An efficient pathfinding system in FPGA for edge/ fog computing

Alexandre Solon Nery*

Departamento de Engenharia Elétrica,
Universidade de Brasília,
Brasília, DF, Brazil
Email: anery@unb.br
*Corresponding author

Alexandre da Costa Sena

Departamento de Informática e Ciência da Computação,
Universidade do Estado do Rio de Janeiro,
Rio de Janeiro, RJ, Brazil
Email: asena@ime.uerj.br

Leandro S. Guedes

Departamento de Informática,
Instituto Federal de Educação,
Ciência e Tecnologia de Mato Grosso do Sul,
Corumbá, MS, Brazil
Email: leandro.guedes@ifms.edu.br

Abstract: Pathfinding algorithms are at the heart of several classes of applications, such as network appliances (routing) and autonomous vehicle navigation. Thus, this work aims at designing and evaluating an efficient pathfinding FPGA accelerator based on Dijkstra's shortest path algorithm to mitigate the increasing network traffic problem at the edge of the network. The system is designed using Xilinx High-Level Synthesis (HLS) compiler and is implemented in the programming logic of a Xilinx Zynq FPGA, embedded with an ARM microprocessor which is not only in charge of controlling the co-processor but also in charge of lightweight TCP/IP network communication. Extensive performance, circuit-area, and energy consumption results show that the co-processor can find the shortest path about 2.5 times faster than the system's ARM microprocessor, on a simulation scenario test case based on touristic locations in the city of Rio de Janeiro, acquired from the OpenStreetMap database.

Keywords: pathfinding; FPGA accelerator; high-level synthesis; fog computing; edge computing.

Reference to this paper should be made as follows: Nery, A.S. and da Costa Sena, A. and Guedes, L.S. (2019) 'An efficient pathfinding system in FPGA for edge/fog computing', *Int. J. Grid and Utility Computing*, Vol. 10, No. 3, pp.212–223.

Biographical notes: Alexandre Solon Nery graduated in Computer Science from Universidade Católica de Brasília (2006), having received his MSc degree (2010) and DSc degree (2014) in Systems Engineering from Universidade Federal do Rio de Janeiro. He is currently acting on the following subjects: reconfigurable computing, distributed and parallel computing.

Alexandre da Costa Sena graduated in Computer Science from Universidade Federal Fluminense (1995), having received his MSc degree (2000) and DSc degree (2008) in Computer Science from Universidade Federal Fluminense. His research interests include parallel computing, high-performance computing, fog computing and dynamic scheduling.

Leandro S. Guedes graduated in Computer Science from Universidade Federal de Pelotas (2013), having received his MSc degree (2016) in Computer Science from Universidade Federal do Rio Grande do Sul. He is currently acting on the following subjects: computer graphics, information visualisation, human-computer interaction and distributed systems.

This paper is a revised and expanded version of a paper entitled 'Efficient Pathfinding Co-Processors for FPGAs' presented at the '2017 International Symposium on Computer Architecture and High-Performance Computing Workshops (SBAC-PADW)', Campinas, Brazil, 17–20 October 2017.

1 Introduction

The rapid development of pervasive computing and Internet of Things (IoT) applications poses several challenges (e.g., efficiency and low latency) on traditional centralised cloud computing systems that are the basis for most web services (Ai et al., 2017). To overcome such challenges, new technology is changing the centralised cloud computing architecture to edge devices on the network border, in a trend called fog or edge computing (Cerina et al., 2017; Li et al., 2018). Hence, while some functions are better suited for cloud computing, others are naturally more advantageous to be carried out by fog nodes (Liu et al., 2018; Ahn et al., 2017). Thus, the focus has shifted to bring specialised computation and communication devices nearer to the user. Many modern embedded systems, such as smartphones and tablets, are already being equipped with in-house customised Multi-Processing Systems-on-Chip (MPSoC), often built around ARM multi-core architectures, like the Cortex-A53, A72 and A73 chips. These application-specific integrated circuit systems include inside a single chip key advanced heterogeneous components specialised in media and communication processing, yielding higher efficiency and reduced energy consumption.

