing on accelerators like field programmable gate arrays (FPGAs) [8]. The calculation bottleneck mainly focuses on two parts, calculating and sorting the evaluation value of each node. The evaluation value is used to determine the next searching steps. To make them parallel, a scalable array-based architecture that contains eight parallel processing lines was designed. Each processing line is responsible for one searching direction. The architecture was implemented in Xilinx Kintex-7 FPGA and compared to the software algorithm. FPGAs offer high flexibility to Application-Specific Integrated Circuit (ASIC) when implementing the algorithm with a high degree of parallelism [9, 10]. Results show that 37–75 times performance enhancement could be achieved with the accelerator's clock frequency at 100 MHz.

This research makes the following contributions:

(1) The exploration of the way to accelerate the performance of the A* algorithm by parallelizing the iteration operations.

(2) The architecture design of the hardware accelerator for the A* algorithm. Efficient 8-port cache design is achieved to load the nodes data of 8 directions in parallel. The most suitable parameters were decided by the design space exploration. An array-based OPEN list architecture was proposed to achieve sorting while data is flowing in the array.

(3) Evaluation of the system-on-a-chip (SOC) design in the FPGA circuit board. Experimental results show that parallelizing the iteration operations of A* algorithm can achieve massive performance enhancement, and the hardware design is suitable for applications with real-time performance requirements.

The rest of this paper is organized as follows. Related work is discussed in Section 2. The A* algorithm is analyzed in Section 3 to show the performance bottleneck. Then the hardware accelerator is introduced in Section 4 to tackle the bottleneck. Section 5 introduces the system design, and experiments to analyze this work are devised in Section 6. In the end, concluding remarks are drawn in Section 7.

2. Related work

2.1. Heuristic Path-Planning Algorithm. The goal of path planning is to find the most direct and shortest path from the starting point to the target point according to the terrain and obstacles in the map. Global path-planning algorithms have been applied frequently and widely because of their advantages in computation time and avoidance of local optimum [11]. The most well-known algorithm of this type is Dijkstra's algorithm [12]. It finds the shortest path by traveling from the start cell to the neighboring cells and calculates the path's cost. Then it chooses the lowest cost cell to travel again until the target cell is reached. The defect of Dijkstra's algorithm is that nearly all the cells are expanded before the shortest path is found. The A* algorithm [4] improved this by adding a heuristic value to evaluate the path's cost function during the iteration of choosing the next cell. The heuristic value can lead the search path towards the goal. Then Focused Dynamic A* (D*) algorithm [13] and D* lite algorithm [14] were proposed to extend the ability to cope with dynamic changes in the graph used for planning. They have been used for path planning on a large assortment of robotic systems [15–17]. Anytime Dynamic A* (AD*) [18] uses an inflation factor to get to a suboptimal solution quickly, meeting real-time requirements. Liu et al. [2] further improved the A* algorithm for more complicated environments.

To make A* algorithm converge more efficiently, Szczerba et al. [19] proposed a sparse A* search (SAS). This algorithm accurately and efficiently "prunes" the search space according to the constraint, which lowers the number of cells to be expanded. Block A* [20] is a database-driven search algorithm. They load the map in advance and calculate the Local Distance Database (LDDB) that contains distances between boundary points of a local neighborhood. Then the search process is based on blocks. This method effectively lowers the number of iteration operations and achieves about 4x performance enhancement compared to A* algorithm, but it needs the map to be loaded in advance. Yao et al. [7] proposed a way of weight processing of evaluation function to reduce the number of cells to be expanded. They made the searching steps reduced from 200 to 80 (reduced to 40%) but the searching time only reduced from 4.359 s to 2.823 s (reduced to 65%).

Those previous works can be summarized as improving the performance of A* by lowering the number of iteration operations or expanding nodes. But the performance enhancement is not so obvious. In this paper, we tried another way of parallelizing the iteration processing of expanding the cells by hardware accelerator in FPGA.

2.2. FPGA Implementation of Path-Planning Algorithms. One of the most popular path-planning algorithms implemented in FPGA is the genetic algorithm (GA) [21]. The GA method is based on Darwin's theory of evolution, where crossovers and mutations can generate better populations. However, the evolution process needs numerous iteration operations. A continuous research activity during the past 20 years proves the effectiveness of hardware acceleration by parallelism. For example, Allaire et al. [22] accelerate the genetic operators on FPGA and achieve up to 50,000x performance enhancement in the population update operation. Hachour [23] shows the FPGA implementation for the GA path planning of autonomous mobile robots. dos Santos et al. [24] achieve the parallelism by array-based architecture and achieve obvious performance enhancement.

Lots of researchers have also investigated into the FPGA implementation of heuristic path-planning algorithm. Fernandez et al. [25] proposed a parallel architecture for implementation of Dijkstra's algorithm. The node processor architecture was introduced to achieve parallelism for iteration process. For a 256 graph, the computation takes only 42 microseconds, which shows that FPGA implementation can achieve real-time performance. Jagadeesh et al. [26] also implemented Dijkstra's algorithm on FPGA and achieved 2.2 times performance enhancement compared with CPU. Idris [27] proposed the hardware architecture of accelerator for A* algorithm but did not provide simulation result. Nery et al. [28] provided the coprocessor design based on Xilinx High-Level Synthesis (HLS) compiler and achieved 2.16x speedup for A* algorithm. However, they did not tackle the bottleneck problem of sorting OPEN list.

3. Algorithm

In order to achieve parallelism, the performance bottleneck of A* algorithm needs to be analyzed. The traditional A* algorithm will first be introduced and then the performance bottleneck problems will be analyzed for hardware implementation.

3.1. A Algorithm on Grid Maps.* The traditional A* algorithm was first proposed in [4] and targeted for determining the minimum cost path through a graph. It is also suitable for finding the minimum cost path from the start node to the destination node on grid maps. Grid maps are a standard simplified model of real maps, commonly used in mobile robotics [20]. Grid maps are made up of square nodes, and each node stands for a step when moving in the map. An example is shown in Figure 1.

