Fast RTL Implementation of A* Path Planning Algorithm

Adham Osama[1], Ahmed Mostafa[1], Eslam Mamdouh[1], Mohamed Gamal[1], Usama Imam[1], Mohamed Taha[2], Ahmed Khalil[2], Islam Ahmed[2], Hassan Mostafa[1,3]
[1] Electronics and Communication Engineering Department, Cairo University, Giza 12613, Egypt.
[2] IC Verification Solutions, Siemens EDA, Cairo Egypt.
[3] University of Science and technology, Nanotechnology and Nanoelectronics Program, Zewail City of Science and Technology, October Gardens, 6th of October, Giza 12578, Egypt
{ adhamosama242@gmail.com, Ahmedmostafa8298@gmail.com, emamdouh123@gmail.com, muhamedtawfik@outlook.com, usama.emam02@gmail.com, Mohamed_Taha@mentor.com, Ahmed_Khalil@mentor.com, Islam_Ahmed@mentor.com, hmostafa@uwaterloo.ca }

2021 International Conference on Microelectronics (ICM) | 978-1-6654-0839-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICM52667.2021.9664960

***Abstract*— The conventional A* algorithm consumes a lot of time due to its large number of iterations. In every iteration, the memory is accessed for multiple data structures, functions are evaluated then sorted into queues which makes it sometimes not suitable for real-time applications. This paper proposes a fast implementation for the A* algorithm to meet requirements of real-time applications. The proposed implementation uses parallelism and caching to achieve better performance. We used Register Transfer Level (RTL) simulation and formal verification to do functional verification of the implemented design.**

The design is implemented on Xilinx Virtex-7 to be evaluated. Experiments prove that this implementation achieves 100 times enhancement for low obstacle maps and 50 times for high ones relative to software implementation. The design is suitable for real-time applications.

I. Introduction

Path planning is a computational problem that aims to find a sequence of movement of a certain object to travel between two points in a defined space. In some applications such as autonomous vehicles, the path-planning algorithm needs to find the correct path in real-time since the map is a dynamic map which changes rapidly when any object in the environment changes its location.

Depending on how much information is known about the environment, the path planning could be used in the autonomous real-time systems to predict a guidance path which makes the job of the autonomous driving systems [1] and crash avoidance systems [2] a lot easier than unguided navigation.

There are many path planning algorithms. One of the most notable is the A* algorithms Which is theoretically guaranteed to find the best existing solution for a map in addition to having a heuristic function which acts like a guide to reach the goal in fewer iterations.

This paper introduces a hardware implementation to accelerate the A* algorithm to reach real-time performance by using parallelism, solving memory bottlenecks and using optimized designs for every block in the overall design. The design was implemented on Xilinx Virtex-7 FPGA. The design is implemented in Verilog, synthesized using Xilinx Vivado Design Suite, and verified using UVM and formal verification. The resulting timing constraints were compared to previous implementations of the same algorithm

Yuzhi Zhou et al [3] introduced a hardware design that uses parallelism and implemented on kintex-7 FPGA which had results showing an improvement of 37-75 times performance could be achieved compared to software implementation. However, in this paper the results show that performance of the algorithm can be enhanced by 79-430 times depending on the map that the design deals with.

II. A* Algorithm

A* algorithm, pronounced as A* star algorithm [4], is suitable for determining the least cost path in a grid map. Grid map is formed of nodes, each one can be a square. Reaching any node must be through one of its eight neighbors or what can be called "children nodes". Hence, moving from one node to another can be vertically, horizontally, or diagonally.

To calculate the cost of a node, the value F is calculated using the formula:

$$F(n) = G(n) + H(n)$$

Where G(n) is the accumulative cost from the start node to the current node that is being expanded and H(n) is the heuristic function which gives the algorithm an estimated cost from the current node to the goal node.

