Efficient A* Co-processor for Reconfigurable Gaming Devices

Alexandre S. Nery
Departamento de Engenharia Elétrica
Faculdade de Tecnologia
Universidade de Brasília - UnB
Brasília, Brasil
Email: anery@redes.unb.br

Alexandre C. Sena
Departamento de Informática e Ciência da Computação
Instituto de Matemática e Estatística - IME
Universidade do Estado do Rio de Janeiro - UERJ
Rio de Janeiro, Brasil
Email: asena@ime.uerj.br

***Abstract*—Pathfinding algorithms are at the heart of most games, especially to fulfill increasingly demanding Artificial Intelligence and Level Design tasks. Recent smartphones and tablets are equipped with efficient Multi-Processing Systems-on-Chip (MPSoC) devices, with demanding performance requirements and energy consumption constraints. While not primarily designed for gaming, such mobile machines are quickly climbing to the top of the list of preferred gaming devices, augmented at each new product iteration with state-of-the-art multimedia subsystems and co-processors. Therefore, this work aims at designing and evaluating an efficient A* pathfinding co-processor for reconfigurable gaming devices. The co-processor is designed using Xilinx High-Level Synthesis (HLS) compiler and is implemented in the programming logic of a Xilinx Ultrascale+ Field-Programmable Gate Array (FPGA) embedded with a 64-bit quad-core ARM Cortex-A53 MPSoC, dual-core Cortex-R5 real-time processors, and a Mali-400 MP2 graphics processing unit. Extensive performance, circuit-area and energy consumption results shows that the co-processor running at only 200MHz can efficiently find paths approximately four times faster than one ARM processor running at 1.2GHz for a set of pathfinding benchmarks based on artificial maps and commercial games such as StarCraft and Baldur's Gate, paving the way for novel dedicated gaming co-processors. Moreover, the co-processor only requires about one third of the system's total dynamic power.**

***Keywords*-Pathfinding; FPGA; High-Level Synthesis; Hardware Accelerator;**

I. Introduction

The gaming industry has become one of the most profitable entertainment business, with only the mobile market expected to reach a revenue around $40 billions in 2017 [1]. Most game developers continue to make games primarily for the PC and mobile markets, with an increasing interest for Virtual/Augmented Reality (VR/AR) technologies [2], [3]. Many recent gaming devices, such as smartphones and tablets, are equipped with in-house customized Multi-Processing Systems-on-Chip (MPSoC), often built around ARM architectures. Such integrated circuit design includes in a single chip key advanced co-processors and components, such as GPUs, memories, communication modules, among others, yielding reduced energy consumption. More recently, vendors of Field-Programmable Gate Arrays (FPGAs), such as Xilinx and Altera, have embedded ARM microprocessors around the programmable logic of their reconfigurable chips, allowing the extension of the ARM basic functions. Together with High-Level Synthesis (HLS) compiling tools [4] that are able to translate C code to Register Transfer Level (RTL) Hardware Description Language (HDL), such as VHDL or Verilog, it is not only possible to quickly prototype novel hardware components, but also to offload code execution to efficient and dedicated parallel hardware accelerators implemented on the FPGA-side.

In this paper we propose an efficient A* pathfinding co-processor suitable for reconfigurable (ARM+FPGA) gaming devices compliant with the Advanced Microcontroller Bus Architecture (AMBA4) specification. The co-processor was implemented and evaluated in the programmable logic of a modern Zynq Ultrascale+ Field-Programmable Gate Array (FPGA) from Xilinx [5]. The Zynq Ultrascale+ architecture is embedded with a 64-bit quad-core ARM Cortex-A53 MPSoC, dual-core Cortex-R5 real-time processors, and a Mali-400 MP2 graphics processing unit. Thus, game control, input/ouput logic and graphics processing can be run on the embedded MPSoC-side, while the FPGA programmable logic runs the dedicated performance demanding and energy efficient A* algorithm for the games which may require it. A well-known 2D pathfinding benchmark [6] is used to evaluate the A* co-processor. It features artificial benchmarks and commercial game benchmarks, including maps from famous games such as Baldur's Gate, WarCraft 3 and StarCraft. Each map is a 2D grid (matrix) and must be up to 512×512 wide. The whole system runs a custom Linux OS based on Xilinx Petalinux [7] to facilitate the A* co-processor testing process and to leverage game design in future iterations. Our A* HLS accelerator not only allow a fast execution of the A* pathfinding algorithm, but with low energy consumption, which may help to improve battery life. Despite the gaming system and its A* co-processor being FPGA-oriented, an ASIC (Application-Specific Integrated Circuit) implementation can also easily be produced from it.

The aim of this paper is to show the feasibility of extending the hardware of reconfigurable gaming devices to execute dedicated algorithms in FPGA hardware. It also

evaluates the efficiency of the RTL architecture produced when using Xilinx HLS compiler. Although this work focuses on the A* algorithm implementation in FPGA logic, different algorithms can be implemented in the same way, possibly shifting the gaming industry towards the development and implementation of novel dedicated gaming co-processors. Also, despite the fact that Graphics Processing Units (GPUs) can be reprogrammed to execute different algorithms, their power consumption is often higher than that of FPGAs running dedicated parallel accelerators for the same algorithms. Moreover, the GPU architecture is designed to benefit SIMD-like, data parallel, floating-point demanding applications, while the FPGA configurable logic blocks can be (re-)programmed with just the necessary operations and control logic required by a given application or class of applications. While such FPGA-based gaming devices do not yet exists, there is a recent trend towards reviving classic gaming architectures [8]–[11] in FPGA. Hence, we envision a future where gaming devices hardware ought to be reconfigurable, enabling a whole new market place for hardware accelerators suitable for specific games or a wider range of games.

The rest of this paper is organized as follows: Section II describes the state-of-the-art related works. The reconfigurable gaming system is presented in Section III, together with the implemented A* co-processor. Extensive experimental results are presented in Section IV, including performance, circuit-area and energy consumption results for a set of artificial and commercial (game) maps. Finally, Section V concludes and presents ideas for future work.