In this paper, we assume that a specific node on the grid map is only allowed to reach one of its eight neighbors. That is, the angle of the path is confined to a 45- or 90-degree turn. Previous works also researched about paths with any-angle turn [29, 30]. But it is not the critical point of this paper. The flowchart of the traditional A-star algorithm is shown in Algorithm 1.

The open_list (which is defined as OPEN list in the manuscript) in the algorithm is an array that contains the nodes to be calculated. The goal is to choose the next step which has the minimum cost value from current node to the destination. This process is called expanding nodes. The evaluation value f is the heuristic value to estimate the distance. It is calculated as

$$f(n) = g(n) + h(n). \tag{1}$$

where g(n) is the actual cost from start node N(x_s,y_s) to current node n and $h(n)$ is the heuristic function that estimates the cost from current node n to destination $N(x_g, y_g)$. The heuristic function chooses Euclidean distances.

$$h(n) = \sqrt{\left(x_i - x_g\right)^2 + \left(y_i - y_g\right)^2}. \tag{2}$$

3.2. Algorithm Analysis. On a grid map, an individual cell is able to move to one of its eight closest neighboring nodes (successors). From Algorithm 1, it can be seen that the iteration operations focus on calculating the successor's evaluation function, the process of which is called expanding nodes.

In order to analyze the bottleneck problem in the process of expanding nodes, we performed experiments to monitor the execution time. The software algorithm was written in C language and executed in a single thread. After that, the software algorithm was optimized to run in parallel 8 threads. The compiler is MSVC on the windows platform and more detailed description will be listed in Section 6.3. The 256 × 256 grid map with 10% randomly placed obstacles is used for the experiments. The results of software were obtained on Inter(R) Core(M) i5-3337U @ 1.80 GHz with 4 GB memory. We ran the software code 10,000 times and averaged the execution time. The experiments' results are shown in Figure 2. The process was divided into two parts. The "OPEN list" process included the process of inserting, sorting, and deleting the OPEN list and the "calculation" process included the other calculations of expanding nodes.

When running in the single thread, the averaged execution time is 330 ms. The operations of OPEN list consume 95% of total time. After distributing the software process into 8 threads, the total execution time is 136 ms, with 2.4x speedup. However, the OPEN list operations are still the most time-consuming part.

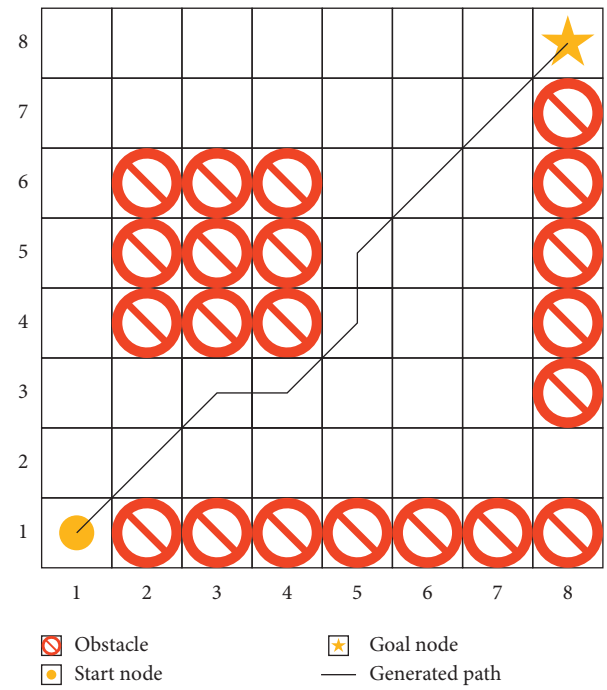


Figure 1: An example of the grid map and optimization path generated by traditional A* algorithm.

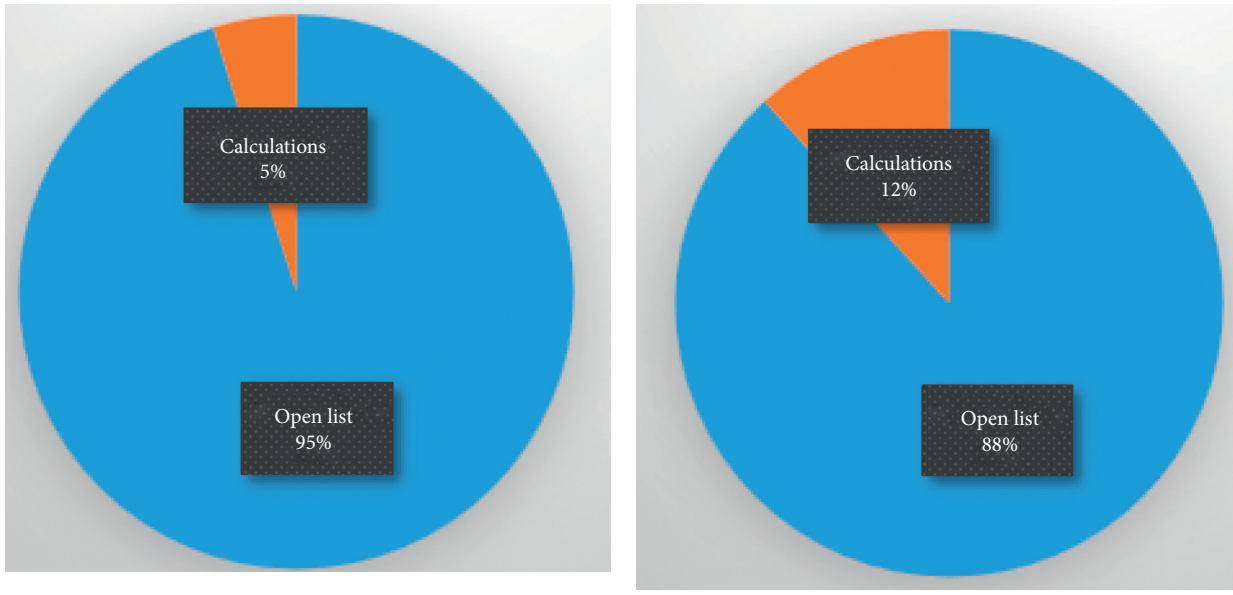
To tackle the bottleneck problems of the OPEN list operations (inserting, sorting, and deleting nodes), the hardware architecture is designed to have 8 parallel processing lines. Each line is responsible for one direction. The OPEN list is an array-based architecture that contains 8 parallel sequence queues, which are called OPEN list array. The OPEN list array will sort the data in parallel.