There are some well-known heuristic functions that can be used. These functions were compared in [5]. The most popular one is to calculate the Euclidean distance between the current node and the goal node, so:

$$H(n) = \sqrt{dx^2 + dy^2}$$

However, there are some other heuristic functions that must be considered before choosing one. One of these functions is the octile distance function, which is also known as "Chebyshev distance" or "diagonal distance". Octile distance function is calculated as:

$$\mathrm{H(n)} = D * (|dx| + |dy|) + (D2 - 2 * D) * min(dx, dy)$$

Where D and D2 are constant weights, dx and dy are horizontal and vertical distances from current node to goal node. Using this function was more suitable than Euclidean distance for the design introduced in this paper for two main reasons:

1. The number of iterations decreases when using Octile distance as the heuristic function as the design expands a smaller number of nodes and still manages to find the correct path.
2. The evaluation of cost function F(n) is simpler as it can be implemented using conventional blocks like comparator, addition, subtraction and absolute, unlike Euclidean distance function which uses square root block that is more complicated and usually has longer combinational delay.

Authorized licensed use limited to: Hassan Mostafa. Downloaded on March 19,2022 at 23:13:39 UTC from IEEE Xplore. Restrictions apply.

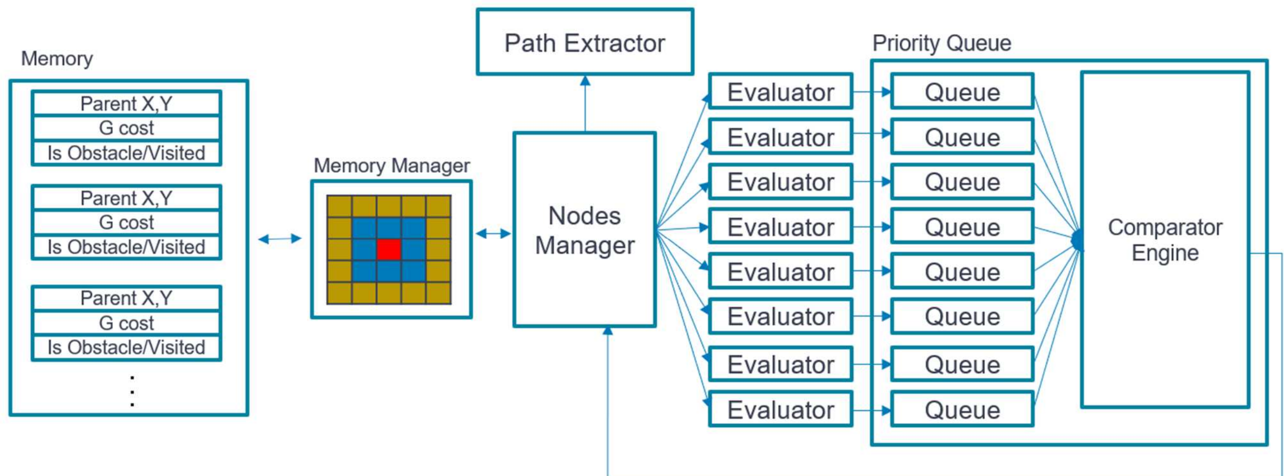

Fig. 1. Full Design of the A* Accelerator

Each iteration, the algorithm computes the F(n) cost for all the neighboring nodes of the current node and then chooses the node with the lowest total estimated cost to be the new "current node". The algorithm saves the data of the node being processed as a parent node for all of the eight children if it provides the lowest G(n) for that child node.

III. Digital Design of A* Accelerator

A. Design Overview

Figure 1 shows the full design of the A* Accelerator. The grid map is initialized in the memory and the start and end nodes are sent to the design so it can start solving the map. The Nodes Manager is the main controller of the algorithm. The memory manager is a cache-like module that is used to overcome the memory access bottleneck which will be discussed in the next section. The design uses parallelism to calculate the F(n) costs of all child nodes at the same time, in the evaluator modules. Each evaluator inserts its results in its corresponding priority queue. The comparator engine selects the best node out of all eight queues and sends it to the Nodes Manager so it can request the needed data from the memory manager and start a new iteration.