II. Related Work

In many games, regardless of the platform, characters need to move within the scenario, whether to patrol around a point of interest, *e.g.*, a tower, or to chase an enemy. In both cases it is possible to simply tie the characters to determined travelling routes. While simple, characters may seem to be roaming aimlessly or may get trapped [12]. Moreover, in more complex games, characters are not supposed to know in advance to which location they should be moving. Instead, they should venture across the scenario in search of resources, enemies, etc. Pathfinding, also known as path planning, is a class of algorithms that can be used to determine the (sub-)optimal route between a starting point and a goal. Thus, they can be used to solve the problems described above, especially in Artificial Intelligence (AI) tasks that continuously seek for suitable routes between players and enemies in the game. Thus, the pathfinding algorithms should execute quickly enough to not stall the character's movement.

Hardware accelerators and co-processors are often used to execute the most timing consuming parts of a specific application or group of applications. Besides graphics [13], [14], state-of-the-art games are more than ever bursting with physics simulation tasks. Thus, Software Developments Kits (SDKs) and Physics Processing Units (PPUs) have been developed in the past few years to ease the burden of physics modelling and to accelerate physics-oriented operations [15]–[17], such as clothing [18], 3D-object collision [19] and model-based robotics [20], [21].

Pathfinding is also a common operation embedded into the AI of many games, networking and route planning applications. An evaluation of pathfind algorithms has been presented in [22]. The paper analyzes the Breadth-first, Depth-First, Ordered, Greedy and A* algorithms on Android platforms. It presents the size of each computed path, their execution time and the number of generated/expanded states, *i.e.*, the number of visited/unvisited cells. Results show that heuristics-based methods, such as A*, are more efficient in terms of execution time.

An implementation of an FPGA Bellman-Ford pathfinding algorithm can be seen in [23]. The architecture distributes the input graph among adjacency RAMs of several Processing Elements (PEs) implemented on a Xilinx Virtex-5 SX95-T FPGA, with each PE in the design being mapped to a node of the graph. The architecture runs at 143MHz for a 128 node and 466 edges graph, taking around 2418 cycles to compute a path on such graph. While fast, the authors assume that the graph topology has already been supplied to each PE. Also, detailed information about the host processor architecture or the communication protocol used among the PEs is not provided, as well as energy consumption results, which makes it hard to compare to the work presented here. Moreover, differently from the work proposed here that executes several distinct benchmarks for games, only four small graphs were executed.

The work in [24] describes an FPGA implementation of the A* algorithm, but it does not present several important results, such as execution time, circuit-area and energy consumption. Also, only the heuristic cost function is implemented on the FPGA, while the rest is expected to run elsewhere.

Differently from all previous work, the work proposed here designs and evaluates an efficient A* pathfinding co-processor for reconfigurable gaming devices. Extensive performance and energy consumption results shows that the co-processor can efficiently execute the A* algorithm approximately 4× faster than the embedded ARM processors when running benchmarks based on commercial games such as StarCraft and Baldur's Gate.

III. The Reconfigurable Gaming System

The reconfigurable gaming system prototype based on FPGA technology is briefly shown in Fig. 1. The key is to enable the extension of the gaming hardware architecture (MPSoC) with any dedicated efficient co-processor specified in High-Level Synthesis (HLS) and implemented on the FPGA-side, possibly saving computation time and energy.

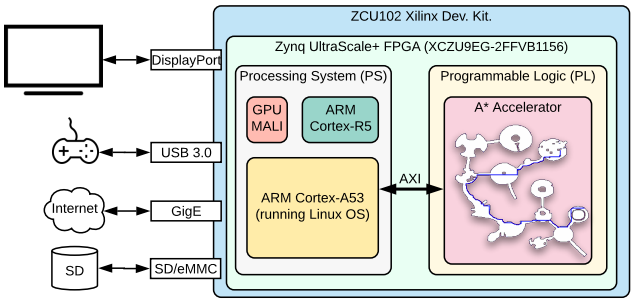


Figure 1. The Reconfigurable Gaming System augmented with the proposed A* Co-Processor.

The game developer community can thus take advantage of such reconfigurable architecture to design accelerators to meet specific performance requirements of a game while also improving the system's battery life. Moreover, once the co-processor has been designed, synthesized and implemented, a *bitstream* file is produced which enables the co-processor to be programmed onto the FPGA on the fly, thus creating a potential new market for game co-processors that can be bundled with the games they are intended to accelerate through well-known digital distribution platforms such as Steam, Apple App-Store and Google Play Store.

In this work we implemented an A* co-processor to demonstrate the gaming system acceleration proposal on a Xilinx ZCU102 Development Kit, which represents the reconfigurable gaming device. It is equipped with a Zynq Ultrascale+ FPGA and several peripheral interfaces, including HDMI DisplayPort, USB3.0, Gigabit Ethernet and SD card. The Zynq FPGA architecture is split into Processing System (PS) and Programmable Logic (PL) parts, also shown in Fig. 1. Besides, the Zynq Ultrascale+ PS integrates a 64-bit quad-core ARM Cortex-A53 MPSoC, a dual-core Cortex-R5 real-time processors, and a Mali-400 MP2 graphics processing unit. The PL part is the reconfigurable logic, which can be programmed with user-created hardware accelerators using Hardware Description Languages (HDL), such as VHDL and Verilog, or High-Level Synthesis (HLS) tools, which are capable of producing Register Transfer-Level (RTL) architectures from a C/C++ code specification, as will be shown in Section III-B.

The PS-PL communication interface operates according to the Advanced eXtensible Interface (AXI4) protocol, which is part of the AMBA4 specification [25]. The Zynq Ultrascale+ PS-PL has several interfaces operating into the Full-Power Domain (FPD) or Low-Power Domain (LPD) interface of the Zynq chip. The FPD-PL interfaces are designed to provide high-throughput data transmission between the PL and the PS. The main FPD-PL interfaces are listed as follows: 2× High-Performance (HP) master ports from the FPD into the PL and 6× High-Performance (HP) slave ports from the PL into the FPD. Other interfaces (*e.g.*, ACP) are recommended for medium-grain sized workloads and adhere to specific cache-coherency policies. The proposed co-processor interface connects to both master and slave HP interfaces of the PS, as will be described in Section III-B.