4. Design of the A* Accelerator

The hardware architecture is designed to tackle the bottleneck problems of the A* algorithm discussed in Section 3.2. Although it is targeted for FPGA implementation in this manuscript, it is also suitable for ASIC chip implementation.

4.1. Data Structure of the Nodes. The information of a node includes parents' coordinates, actual cost value g, evaluation value f, and information about whether this node is an obstacle or in the OPEN list. Since the heuristic function value h is calculated for each node, it is not necessary to be stored along with the nodes. The grid map in this manuscript is confined to smaller than 256 × 256. The extension to bigger maps will be discussed in the future work.

Under such situation, the data type of cost value and evaluation value are designed as signed integer. The data structures are concluded in Table 1. The 1-bit information is combined with cost value to form a 32-bit integer value. Then, the total size of a node's data structure is 10 bytes.



(a) (b)

Figure 2: Execution time of software code. (a) The software runs in a single thread. (b) The software runs in 8 threads.

```
Initialize grid maps with obstacle matrix
Mark N(x_s, y_s) as open_list
while open_list ≠ empty set do
  find the node in open_list with least value f marked as N(x_s, y_s)
   mark N(x_i, y_i) as open_list
  if N(x_i, y_i) = N(x_g, y_g) do
    Construct the path
    return "path is found"
  else
    Mark N(x_i, y_i) as close_list
    generate N(x_i, y_i)'s 8 successors
    for each successor do
      if successor does not belong to close_list or obstacle_node do
        calculate successor's value f and marked as f_new = f (successor)
        If f_new is lower than successor's original value f or successor is not in open_list
          mark successor as open_list
          successor.f = f_new
          set successor's parent to N(x_i, y_i)
        end if
      end if
    end for
  end if
end while
```

Algorithm 1: Traditional A* algorithm (Start $N(x_s,y_s)$ to $N(x_g,y_g)$).

4.2. Hardware Framework Design. An overall hardware framework is shown in Figure 3. The grid map is initialized and stored in the memory. After the start node is loaded, the nodes management module calculates the successors' coordinate and read data information into 8 parallel evaluation engines. The cells' information is transferred to the accelerator through the nodes cache. The evaluation engine computes the value f of the nodes according to the evaluation function and sends it to the OPEN list array. OPEN list array is a sequence queue where the node with the lowest f is on the top of the array. The comparison engine will compare the 8 values in the evaluation engine with the 8 top values in the OPEN list array and pop out one node with the least value f to the nodes' management module. The path is found when the nodes' management module reads the coordinate of the goal node. The path cannot be found if all 8 OPEN list arrays are empty.

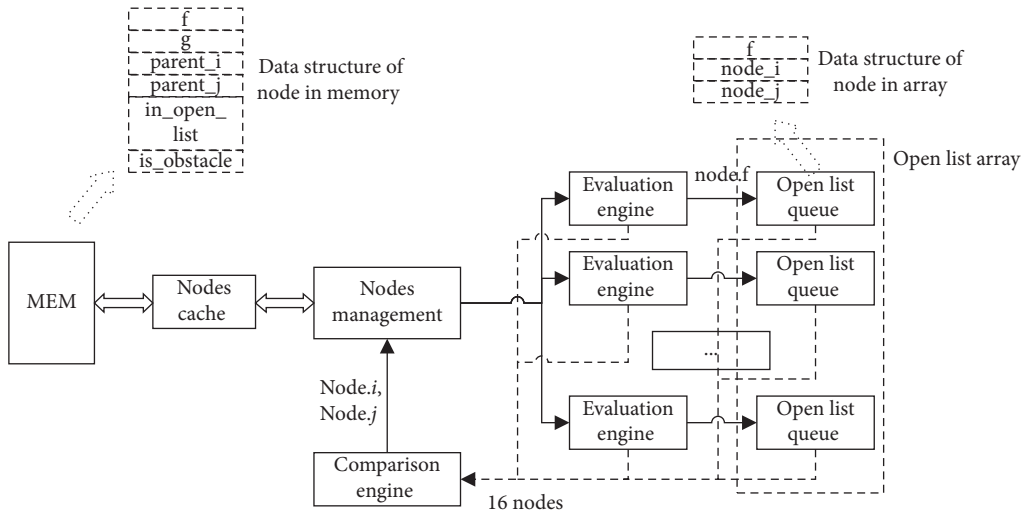


Figure 3: Accelerator's framework for A* algorithm.

The two most critical modules are nodes cache and the OPEN list array. Since the nodes' management module handles 8 nodes in parallel, the nodes cache must transfer 8 nodes' information in one cycle if not missed. So, it is designed as an 8-port cache. In addition, the efficiency of sorting nodes in the OPEN list array determines the throughput of the accelerator.

4.3. Design of the Cell Cache. To load successors' information in parallel, an 8-port cache is needed. On a grid map, the successors of a certain node are all in a 3×3 square. According to this characteristic, one cache line is designed to store a square block of $N \times N$ nodes. The size of the cache line is determined by design space exploration (DSE). The block-based arrangement is shown in Figure 4.

The worst situation is when the nodes are in the corner of the block, and the successors are divided into 4 neighbor blocks. Therefore, the 8-port cell cache consists of 4 banks to achieve reading blocks in parallel. The bank selection is determined by the lower bits of block coordinate.

$$\text{bank Selector} = \left\{\text{block}_j[1], \text{block}_i[1], \text{block}_j[0], \text{block}_i[0]\right\}. \tag{3}$$

This bank selection strategy will ensure that the four parallel requirements will request different cache lines in the same cache. An example of the worst case is shown in Figure 5.