B. Memory Access

Previous digital implementations of A* accelerator suffered from memory access bottlenecks. As mentioned before, the algorithm needs eight data structures every iteration for its calculations. Simply giving the algorithm direct access to the memory will limit the design at a certain timing and going lower will be extremely hard. This implementation overcomes this bottleneck by using a cache-like block, called memory manager, that handles the memory reads and writes that the algorithm needs.

The algorithm operates on a 3x3 block of the eight nodes surrounding the current node. Since the algorithm has a heuristic, the next expanded node is usually one of the nodes next to the current node. The memory manager stores a 5x5 block surrounding the same node, Figure 2. This way, when the algorithm moves to a neighboring node, the data it needs is already stored out of memory in the memory manager and it receives the data instantly without any memory access delay.

Figure 3 shows a scenario where the algorithm takes a step in the northern east direction. The node's data marked with purple will instantly be transmitted to the algorithm so it can start a new iteration. In the meantime, the memory manager shifts its internal registers and accesses the memory to return to its original state, Figure 2, being ready for another data request from the algorithm.

In the case of the algorithm moving to a node other than the neighboring ones, the memory manager signals to the algorithm to halt while it accesses the memory to read the needed data. This is equivalent to a cache miss in a microcontroller system.

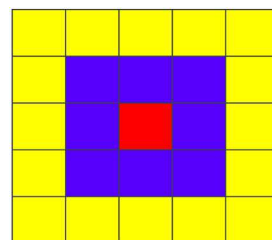

Fig. 2. Data Stored inside Memory Manager. The red node is the node currently being investigated. Blue Nodes are its children which the F cost is being calculated for. The yellow nodes are extra data inside the memory manager that will help overcome the memory access bottleneck

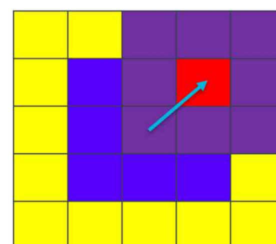

Fig. 3. Purple Node's data sent to the algorithm in the case of northern east movement

Authorized licensed use limited to: Hassan Mostafa. Downloaded on March 19,2022 at 23:13:39 UTC from IEEE Xplore. Restrictions apply.

C. Priority Queues

After calculating the cost of eight children, they are inserted in an open list. The open list is a sorted queue. This sorting operation is a bottleneck in the algorithm itself. To overcome this bottleneck in this implementation, several comparisons were done to choose the most optimum based design and the number of parallel queues.

By comparing different architectures of sorting queues and according to the results in [6], the most optimum design for this implementation would be the one based on the shift register. This queue implementation inserts and sorts any new value in just one cycle. This happens by comparing the new input value with all the stored values in the queue to decide its correct order inside the queue. Comparing the input value with all the queue blocks creates a bus loading problem, which becomes very dominant in large length and limits the maximum operating frequency. This problem makes the decision of the queue length to be very critical.

In the software there is no limitation on the size of the open list, but this is unfeasible for hardware. Therefore, the most suitable size of the queues has to be determined taking into consideration low utilization, clock frequency, parallelism degree and the accuracy of the algorithm. It is important to choose the most suitable size to have the correct functionality and avoid discarding important data, which may prevent the algorithm from finding the shortest path. Sweeping on 10,000 maps for different probabilities of obstacles was done in two different cases: eight parallel queues and four parallel queues. By comparing the results of sweeping according to accuracy, which is the ability of the algorithm to find the shortest path, the chosen length is 313 blocks for each queue in 8 parallel queues. This chosen length achieves 99.6% accuracy and the best performance in terms of maximum clock frequency and minimum number of cycles across the design.

D. Comparator Engine

In each iteration, the algorithm selects the least cost node to be expanded. As there are eight parallel queues in the design, the comparator engine block compares between the top values of all queues and selects the least cost node to be used in the next iteration.

The conventional comparator is a basic arithmetic unit that compares the magnitude of binary numbers and for the algorithm, the comparison mainly focuses on smaller than operations. This technique does not give the best performance as it contains three phases of comparison. It is obvious that the best performance comparator that detects the smallest input in only one phase of comparison and this could be achieved by parallelism of comparison operations and each input is compared to all other inputs in parallel.