Algorithm 1 A* pseudo-code

Require: map adjacency array g, start s node, target t node
Ensure: (sub-)optimal path array p, if it exists

include starting node s to open list
while open list is not empty **do**
  get node v with lowest cost in open list
  get neighbors of v
  for each neighbor n of v **do**
    if v is equal to t **then** ▷ target reached
      break ▷ end of A*
    end if
    compute movement cost of v to n using the input map weight value and manhattan distance
    if movement cost is better than set of costs so far **then**
      update open list with n
      update set of costs so far
      update result path p
    end if
  end for
end while

A. The A* pseudo-code

The A* pseudo-code [26] is presented in Algorithm 1. Like all pathfinding algorithms, it begins from a starting node s and loops through an open list of nodes that keeps track of which nodes have been visited so far. At each loop iteration, the node v with lowest cost is removed from the open list (if it exists). The cost is based on the input map weight value on node v and on Manhattan distance (*a.k.a* snake distance), which is used as a heuristic to guide the search towards the goal (target cell). The Manhattan distance is the sum of the absolute differences of two Cartesian coordinates (s,t) and thus can be used to estimate how far a pair of cells are from each other, as can be seen in Equation 1:

$$distance(s,t) = abs(s.x - t.x) + abs(s.y - t.y) \quad (1)$$

Moreover, the open list is actually a priority queue, so that the visited nodes are ordered according to the lowest costs (priorities) which are known so far, *i.e.*, the nodes that are closer to the target cell based on the Manhattan distance and the input map corresponding cell weight. Thus, the A* algorithm expands faster towards the target, but does not always find the shortest path, which is usually not a problem for games.

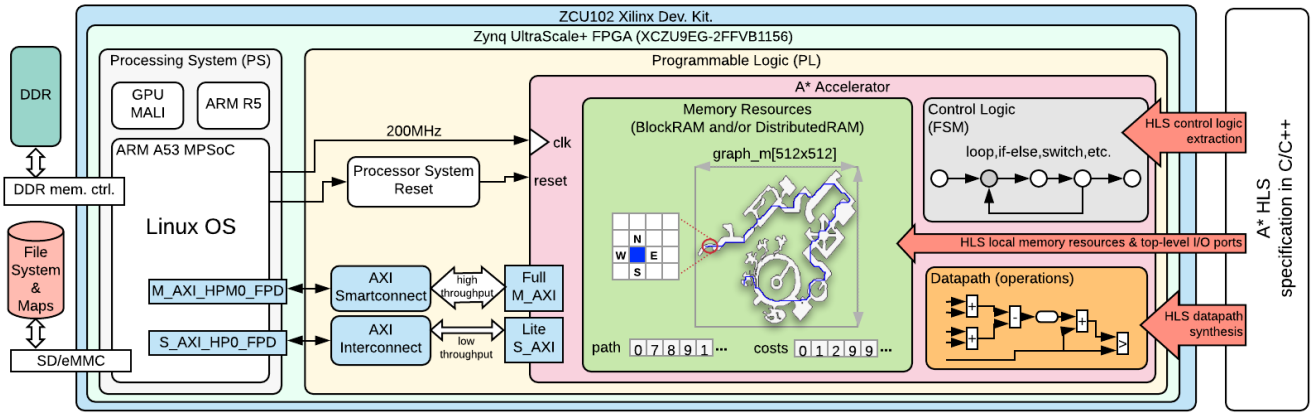


Figure 2. The A* Co-Processor Architecture running at 200MHz and its PS-PL communication interface. The algorithm's HLS specification in C/C++ is analyzed to produce the A* Accelerator (PL), which in turn connects to the Full-Power Domain High-Performance ports of the Processing System (PS), using Smartconnect and Interconnect AXI blocks.

B. The A Co-Processor and Interface*

The A* co-processor (accelerator) is designed using Xilinx HLS compiler, which transforms a C/C++ specification into a RTL implementation suitable for running into Xilinx FPGAs, as shown in Fig. 2. In general, the HLS compiler synthesizes C/C++ functions into blocks in the RTL hierarchy, with the top-level function arguments translated into RTL I/O ports, arrays translated into BlockRAMs (or Distributed RAMs) and the control logic (if-else, switch, for-loop, etc.) turned into a Finite State Machine (FSM), with the loops remaining rolled by default. Whenever possible, the datapath is created with as many parallel operations as possible, as long as there are FPGA resources available. The accelerator communicates to the Processing System Full-Power Domain (FPD) through master (M_AXI_HPM0_FPD) and slave (S_AXI_HP0_FPD) High-Performance ports, using interconnect AXI IP blocks Smartconnect and Interconnect. In this way, game maps stored in the SD card file system can be copied into DDR and mapped onto the accelerator ports for pathfinding processing.

The AXI4 protocol interfaces supported by the HLS compiler include the AXI4-Stream (axis), AXI4-Lite (s_axilite) and AXI4-Master (m_axi). The AXI4-Stream protocol is the fastest, because it can transfer sequential streams of data, with no limitation on the burst length. It is focused on a data-flow paradigm, where the concept of an address is not present. Therefore, it requires a Direct Memory Access (DMA) core on the PL-side connected to a PS high-performance port, translating memory mapped data to stream and vice-versa. The DMA is controlled by the PS via memory-mapped AXI4-Lite interface, connected to a PS general purpose port. This stream protocol is not used in this work due to the need to control DMA, which would also make the PS programming more difficult for non-experienced embedded systems programmers. Also, a DMA controller would consume resources on the FPGA that could be further used on the co-processor's datapath, memory and/or logic control. The AXI4-Lite, on the other hand, is the slowest and, as such, should be applied only for simple, low-throughput memory-mapped communication. Thus, this protocol is used in this work to signal the start of the co-processor and to gather status information, indicating whether the core is idle or the computation has finished. Also, it is used to set the base address, the start and target cells of the path that the core needs to search for.