The detailed architecture is shown in Figure 6. The 8 addresses will be transferred to cache in parallel and distributed to different banks by interconnect crossbar. The reading address of the same block will fall on the same cache line, so the reading requests will be merged. When the data of one bank is missed, it will write data back if it is "dirty" and read data from memory. The cache miss penalty time is the clock cycles of writing back and reading. If the data in the bank is available, it will be distributed to the port according to the requests. The cache is connected to memory controller by AXI bus and the buses data size is 64 bits.

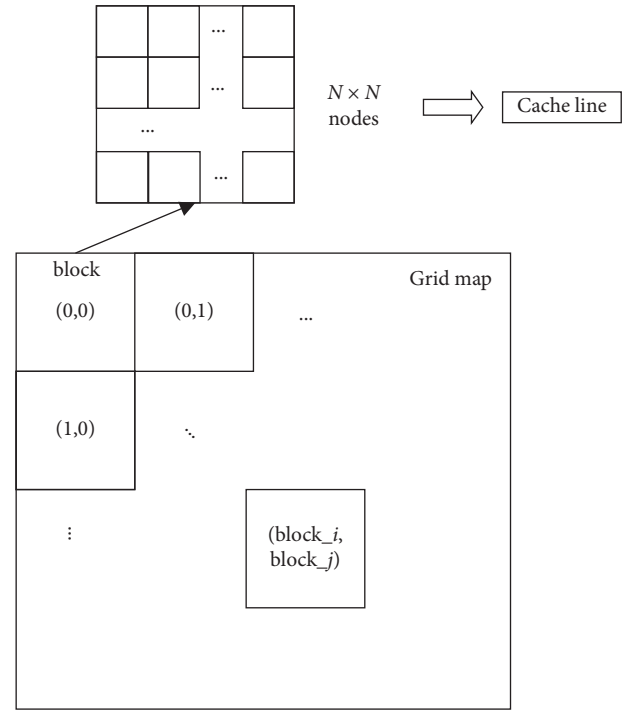


Figure 4: Block-based arrangement for nodes in cache. Each block contains $N \times N$ nodes.

The mapping algorithm of the cache is designed as 2-way set associative. Direct mapping is the easiest mapping algorithm, but it is not suitable for A* algorithm implementation. A* algorithm is a heuristic path-planning algorithm. Therefore, the process of reading nodes is random. Two different nodes in the same cache line will seriously affect the performance. Fully associative mapping algorithm will increase the design complexity of the cache. In order to balance between cache size and performance, we choose 2-way set associative.

The trade-off between the cache size and performance is determined by design space exploration.

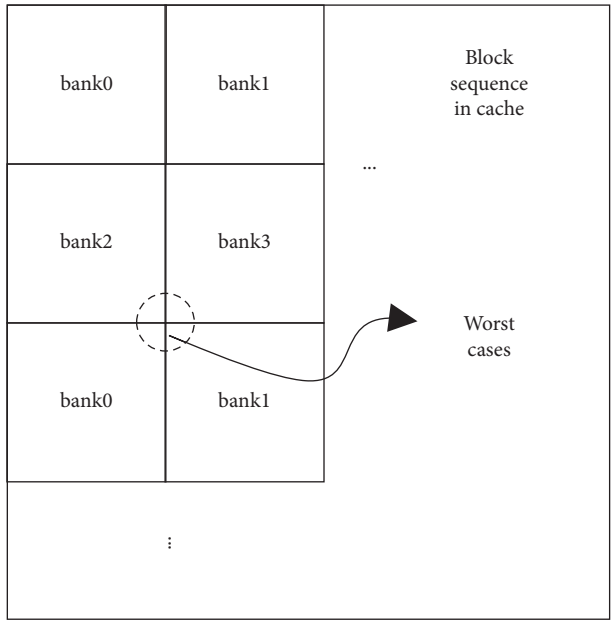


Figure 5: An example of cache mapping strategy. The worst case is when a missing node falls on the corner of the bank and its neighbors fall on 4 banks.

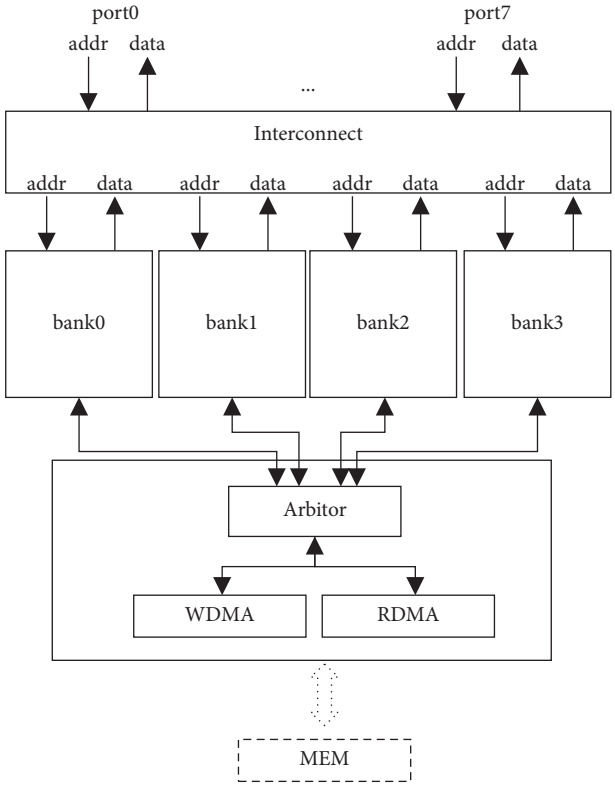


Figure 6: Detailed architecture of the cell cache.