The area of the design plays a main role in parallel comparison operation as it increases gradually by increasing the number of inputs and their bit width. A published paper [7] has provided an optimization solution in area by removing each block with a condition smaller than or equal and replacing them with inverters to signals that provide the opposite condition of the removed block. By applying this optimization, about half of the comparison blocks are replaced by inverters.

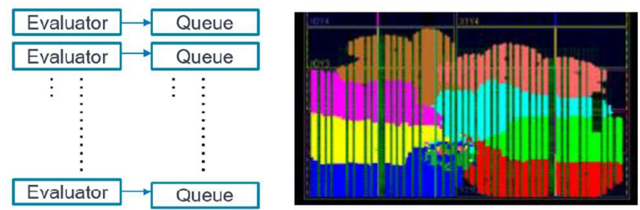

Fig. 4. Pblock Assignment

This is a great solution that provides very high performance with area compared to the conventional comparator.

IV. Function Verfication

The design is verified dynamically using the Universal Verification Methodology (UVM) through a testing environment that reads the start and the goal points from a text file generated by a python script. The environment reads the results generated from the high-level model, and then it passes the start and the goal points to the design and compares the outputs. Because the nature of maps is random, the testing was automated to help in getting the concluding results in Table II.

As the design aims to find the shortest path for a 256×256 map, it has a great number of possible maps, and it cannot be validated completely using simulation. Formal verification helps in validating the design's behavior as it guarantees that the design will operate correctly with any map. The formal verification tools take behaviors and tries to find a counterexample for each behavior searching for bugs. The design is proved using PropCheck ,Mentor Graphics's tool, and verified statically using formal verification by proving 98 assertions and 27 reachable coverpoints.

V. FPGA Implementaion

The A* accelerator was synthesized, placed, and routed by Xilinx EDA tool Vivado 2019.1 targeting Xilinx Virtex-7 FPGA. The performance is improved by limiting the fan-out, changing the default strategies and changing the floorplanning that was done automatically by the tool (Xilinx Vivado Design Suite) and doing it manually as much as possible. Pblocks were assigned for every module and put it near the modules that it is connected. For example, each evaluator writes in a single queue every time, they can be put together in a single Pblock as shown in Figure 4. This results in eight Pblocks that are connected to the comparator engine. Each Pblock's area is double the area of the modules inside it to make routing easier. The Nodes Manager, Memory Manager, and the main memory were put together in the same Pblock. The Comparison Engine with the register holding the current node data together in the same Pblock. This manual floorplanning is then put in the constraints to make the tool restricted with it. Also, slack setup violations don't exist until the tool enters the routing phase. The setup violations appear while the tool is trying to solve the hold violations, therefore the final strategies were chosen to guide the tool to start the early stages with the hold violations in consideration.

During FPGA deployment, the number of I/O pins was not enough, so Vivado's built-in IPs were used to allow the usage of as many I/O pins as needed. These IPs are Virtual I/O (VIO), Integrated Logic Analyzer (ILA) and Clocking wizard.

Authorized licensed use limited to: Hassan Mostafa. Downloaded on March 19,2022 at 23:13:39 UTC from IEEE Xplore. Restrictions apply.

VI. Experimental Results

A. FPGA Implementation Results

The results of the design after placement and routing, using the previously mentioned techniques, are illustrated as shown in table I below. The maximum operational frequency of the A* Accelerator is 200MHz. The total on-chip power is 1.569 Watts.

TABLE I. AREA RESULTS ON XILINX VIRTEX-7

Cells	Used	Available	Utilization
Slice registers	100735	866400	11.63%
Slice LUTs	121222	433200	27.98%
Block RAMs	82	1470	5.58%

B. Performance

The benchmarked results are from trials on a 256×256 map with randomly placed obstacles. To calculate the time needed by the algorithm to finish, the worst case for the algorithm was considered. Selecting the start point at (0,0) and the goal point at (255,255) ensured the highest number of computations for the algorithm which is the worst case. Using the 200MHz clock frequency, the average operating time to finish the algorithm and give outputs for every probability of obstacles based on 1000 different maps for every case was calculated. These timing results are illustrated as shown in table II.