Lastly, AXI4-Master (also known as AXI4-Full), provides high-performance memory-mapped PS-PL data transfers. This protocol implements burst mode data transfers, *i.e.*, it can burst up to 256 words of data based on a single memory-mapped address, connected to a PS high-performance port. If more data needs to be transferred, the protocol must be granted bus access again in order to burst more data. This protocol is used in this work to transfer the map adjacency array that represents the input graph map, as well as to transfer back to the host (ARM) the resulting path. Using this protocol, the programmer just needs to specify the address of the input grid array to the core, and the address where the resulting path should be stored.

C. The A HLS implementation*

The A* HLS hardware accelerator implementation operates given the input graph (map) base address $*g$, the output path base address $*p$, the start s and target t cell coordinates, and the compiler directives (*pragmas*), as shown in Listing 1. The HLS compiler implements an AXI4-Full interface port which is used to transfer the adjacency array map data and the resulting path back to their respective base addresses in external memory (*e.g.*, DDR). Also, an AXI4-Lite interface port is implemented to transfer the addresses themselves and to transfer the other parameters of the function, such as the start and target cells. The *memcpy* function call indicates

Listing 1. A* HLS implementation.

```
1 #define N_NODES (512*512) //2D grid size 512x512
2 static uchar graph_m[N_NODES]; //adjacency array
3 static int path[N_NODES]; //resulting path array
4 static int costs[N_NODES]; //path costs array
5 void aStar(volatile uchar *g, volatile int *p, int s, int t) {
6 #pragma HLS INTERFACE m_axi depth=N_NODES port=g offset=slave
7 #pragma HLS INTERFACE m_axi depth=N_NODES port=p offset=slave
8 #pragma HLS INTERFACE s_axilite port=g bundle=AXI_Lite
9 #pragma HLS INTERFACE s_axilite port=p bundle=AXI_Lite
10 #pragma HLS INTERFACE s_axilite port=s bundle=AXI_Lite
11 #pragma HLS INTERFACE s_axilite port=t bundle=AXI_Lite
12 #pragma HLS INTERFACE s_axilite port=return bundle=AXI_Lite
13 #pragma HLS ARRAY_PARTITION variable=graph_m cyclic factor=4 dim=1
14 #pragma HLS ARRAY_PARTITION variable=path cyclic factor=4 dim=1
15 #pragma HLS ARRAY_PARTITION variable=costs cyclic factor=4 dim=1
16     //AXI4 burst mode copy
17     memcpy(graph_m,(const int*)g,N_NODES*sizeof(uchar));
18     int n1,n2,n3,n4;
19     int new_cost1,new_cost2,new_cost3,new_cost4;
20     int prior1,prior2,prior3,prior4;
21     int neig[4] = {-1,-1,-1,-1}; //neighbor array
22     list_t perim; //perimeter array-list
23     init_list(&perim); //init ''perim'' list size to 0
24     init(path,costs); //init ''path'' and ''costs'' static arrays to -1
25     put_item_prior(&perim, s, 0);
26     costs[s] = 0;
27     while (!is_empty(&perim)) { //A* main-loop
28         int v = remove_first(&perim);
29         if (v == t) //early exit
30             break;
31         neighborhood(neig, v); //get all 4 neighbor cells of v
32         n1 = neig[0]; //north neighbor
33         n2 = neig[1]; //east neighbor
34         n3 = neig[2]; //south neighbor
35         n4 = neig[3]; //west neighbor
36         if (n1 >= 0) { //north neighbor analysis
37             if (graph_m[n1] > 0 && graph_m[n1] < 255){
38                 new_cost1 = costs[v] + graph_m[n1];
39                 prior1 = new_cost1 + heuristic(n1, end_node);
40                 if (costs[n1] < 0 || new_cost1 < costs[n1]) {
41                     append_item_prior(&perim, n1, prior1);
42                     costs[n1] = new_cost1;
43                     path[n1] = v;
44         } } }
45         if (n2 >= 0) { //east neighbor analysis
46             if (graph_m[n2] > 0 && graph_m[n2] < 255){
47                 new_cost2 = costs[v] + graph_m[n2];
48                 prior2 = new_cost2 + heuristic(n2, end_node);
49                 if (costs[n2] < 0 || new_cost2 < costs[n2]) {
50                     append_item_prior(&perim, n2, prior2);
51                     costs[n2] = new_cost2;
52                     path[n2] = v;
53         } } }
54         if (n3 >= 0) { //south neighbor analysis
55             //removed for the sake of simplicity
56         } } }
57         if (n4 >= 0) { //west neighbor analysis
58             //removed for the sake of simplicity
59     } } } }
60     //AXI4 burst mode copy
61     memcpy((int*)p,path,N_NODES*sizeof(int));
62 }
```

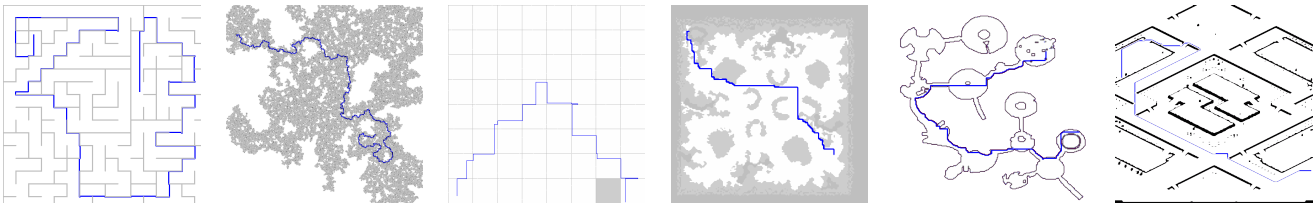
that the graph adjacency array (*graph_m*) should be transferred in burst mode, whenever possible. The HLS compiler automatically implements the protocol handshaking signals, which greatly simplifies the design process of the PS-PL interface. Beyond that, it also produces C-Drivers that can be compiled and used by the ARM programmer to control the co-processor. These drivers are basically C function calls to control each pathfinding core (*e.g.*, set start node, target node, adjacency array base address, start execution, etc.).