4.4. Design of the OPEN List Array. The OPEN list array is composed of 8 parallel sequence queues called OPEN list queues. Each queue contains input buffer (IB) structure to store the input data and output buffer (OB) for nodes ready for output. The overall architecture is shown in Figure 7. The nodes in OB

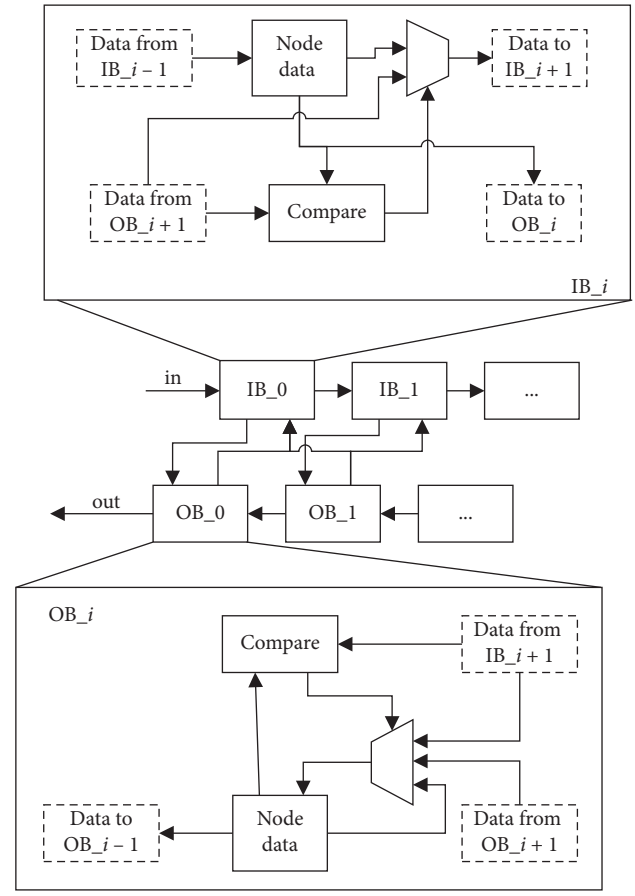


Figure 7: Detailed architecture of the OPEN list queue.

are sequenced by evaluation function's value f. The input node is sent to IB when a new node is inserted into the OPEN list.

The sorting process is similar to the bubble sort algorithm. When the input node is flowing in IB and reaches the position IB_i, it will compare with the next OB buffer's position OB_{i+1}. If the value f of the node in IB_i is lower than that in OB_{i+1}, these two nodes will be swapped and the node of OB_{i+1} will be sent to the next buffer IB_{i+1}. In this way, the larger OB's value will be swapped into IB.

Another situation is when the head of OB pops from the queue, it leaves a "bubble" in position OB_0. Then the corresponding IB cell IB_0 needs to compare with the next OB cell OB_1. If the value f of IB_0 is lower than OB_1, the node in IB_0 will be put in the position of IB_0. Otherwise, the node in OB_1 will be put in that position and the bubble shifts right in OB buffer.

Figure 8 shows an example of the above two processes. At cycle 1, node 3 is inserted into IB and needs to swap with node 5 in OB. At cycle 2, node 3 takes the position of OB_1 and node 5 goes to IB_1. Assume that the next node inserted is node 4 and node 2 pops out. Then node 3 will take the position of OB_1. At cycle 3, node 4 reaches the position of IB_1 and will take the bubble position. At cycle 4, the inserted node finds the correct position.

4.5. Design of the Other Modules. The node with the lowest value f must be the one of the nodes produced by the evaluation engine or on the head of the OPEN list queue.

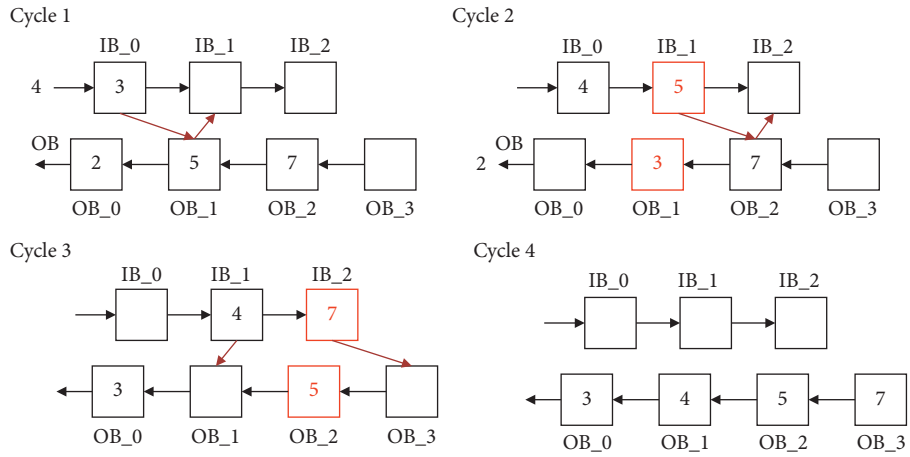


Figure 8: An example of inserting and popping out nodes in the OPEN list.

Therefore, the comparison engine module will compare the above 16 nodes and choose the node for the nodes' management module to expand.

The nodes' management module gets the node's coordinate from the comparison module and transfers the nodes' information from the node cache to 8 parallel evaluation engines.

The evaluation engine computes the value f of each node and inserts it into the OPEN list array. The computation of evaluation value contains the root operations, which is done by the lookup table.

5. System Design

As described in the above section, the performance bottleneck lies in the data fetching and sorting efficiency. In this section, we perform the overall optimization problem as maximizing the system throughput under resource constraints. We demonstrate the design's implementation to maps with 256×256 resolution, and it is scalable to bigger maps. Then the hardware system-on-a-chip (SOC) will be introduced briefly to show the implementation process in field programmable gate array (FPGA).

5.1. Analytical Model of Cache. The system throughput is determined by cache performance. We built a high-level model (C++) to perform the design space exploration (DSE) to identify hardware parameters with maximum system throughput. The design space includes two dimensions.

(1) The cache line size (marked as N). Higher cache line size will lower the miss rate of the cache. But it increases the possibility of loading useless nodes and cache misses rate, which affects the cache miss penalty.

(2) The number of cache lines in a bank (marked as M). Higher number of blocks is better, but it increases the size of the cache. Therefore, some design option needs to be "pruned" due to the limitations of memory size on FPGA. The memory size should be less than 1 MB.