TABLE II. EXPERIMENTAL RESULTS OF RANDOMLY GENERATED MAPS FOR THIS IMPLEMENTATION AND RELATED WORK

Probability of a node being an obstacle	This implementation Time (ms)	Yuzhi Zhou et al [3] Time (ms)
10%	0.198	1.059
20%	0.379	1.087
30%	0.556	1.160
40%	0.765	1.144
50%	1.078	1.088

C. Comparison

In table II the results are compared with related work [1]. This implementation achieves better performance due to the Memory Manager's mechanism of fetching data from the memory. As the probability of a node being an obstacle increases, the timing becomes closer to the older implementation. This is because more obstacles mean more cache misses which leads to halting the design to grab the needed data. Consequently, bringing the implementation closer to direct memory access implementation.

VII. Conclusion

This paper proposed a fast RTL implementation of A* Path Planning algorithm. The design uses parallelism to do the eight calculations and queue insertions at the same time. It also uses a cache-like module to overcome the memory bottlenecks present in other hardware implementations. Using shift registers with the optimal parameters in the internal design of the priority queue led to one cycle read and write with the least area usage available. This implementation shows an average of time enhancement by 50% in solving the map reaching up to five times speedup at maps with low probability of obstacles when comparing it to previous implementations of the algorithm.

This paper proposes a fast implementation that meets requirements of real-time applications which can relax the constrains of the modules in the real-time systems. This give more room for other modules, like V2V Communication [8], to increase their accuracy using the saved path planning time.

VIII. Acknowledgment

This work was partially funded by ONE Lab at Zewail City of Science and Technology and Cairo University, Siemens EDA (Mentor Graphics), ASRT, NTRA, and ITAC

IX. Refrences

[1] M.A. Hassan, M.K. Abbas, A. Osama, D. Anwar, M. Azzam, S. Shafiey, H. Mostafa, and I. Sobh, "GG-Net: Gaze guided network for self-driving car", Society for Imaging Science and Technology, Electronics Imaging – Autonomous Vehicles and Machines (EI-AVM'2021), no. 171, United States, pp. 1-7, 2021.

[2] M. Abdou, R. Mohammed, Z. Hosny, M. Essam, M. Zaki, M. Hassan, M. Eid, and H. Mostafa, "End-to-End Crash Avoidance DeepIoT-based Solution", IEEE International Conference on Microelectronics (ICM 2019), Cairo, Egypt, pp. 103-107, 2019.

[3] X. J. ,. a. T. W. Yuzhi Zhou, "FPGA Implementation of A∗ Algorithm for Real-Time," International Journal of Reconfigurable Computing, p. 11, 2020.

[4] P. Lester, "A* Pathfinding for Beginners," 2005. [Online]. Available: http://www. gamedev. net/reference/articles/article2003..

[5] Red Blob Games, "Heuristics From Amit's thoughts on path finding," [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html.

[6] J. R. a. K. G. S. Sung-Whan Moon, "Scalable Hardware Priority Queue Architectures for High-Speed Packets Switches," vol. 49, no. 11, p. 13, November 2000.

[7] S.-H. P. a. D.-W. K. Young-Ho Seo, "High-level hardware design of digital comparator with multiple inputs," Integration, the VLSI Journal, vol. 68, p. 9, 2019.

[8] A. Hosny, M. Yousef, W. Gamil, M. ADEL, H. Mostafa, and S. M. Darwish, "Demonstration of Forward Collision Avoidance Algorithm Based on V2V Communication ", IEEE International Conference on Modern Circuits and Systems Technology (MOCAST 2019), Thessaloniki, Greece, pp. 1-4, 2019.

.

Authorized licensed use limited to: Hassan Mostafa. Downloaded on March 19,2022 at 23:13:39 UTC from IEEE Xplore. Restrictions apply.