As described in Section III-A, the A* algorithm [26] uses Manhattan distance as heuristic to guide the search towards the goal (target cell). Such distance calculation is performed by a separate function, which is *inlined* into the top-level function. Furthermore, the open list is implemented as a priority queue (*perim*) to keep track of the perimeter cells which are closer to the target cell. The queue is also specified in HLS as a *C-struct*, containing the queue size and a pair of arrays: one that holds the item value and the other that holds the item priority. Each array can store up to 262144 items, which is equivalent to the input map maximum size, *i.e.*, 512×512. Each array is mapped onto a BlockRAM of the target FPGA. Inserting an item with a given priority is $O(n)$ in the worst case, because the item needs to be inserted in the position according to its priority. Removing the lowest priority item, *i.e.*, lowest cost node, is O(1).

Furthermore, notice that the processing code for all four neighbors (north, east, south and west) of cell v under evaluation are specified in Listing 1, from lines 36 to 59, with the processing code for the south and west neighbors omitted for the sake of simplicity. The code for each neighbor cell is written in a different *if* clause to ensure that the HLS compiler can produce a datapath and its given control logic capable of processing each neighbor separately, *i.e.*, in parallel. The *ARRAY_PARTITION* pragma directives enables the programmer to split the given arrays onto different BlockRAMs automatically, in order to allow parallel accesses to different positions of the original array. Otherwise, every array access would have to be serialized, which would hurt the overall A* co-processor performance. For instance *graph_m*, *path* and *costs* static arrays in Listing 1 are spread onto four BlockRAMs, so that all the four neighbors of a cell can potentially be analyzed in parallel. On the other hand, access to the perimeter array-list (*perim*) is serial, because in fact it implements priority queue that hinders parallel insertions/deletions. Still, several expressions and memory accesses in the code can be processed in parallel, as will be shown in Section IV-B. The whole A* HLS source code has been made available in github [27] so that this work can be reproduced by different research groups. It includes the HLS code for array-backed lists and graph management. Nevertheless, the HLS code is partially shown in Listing 1 to better explain the idea and the concepts behind this work.

IV. Experimental Analysis

The design process of the A* co-processor using HLS encompasses three main development stages. First, the co-processor must be specified in C/C++ using the HLS compiler subset of ANSI-C allowed operations and transformed into synthesizable VHDL (or Verilog) RTL hardware description. The second stage is the architecture specification, which connects the co-processor to the processing system using different types of interconnects. The last stage is the



(a) *Maze* 32-5. (b) *Random* 40-4. (c) *Room* 64_5. (d) *WC3* golem. (e) *BG* AR0411. (f) *SC* CatwalkAlley.

Figure 3. Examples of artificial and commercial maps, and path examples.

processing system development, which builds on top the designed architecture and uses Xilinx customized petalinux OS. All the experiments were executed in the Zynq-FPGA architecture described in Section III-B.

A. Benchmark

In this work, the 2D grid world benchmark [6] is used to evaluate the A* pathfinding co-processor. It provides several maps that are available to all researchers to use, allowing the comparison of results. This Benchmark set contains commercial games maps, city/street maps and also artificial benchmarks. As the focus of this work is a gaming co-processor, game maps, random maps and artificial mazes were used. As previously explained, the 2D grid maps must be up to of 512×512 wide to fit in the co-processor memory. This limitation has virtually no impact on this benchmark since almost all maps are less than or equal to 512×512. It is important to notice that the original benchmark results are based on Dijkstra's algorithm and allows movements on all 8 directions of each cell, *i.e.*, including diagonals, while our A* implementation allows movements on 4 directions only (north, east, south, west). This is because our A* co-processor tries to analyze all for directions in parallel and could not be properly synthesized for all eight directions due to timing constraints. Fig. 3 presents a set of examples of the maps used in the experiments, organized into Artifically created maps and commercial game maps. Each map cell with unitary cost (1) represents a passable terrain, while others represent non-passable terrain, *i.e.*, obstacles. Maze maps reproduce labyrinths with varying corridor widths (1,2,4,8,16 and 32), as can be seen in Fig. 3a. Random maps are generated randomly with varying probabilities (0.10,0.15,0.20,0.25,0.30 and 0.40) of producing obstacles (Fig. 3b). In turn, Room maps reproduce square rooms with varying sizes (8×8, 16×16, 32×32 and 64×64). An example of a 64×64 room map can be seen in Fig. 3c. Fig.s 3d, 3e and 3f represent scenarios adapted from commercially available games WarCraft3, Baldur's Gate and StarCraft, respectively.