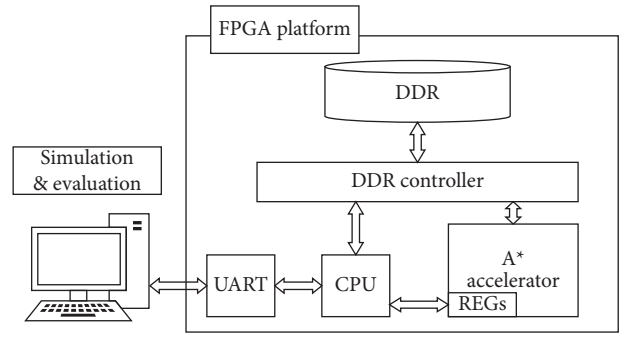


Figure 9: Diagram of the experimental platform.

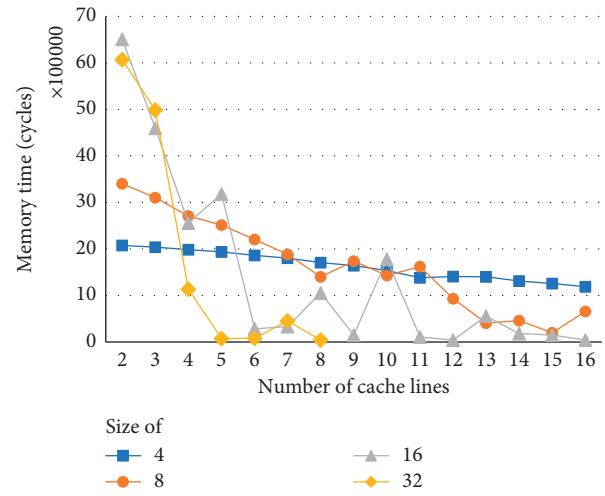


Figure 10: Design space exploration for the nodes cache. The size of cache line indicates the number of nodes in one cache line. The y coordinate indicates the clock cycles of memory time when cache misses.

(3) Mapping algorithm. Direct mapping algorithm is not suitable for the situation of reading different nodes into the same cache line. Fully associative algorithm

Table 1: Design of the data structure.

Name	Data type	Explanation
g	Signed integer	The cost value from the start node to current node
h	Signed integer	The estimated cost value from current node to the goal node
parent_i	Unsigned short	Horizontal coordinate of parent's node
parten_j	Unsigned short	Vertical coordinate of parent's node
in_open_list	Bool	Whether current node is in the OPEN list
is_obstacle	Bool	Whether current node is obstacle

makes the structure too complex to implement. Therefore, we choose 2-way set associative to balance between cost and performance.

The total number of nodes in the cache is calculated to be $N \times M$. The size of cache is $N \times M \times$ sizeof (node). The number of cache misses is marked as m. The cache miss penalty is marked as T. The cache's performance is modeled by memory time MT.

$$\begin{aligned} MT &= \text{number\_of\_cache\_miss} * (\text{block\_size} \\ &\quad + \text{penalty\_cycle\_from\_MEM}) = m \times (N * N + T). \end{aligned} \tag{4}$$

Although it is not the actual cache miss penalty time, it can be used to evaluate the trend of design options. The software model runs on 256×256 resolution maps with obstacles randomly generated. The experimental results will be introduced in Section 6.1.

5.2. System Architecture. A brief diagram of the experimental platform is shown in Figure 9. The A* accelerator was developed by Verilog RTL language and implemented into Xilinx Kintex-7 FPGA. The reason we chose this FPGA is that another project of our team was developed on this FPGA. Other components of the SOC (such as CPU and buses) were also developed in that project. The CPU is based on ARM cortex-M0 ISA which is used to transfer maps' data to DDR and transfer A* accelerators' data to PC. The DDR controller and the A* accelerator are connected by 64-bit wide AXI buses.

The experimental maps were loaded into DDR3 memory in advance. The CPU controls the A* accelerator through 16-bit APB buses. The calculating time (counted in clock cycles) will be read back from REGs and transferred to PC for evaluation. The implementation results will be discussed in Section 6.2.

6. Experiments Results and Discussions

6.1. Design Space Exploration of the Cache. The software model for the design space exploration runs on a 256×256 resolution map with obstacles randomly placed. The start point is fixed to (0, 0) and the goal is fixed to (255, 255). The experiment results of different design options are shown in Figure 10.

From the chart, three options perform relatively better results. The results with cache size are shown in detail (see Table 2). The cache size is calculated by

Table 2: Design option of three best choices.

Design option	Block size M	Block number N	Penalty/cycles	Cache size/KB
1	8	15	196026	38.4
2	16	12	37506	122.88
3	32	5	72380	204.8

Table 3: Summarization of synthesis results.

Cells	Used	Available	Utilization (%)
Slice registers	50930	508400	10
Slice LUTs	134578	254200	52
Block RAMs	14	795	1.8
Maximum frequency	255.188 MHz		

Table 4: Software experimental environment.

OS version	Windows 10 enterprise 1067
Compiler	Microsoft MSVC 16.6
Optimization options	/Od (disable optimization)
Code generation	/Gm (enables minimal rebuild)
Linking	/MDd (compiles to create a multithreaded DLL)
OpenMP	/openmp (enables OpenMP 2.0 language support)

$$\begin{aligned} \text{Cache\_size} &= \text{cache\_line\_size} * \text{block\_number} \\ &\quad * \text{bank\_number} * \text{size\_of\_cell} = M * N * 4 * 10. \end{aligned} \tag{5}$$

Although design option 2 gives the best performance, its size makes it unfeasible for the implementation of most FPGAs. What is more, the performance of design option 1 has already tackled the performance bottleneck of fetching nodes' data. Therefore, the block size is designed to be 8×8 and the block number of each bank is 15.

6.2. FPGA Implementation Results. The A* accelerator was developed by Verilog RTL language and synthesized by Xilinx EDA tool Vivado 2019.2. The FPGA used to implement the whole design is Xilinx Kintex-7 XC7K410T. The synthesis results of the accelerator are illustrated (see Table 3).

The maximum frequency of the A* accelerator is 255 MHz. In order to achieve better timing results, the core's

Table 5: Experimental results of randomly generated maps.

Data set	Cases	Implementation	Distance	Expanded nodes	Time (ms)	Speedup
Random 0%	Worst	Software	357	65527	85	75
		Hardware			1.1367	
	Random	Software	136.5	29720	38.2	74
		Hardware			0.516	
Random 10%	Worst	Software	361.8	45284	76.4	72
		Hardware			1.059	
	Random	Software	140.9	26278	31.2	69
		Hardware			0.467	
Random 20%	Worst	Software	370.2	41202	66.4	61
		Hardware			1.087	
	Random	Software	131.8	7986	21.7	44
		Hardware			0.493	
Random 30%	Worst	Software	374.4	35597	58.4	50
		Hardware			1.160	
	Random	Software	152.0	7543	20	47
		Hardware			0.429	
Random 40%	Worst	Software	392.2	24236	52	45
		Hardware			1.144	
	Random	Software	155.0	6962	19.8	48
		Hardware			0.412	
Random 50%	Worst	Software	433.9	23352	40.4	37
		Hardware			1.088	
	Random	Software	168.1	5882	14.2	42
		Hardware			0.340	

Table 6: Comparison results with related works.

Work	Optimization methods	Time before optimization	Time after optimization	Speedup
Yao et al. [7]	Software	7.879 s	3.061 s	2.58
Yap et al. [20]	Software	4.81 ms	1.03 ms	4.7
Nery et al. [28]	FPGA implementation	—	—	2.16
This paper	FPGA implementation	85 ms	1.1367 ms	75

clock frequency is designed as 100 MHz. The timing constraints come from the combination logic in OPEN list array module. In order to achieve sorting in parallel, the comparison chain is deep. The timing can be further optimized in the further work.

We further analyzed the utilization results. The resource utilization bottleneck falls on Slice LUTs. Moreover, almost all the LUTs are consumed by OPEN list array module to achieve sorting in parallel.

6.3. Performance Enhancement. We compared the software and hardware implementation of A* algorithm using a 256 × 256 grid map filled with randomly placed obstacles, with the probability of a cell being an obstacle ranging from 0% to 50% like [15]. The software algorithm first runs on PC to find whether the shortest path is available. Then the available map will be transferred to the memory on the FPGA circuit board and controls the hardware accelerator for calculation. The results of time and path will be transferred back to the PC for evaluation after FPGA's calculation.

The results of software implementation were obtained on Inter(R) Core(M) i5-3337U @ 1.80 GHz with 4 GB memory. The CPU contains 4 cores and 8 threads. The software code was written in C language and implemented in 8 parallel threads by OpenMP library. The compiler is Microsoft MSVC on Windows platform. More detailed description about the software experimental environment is listed in Table 4. The IDE platform is Visual Studio 2019. The hardware results were obtained on Xilinx Kintex-7 XC7K410T FPGA, synthesized by Xilinx EDA tool Vivado 2019.2.

For each dataset, a total number of 10,000 maps were averaged to form the results (see Table 5). In every map, two situations are tested. In the worst cases, the start point is fixed to (0, 0) and goal point is fixed to (255, 255). In another situation, they are randomly placed on the map.

The results show that 37–75 times performance enhancement was achieved by hardware accelerators. This achievement is mainly because each node calculates the 8 neighboring evaluation values and inserts the OPEN list in parallel. Moreover, the specially designed cache reduces the time of fetching data from memory. Therefore, it is suitable for real-time path-planning applications.

An interesting phenomenon is that performance enhancement decreases when the number of expanding nodes decreases. We further investigated this phenomenon through analyzing the simulation waves and found that most

of the time is during the memory-fetching process. For example, in the random 50% data set, over half of the running time is spent on fetching nodes' data. Therefore, most of the cache's data is wasted because the number of expanding nodes is low.

6.4. Comparison. To compare the performance enhancement, the works of software and hardware implementation of A* algorithm are chosen. Yao et al. [7] proposed the software optimization methods to lower the nodes to be expanded. Their work improved the processing time from 7.879 s to 3.061 s, with 2.58x speedup. Yap et al. [20] proposed block A* algorithm and achieved up to 4.7x speedup. Nery et al. [28] provided the hardware implementation based on Xilinx High-Level Synthesis (HLS) compiler and achieved 2.16x speedup. The comparison results are summarized in Table 6.

The different experimental results are conducted under different situations. Therefore, it is not easy to scale the results for comparison. However, the results of the speedup are a good standard for comparing the effectiveness of the optimization methods.

The proposed architecture achieves great performance enhancement compared to the previous work due to the carefully designed OPEN list array and nodes cache. Nery et al.'s work [28] is also implemented in FPGA, but the execution process is still in serial. Therefore, their work's enhancement is not so obvious.

By the way, from the execution time, it seems that Yap et al.'s work [20] outperforms ours. But their methods need to calculate data related to the grid maps (called LDDB) in advance (costing 1.2 s). It s not suitable for real-time path planning.

7. Conclusions

This article proposed a hardware framework to accelerate the A* path searching algorithm by parallelizing the iteration operations. The 8-port cache is designed to tackle the memory bandwidth bottleneck and OPEN list array to tackle the calculation bottleneck. The proposed architecture shows 37–75 times speedup even at a low clock frequency of 100 MHz. Therefore, it is of research value to implement A* family algorithm for more complicated path-planning applications. The proposed SOC design shows its capability for further implementation as a coprocessor in Application-Specific Integrated Circuits (ASICs).

In the future, the proposed architecture will be investigated to lower the resource consumption of LUTs by optimization of the OPEN list array. The cache will be optimized to adapt to more situations. What is more, the extensible and configurable architecture for general graph applications will be considered in the future work.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The authors would like to thank another group of their laboratory for their help in the implementation of the system-on-a-chip (SOC) when building the experimental platform.

References

[1] S. Long, "Mobile robot path planning based on ant colony algorithm with A* heuristic method," *Frontiers in Neurorobotics*, vol. 13, p. 15, 2019.

[2] C. Liu, Q. Mao, X. Chu, and S. Xie, "An improved A-star algorithm considering water current, traffic separation and berthing for vessel path planning," *Applied Sciences*, vol. 9, no. 6, p. 1057, 2019.