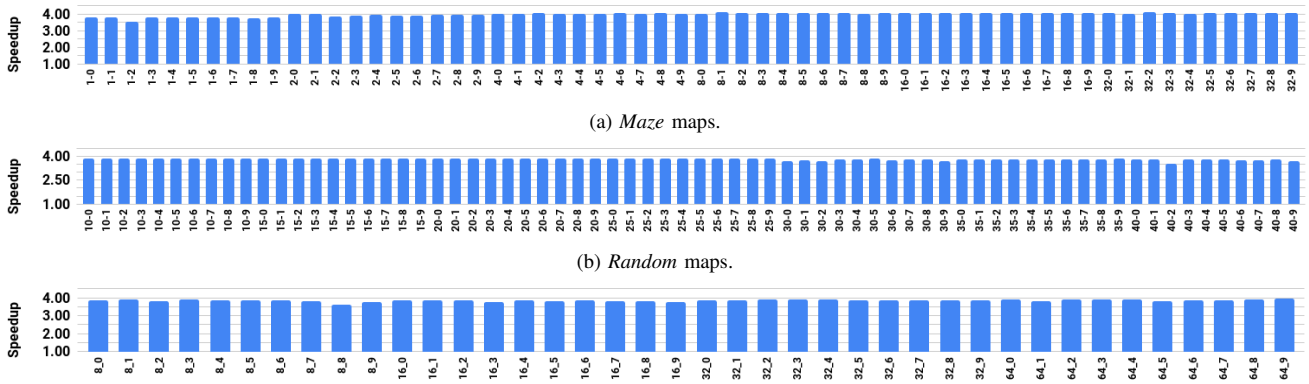
B. Performance analysis

In the first experiment set we evaluate the acceleration of the A* co-processor in comparison to executing the A* algorithm on one ARM core of the MPSoC. Each map (297 in total) was executed 30 times for both A* co-processor (hardware) and A* algorithm on the ARM (software), which resulted in a total of 297×30 executions each. This is to evaluate the representativeness of the results obtained. Hence, the coefficient of variation was calculated for each map. This statistical measure was used because there is a great variation between the execution times of different maps. Considering the A* co-processor, the results obtained are extremely reliable where the average coefficient of variation (CV) for all maps was only 0.0006% (with the smallest CV=0.0002 and the largest CV=0.0114), which shows that execution times are very close to the average. Similarly, the results obtained for the A* algorithm on the ARM are very reliable, where the average coefficient of variation for all maps was only 0.0450% (with the smallest CV=0.0079 and the largest CV=0.4124), despite the great difference between them. The reason for the difference is, basically, that, while the execution on hardware is totally dedicated without overhead, the execution on software requires the creation of threads (or processes) which interferes with execution time. Table I presents the whole execution time (297×30) for each case: using the co-processor and not using it (ARM only). Observe that searching for paths using the A* accelerator takes about 75% less time than using one ARM core, *i.e.*, the co-processor is almost $4\times$ faster.

Table I. Total execution time of the A* accelerator and the ARM when running all the 30×297 maps.

	Co-Processor	ARM
Total execution time (in seconds)	543.80	2096.83

The speedup results are organized into artificial and commercial maps, as can be seen in Fig. 4 and Fig. 5. It can be observed that in almost every map benchmark the A* co-processor is about four times faster when compared to one ARM-core alone, with only a few lower speedup results presented in each benchmark map. The best speed up achieved was 4.10 for Maze-8-1 map, while worst was 2.58 for Baldur's Gate AR0413 map, both depicted in Fig. 6. This good performance was achieved through a efficient exploitation of parallelism in all four directions, while at the same time producing a long path, as depicted in Fig. 6a.

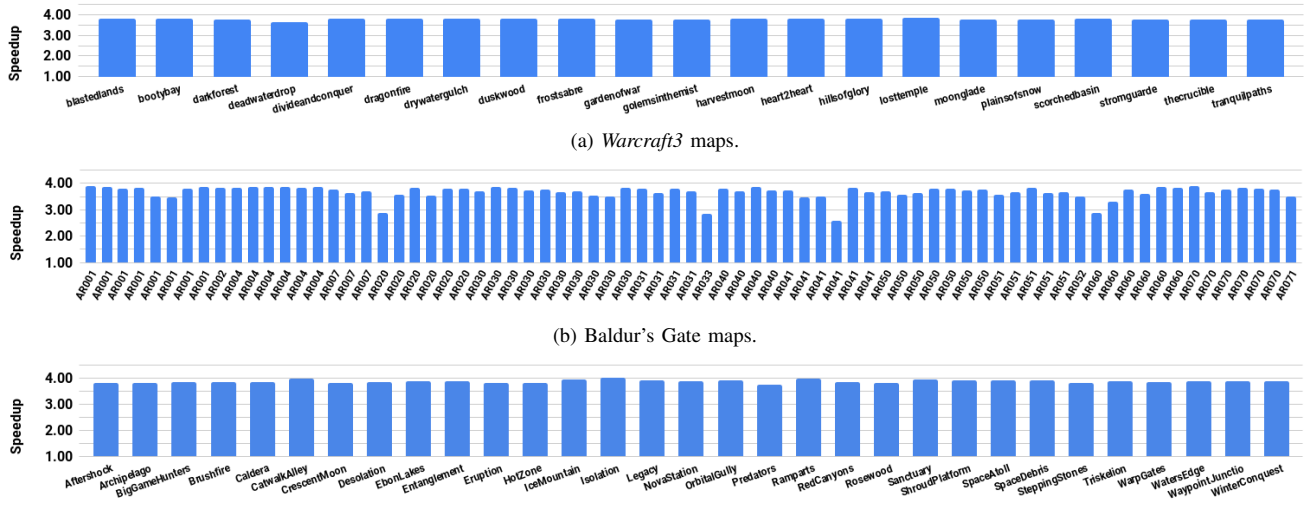


(a) *Maze* maps.

(b) *Random* maps.

(c) *Room* maps.

Figure 4. Speedup results for artificially generated 512×512 grid maps, corresponding to (i) maze maps with variable cooridor width, (ii) random maps and room maps (iii) with random openings between them.



(a) *Warcraft3* maps.

(b) Baldur's Gate maps.

(c) StarCraft maps.

Figure 5. Speedup results for commercial game 512×512 grid maps, corresponding to (i) Baldur's Gate, (ii) WarCraft 3 and (iii) StarCraft commercial games.

On the other hand, the small drop in performance for some maps can be observed when running the accelerator on maps that feature narrow corridors and that produce very short paths, as the one depicted in Fig. 6b. Narrow corridor maps harms parallelism exploitation, as not all four directions can often be explored in parallel. Moreover, using the co-processor to determine short paths might not be as efficient as using it to find long paths, especially due to the PS-PL communication overhead. Still, in all maps, the co-processor always presented better speedup results in comparison to using one ARM core.

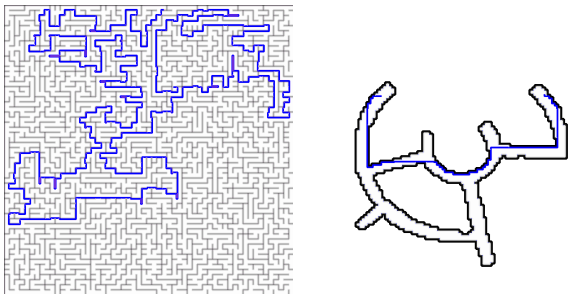
C. Circuit-area analysis

When considering the different resources of the programmable logic (FPGA), the BRAM (*a.k.a*, BlockRAM) is the most used resource, as presented in Table II and in Fig. 7.