[3] C. Zhang, Y. Tang, and H. Liu, "Late line-of-sight check and partially updating for faster any-angle path planning on grid maps," *Electronics Letters*, vol. 55, no. 12, pp. 690–692, 2019.

[4] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[5] S.-W Dang, F.-J He, and P.-Z Cheng, "A path planning method for indoor robots based on partial & global A-star algorithm," in *Proceedings of the 5th International Conference on Frontiers of Manufacturing Science and Measuring Technology (FMSMT 2017)*, April 2017.

[6] D. Ferguson, M. Likhachev, and S. Anthony, "A guide to heuristic-based path planning," in *Proceedings of the international workshop on planning under uncertainty for autonomous systems, international conference on automated planning and scheduling*, ICAPS, Monterey,CA, USA, June 2005.

[7] J. Yao, C. Lin, X. Xie, J. Wang, and C. C. Hung, "Path planning for virtual human motion using improved A* star algorithm," in *Proceedings of the Seventh international conference on information technology: new generations*, April 2010.

[8] S. W. Nabi and W. Vanderbauwhede, "Automatic pipelining and vectorization of scientific code for FPGAs," *International Journal of Reconfigurable Computing*, vol. 2019, Article ID 7348013, 12 pages, 2019.

[9] G. Dinelli, E. Rapuano, G. Benelli, and L. Fanucci, "An FPGA-based hardware accelerator for CNNs using on-chip memories only: design and benchmarking with intel movidius neural compute stick," *International Journal of Reconfigurable Computing*, vol. 2019, Article ID 7218758, 13 pages, 2019.

[10] H. J. Macintosh, J. E. Banks, and N. A. Kelson, "Implementing and evaluating an heterogeneous, scalable, tridiagonal linear system solver with openCL to target FPGAs, GPUs, and CPUs," *International Journal of Reconfigurable Computing*, vol. 2019, p. 13, Article ID 3679839, 2019.

[11] H. Kim, D. Kim, J.-U. Shin, H. Kim, and H. Myung, "Angular rate-constrained path planning algorithm for unmanned surface vehicles," *Ocean Engineering*, vol. 84, pp. 37–44, 2014.

[12] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[13] A. Stentz, "The focussed d* algorithm for real-time replanning," in *IJCAI. International Joint Conferences on Artificial Intelligence*, vol. 95, Pittsburgh, PA, USA, August 1995.

[14] S. Koenig and M. Likhachev, "Improved fast replanning for robot navigation in unknown terrain," in *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292)*, vol. 1, May 2002.

[15] A. Stentz and M. Hebert, "A complete navigation system for goal acquisition in unknown environments," *Autonomous Robots*, vol. 2, no. 2, pp. 127–145, 1995.

[16] S. Koenig, D. Furcy, and C. Bauer, "Heuristic search-based replanning," in *Proceedings of Sixth International Conference on Artificial Intelligence Planning Systems*, Toulouse, France, April 2002.

[17] M. Hebert, R. MacLachlan, and P. Chang, "Experiments with driving modes for urban robots. mobile robots XIV," in *Proceedings of SPIE -Society of Photo-Optical Instrumentation Engineers*, vol. 3838, Boston, MA, USA, January 1999.

[18] M. Likhachev, "Anytime dynamic A*: an anytime, replanning algorithm," in *Conference on Automated Planning and Scheduling (ICAPS)*, vol. 5, Monterey, CA, USA, June 2005.

[19] R. J. Szczerba, P. Galkowski, I. S. Glicktein, and N. Ternullo, "Robust algorithm for real-time route planning," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 36, no. 3, pp. 869–878, 2000.

[20] P. Yap, N. Bruch, R. Holte, and J. Schaeffer, "Block A*: database-driven search with applications in any-angle path-planning," in *Twenty-Fifth AAAI Conference on Artificial Intelligence*, pp. 120–125, San Francisco, CA, USA, August 2011.

[21] Y. Hu and S. X. Yang, "A knowledge based genetic algorithm for path planning of a mobile robot," in *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 5, IEEE, New Orleans, LA. USA, May 2004.

[22] F. C. J. Allaire, G. Labonté, and G. Fusina, *FPGA Implementation of Genetic Algorithm for UAV Real-Time Path Planning. Unmanned Aircraft Systems*, Springer, Dordrecht, Netherlands, 2008.

[23] O. Hachour, "The proposed genetic FPGA implementation for path planning of autonomous mobile robot," *International Journal of Circuits, Systems and Signal Processing*, vol. 2, pp. 151–167, 2008.

[24] P. V. dos Santos, J. C. Alves, J. C. Alves, and J. C. Ferreira, "An FPGA array for cellular genetic algorithms: Application to the minimum energy broadcast problem," *Microprocessors and Microsystems*, vol. 58, pp. 1–12, 2018.

[25] I. Fernandez, J. Castillo, C. Pedraza et al., "Parallel implementation of the shortest path algorithm on FPGA," in *Southern Conference Programmable Logic*, pp. 245–248, San Carlos de Bariloche, Argentina, March 2008.

[26] G. R. Jagadeesh, T. Srikanthan, and C. M. Lim, "Field programmable gate array-based acceleration of shortest-path computation," *IET Computers & Digital Techniques*, vol. 5, no. 4, pp. 231–237, 2011.

[27] M. Y. I. Idris, "High-speed shortest path Co-processor design," in *Proceedings of the Asia International Conference on Modelling and Simulation*, pp. 626–631, Bandung, Bali, Indonesia, May 2009.

[28] A. S. Nery, A. Da Costa Sena, and L. S. Guedes, "Efficient pathfinding co-processors for FPGAs," in *Proceedings of the Symposium on Computer Architecture and High Performance Computing*, IEEE, Campinas, Brazil, pp. 97–102, October 2017.

[29] K. Daniel, A. Nash, S. Koenig, and A. Felner, "Theta*: any-angle path planning on grids," *Journal of Artificial Intelligence Research*, vol. 39, pp. 533–579, 2010.

[30] A. Nash, K. Daniel, S. Koenig, and A. Felner, "Theta*: any-angle path planning on grids," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 7, American Association for Artificial Intelligence, Vancouver, British Columbia, Canada, July 2007.