Table II. FPGA resources utilization.

Resource	Utilization	Available	Utilization %
LUT	7665	274080	2.80%
LUTRAM	410	144000	0.28%
FF	6868	548160	1.25%
BRAM	617	912	67.65%
BUFG	2	404	0.50%

This is due to the fact the every array in the HLS specification is translated to FPGA-specific BlockRAM slices or implemented as Distributed RAMs using the Lookup Tables that are distributed across the FPGA. The HLS compiler and the Vivado synthesis tool decide how the



(a) *Maze* 8-1, with path length of 5039 cells.

(b) *BG* AR0413, with path length of 687 cells.

Figure 6. Examples of maps with best (a) and worst (b) speedups.

arrays should be implemented based on performance, circuit-area and energy consumption trade-offs, which is a well-known difficult multi-objective optimization problem. Most electronic design automation tools, such as the ones used in this work, must rely on heuristics to overcome the VLSI design complexity.

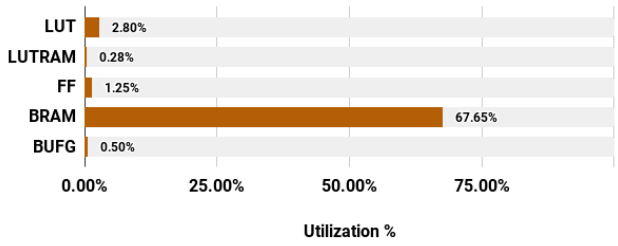


Figure 7. FPGA occupancy results, with the Y-axis representing the FPGA resources.

Followed by the BRAM, the Look-Up Table (LUT) was the second most used resource, while the Flip-Flops was the third most used resource. The LUT can be used to implement any logic function required by the design, which explains its high FPGA occupancy. The Flip-Flops are basic components to small memory elements, such as registers. Thus, most existing variables specified in HLS must have been translated to FFs, while everything else was possibly translated to a logic function implemented in LUT, such as arithmetic operations, multiplexers, decoders, etc.

Finally, observe that more than one A* co-processor would not fit in the Zynq FPGA used in this work (XCZU9EG-2FFVB1156) mainly because of the lack of BlockRAM resources left in the chip. The input map is represented as an array of 262144 8-bit cells, which corresponds to a grid of 512 rows and 512 columns, with each cell value ranging from 0 to 255. The priority queue alone specifies two arrays: one that stores the item priority and another that stores the item value, both 18-bit integers, which suffices to store the values of the input map cell identifier and each list item priority value. Also, the length of both arrays needs to be equivalent (in the worst case) to the number of cells in the input map, *i.e.*, 262144 cells each. Hence, the A* co-processor requires at least 2×262144 18-bit array positions to store the priority queue, 262144 8-bit array positions to store the input map, 262144 32-bit array positions to store the produced path and 262144 32-bit array positions for the costs associated to the path, with negative values included to indicate unreachable cells. While Vivado HLS supports a wide range of the C language, some constructs are not synthesizable, such as system calls for dynamic memory allocation. Thus, every co-processor design (including A*) must be fully self-contained, specifying all required resources before synthesis.

D. Dynamic Energy consumption analysis

The system's (ARM + A*) dynamic energy consumption is estimated based on Vivado vectorless power engine [28], which feeds the design inputs with probabilistic signals over time to gather switching activity information about the circuit. The result is the average power consumption of the logic elements within the design. While the vectorless estimation is not as accurate as a post-route simulation, it still provides reasonable power estimation results. Table III presents the average dynamic power consumption of the A* co-processor (Clocks, Signals, Logic and BRAM) and of the ARM processing system (PS). It can be observed that, together, the system's average dynamic power consumption is 4.157 Watts. Hence, the ARM represents about 65% of the dynamic power, while the A* co-processor represents the rest, thus being less power-hungry than the PS, on average.

Table III. Average power estimation (in Watts) with default toggle rate set to 12.5% and static probability set to 0.5, in vectorless mode.

	Clocks	Signals	Logic	BRAM	PS
Dynamic Power (W)	0.062	0.433	0.081	0.884	2.697

The main reason for such efficiency is due to the dedicated RTL architecture in FPGA, which performs as many parallel operations as possible, in shorter time. Using the execution times given in Table I and the average dynamic power in Table III it is possible to estimate the energy consumption when using the A* co-processor to find paths in comparison to using one ARM core, as shown in Fig. 8. Observe that the A* consumes about 85% less energy than the PS when running the 297×30 maps.

V. Conclusion & Ideas for Future Work

This paper presented an efficient A* pathfinding co-processor suitable for FPGA-based mobile gaming devices using the Advanced Microcontroller Bus Architecture (AMBA4) specification. The co-processor was designed, implemented and evaluated in a Zynq Ultrascale+ Field-Programmable Gate Array (FPGA) from Xilinx, running

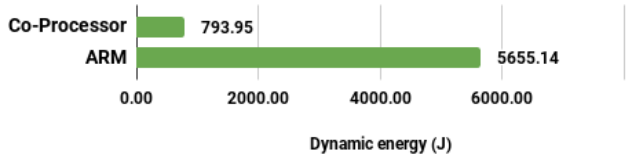


Figure 8. Dynamic energy analysis (in Joules) for the A* co-processor and the ARM (PS) core.

at 200MHz (about $1/6$ of the MPSoC speed). Yet, the co-processor is about four times faster than one ARM microprocessor, enabling the computation of paths in $1/4$ of the time and requiring about 35% of the system's total dynamic power. Moreover, this paper analyzed the feasibility of extending the hardware of mobile devices to execute pathfinding algorithms and to evaluate the efficiency of the RTL hardware produced using Xilinx HLS compiler.

The results clearly show the benefits of using HLS tools to build a pathfinding co-processor, both due to the low energy consumption (bellow 1.46 Watts) but mainly due to its high performance, being up to $4\times$ faster than the ARM microprocessor alone, for an input map of 512×512 cells. Although the co-processor could have been specified directly in hardware description languages, such as VHDL, the development time would possibly be much longer, requiring many testing and verification steps, as usually in any integrated circuit design process. Therefore, specially for mobile devices based on ARM microprocessor, the adoption of co-processors can allow the development of more complex games without the fear of harming its performance and stalling it. Furthermore, the specification of a co-processor using HLS improves the portability of the hardware accelerator and reduces its time-to-market, enabling its implementation in different, more capable FPGA devices, which could possibly fit more co-processors working in parallel.

In the future, the A* co-processor will include an AXI4-Stream interface for faster PS-PL communication, enabling even higher-throughput transfers of larger input graphs and paths, without the burst length limit of AXI4-Full interfaces. Also, arbitrary precision data types will further be used in the co-processor specification in order to avoid the overhead of specifying unnecessary bits, such as when an integer variable is used to store boolean (true or false) values. Moreover, the co-processor is planned to be implemented directly in VHDL to better enable us to evaluate the quality of the RTL architecture produced by the HLS compiler, including its performance, circuit-area and energy consumption when compared to a VHDL optimized design. Finally, we plan to compare our FPGA A* implementation to a GPU OpenCL implementation. As usually, we expect the GPU to require much more energy then the FPGA, which would hurt the gaming device battery life.

Acknowledgment

The authors would like to thank FAPERJ, CNPq and CAPES for the financial support to this work. It is also important to thank Xilinx for the donation of the licenses that allowed the development of this work.

References

[1] "The gmgc mobile games whitebook," http://resources.newzoo.com/gmgc-mobile-games-whitebook, p. 20, January 2017, accessed: 08-08-2017.

[2] "Game developers conference: State of the game industry, 2017." http://reg.techweb.com/GDCSF17-StateOfGame?kcode=pr, p. 14, March 2017, accessed: 08-08-2017.

[3] T. Frikha, N. B. Amor, K. Lahbib, J. P. Diguet, and M. Abid, "Hardware adaptation for multimedia application case study: Augmented reality," in *2014 6th International Conference of Soft Computing and Pattern Recognition (SoCPaR)*, Aug 2014, pp. 411–417.

[4] "Ultrafast high-level productivity design methodology guide," https://www.xilinx.com/support/documentation/sw_manuals/ug1197-vivado-high-level-productivity.pdf, Xilinx, accessed: 08-08-2017.

[5] "Zynq ultrascale+ device: Technical reference manual," https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf, Xilinx, accessed: 18-07-2018.

[6] N. Sturtevant, "Benchmarks for grid-based pathfinding," *Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144 – 148, 2012. [Online]. Available: http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf

[7] "Petalinux tools documentation: Reference guide," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug1144-petalinux-tools-reference-guide.pdf, Xilinx, accessed: 18-07-2018.

[8] "Consolite, a tiny game console on an fpga," https://fotino.me/consolite-fpga/, accessed: 18-07-2018.

[9] "Mega65: open 8-bit computer," http://mega65.org/, accessed: 18-07-2018.

[10] "Fpga replay," http://www.fpgaarcade.com/, accessed: 18-07-2018.

[11] "The zx spectrum reborn: a new machine, fully compatible with the original computer, and packed with improvements and expansions." https://www.kickstarter.com/projects/1835143999/zx-spectrum-next/description, accessed: 18-07-2018.

[12] I. Millington and J. Funge, *Artificial Intelligence for Games*, 2nd ed. CRC Press, 2009.

[13] M. Daga, A. M. Aji, and W. c. Feng, "On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing," in *2011 Symposium on Application Accelerators in High-Performance Computing*, July 2011, pp. 141–149.

[14] S. J. Vaughn-Nichols, "Vendors draw up a new graphics-hardware approach," *Computer*, vol. 42, no. 5, pp. 11–13, May 2009.

[15] T. Y. Yeh, P. Faloutsos, S. J. Patel, and G. Reinman, "Parallax: An architecture for real-time physics," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 232–243, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1273440.1250691

[16] M. Bose and V. Rajagopala, "Physics engine on reconfigurable processor – low power optimized solution empowering next-generation graphics on embedded platforms," in *2012 17th International Conference on Computer Games (CGAMES)*, July 2012, pp. 138–142.

[17] H. Yang, "Floating-point reconfiguration array processor for 3d graphics physics engine," in *2008 Asia and South Pacific Design Automation Conference*, March 2008, pp. 283–283.

[18] C. Liu, X. Ji, Y. Cao, Q. Xu, and L. Chen, "Phusis cloth: A physics engine for real-time character cloth animation," in *Proceedings of 2012 2nd International Conference on Computer Science and Network Technology*, Dec 2012, pp. 1578–1582.

[19] T. Hamano, M. Onosato, and F. Tanaka, "Performance comparison of physics engines to accelerate house-collapsing simulations," in *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, Oct 2016, pp. 358–363.

[20] H. Itoh, "Development of lego mindstorms model construction system on omegaspace platform with physx functions," in *2016 11th France-Japan 9th Europe-Asia Congress on Mechatronics (MECATRONICS) /17th International Conference on Research and Education in Mechatronics (REM)*, June 2016, pp. 038–043.

[21] J. Fabry and S. Sinclair, "Interactive visualizations for testing physics engines in robotics," in *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, Oct 2016, pp. 106–110.

[22] P. V. F. da Silva and S. M. Villela, "Applying pathfinding techniques on the development of an android game," in *Proceedings of SBGames 2016*. SBC, 2016, pp. 73–80.

[23] 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, July 2011.

[24] M. Y. I. Idris, S. A. Bakar, E. M. Tamil, Z. Razak, and N. M. Noor, "High-speed shortest path co-processor design," in *2009 Third Asia International Conference on Modelling Simulation*, May 2009, pp. 626–631.

[25] "Axi reference guide," https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf, accessed: 08-08-2017.

[26] P. E. Hart, N. J. 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, July 1968.

[27] "A* pathfinding specification using xilinx vivado high-level synthesis," https://github.com/Alexandre-Nery/pathfinding, accessed: 12-09-2018.

[28] "Vivado design suite user guide: Power analysis and optimization," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug907-vivado-power-analysis-optimization.pdf, Xilinx, accessed: 04-08-2018.