Vendors of Field-Programmable Gate Arrays (FPGAs), such as Xilinx and Altera, have recently embedded ARM microprocessors into the programmable logic of their re-configurable chips, allowing the extension of the microprocessor's basic functions. Combined with High-Level Synthesis (HLS) compiling tools (Vivado, 2017; Clow et al., 2017; Uliana et al., 2013) 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 Intellectual Property (IP), but also to offload code execution to efficient and dedicated parallel hardware accelerators implemented on the FPGA-side of the computing system, further improving its overall efficiency. For instance, the Zynq architecture (Millington and Funge, 2009) is embedded with an ARM Cortex-A9 microprocessor, which is often used to run the least computation intensive part of the code, while the FPGA runs the high-performance demand and energy efficient part.

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. They can be used in a wide range of applications, such as in network appliances and games (Togelius and Yannakakis, 2016). Lately, pathfinding algorithms have an important role in the implementation of autonomous vehicles (Dolgov et al., 2009). In most cases, tasks need to continuously seek for suitable routes in a graph which represents the paths, such as a map. Thus, the pathfinding algorithms should execute quickly enough to not stall the path planning program.

This work extends on our previous workshop paper (Nery et al., 2017) and proposes an efficient pathfinding system in FPGA suitable for edge/fog computing services targeting autonomous vehicles applications. Differently, from our previous work, that had a random 10×10 lattice graph pre-loaded into each evaluated co-processor, we actually implemented an actual fog node that is able to receive a map from the cloud or other fog node and compute a route in its co-processor. The system consists of a Xilinx ZC706 development board equipped with a Zynq XC7Z045 FPGA, different from the older model (XC7Z010) used in the previous work, allowing the analysis of larger input graphs. Moreover, the system's programmable logic only implements Dijkstra's shortest path algorithm (Dijkstra, 1959), because we want to efficiently determine the best route from a given starting point to a given goal. Thus, Breadth-First Search (Lee, 1961), Greedy (Cormen et al., 2009) and A* (Hart et al., 1968) pathfinding algorithms are not implemented. Also, the ARM microprocessor handles the communication part of the system on top of the FreeRTOS operating system, in order to allow the vehicle to download maps from the OpenStreeMap (OpenStreetMap contributors, 2017) database using the lightweight TCP/IP network stack (Dunkels, 2001). The aim is to design and evaluate specialised computing devices placed at the edge of the network to efficiently determine the best route in edge/fog computing architectures based on real map locations.

The rest of this paper is organised as follows: Section 2 describes the state-of-the-art related works. The system's architecture is described in Section 3, which also explains the pathfinding algorithm implemented in the FPGA programmable logic. Extensive experimental results are presented in Section 4. Finally, Section 5 concludes and presents ideas for future work.

2 Related work

Hardware accelerators, such as GPUs and FPGAs, are often used to execute the most timing consuming parts of a specific application or group of applications, leaving the less critical part of the computation to the host micro-processor. They are often designed for applications with high potential for parallelism exploitation, such as physics simulation (Yeh et al., 2007; Bose and Rajagopala, 2012; Yang, 2008), clothing (Liu et al., 2012), 3D-object collision (Hamano et al., 2016), model-based robotics (Itoh, 2016; Fabry and Sinclair, 2016) and others (Daga et al., 2011; Vaughn-Nichols, 2009). More recently, hardware accelerators are also being used in the

edge/fog computing context (Tan et al., 2017; Cerina et al., 2017; Koromilas et al., 2017).

Pathfinding is also a common operation found in artificial intelligence, networking and route planning applications, such as autonomous vehicles (Dolgov et al., 2009; Kok et al., 2013). An evaluation of pathfinding algorithms has been presented in da Silva and Villela (2016), which analyses the Breadth-First, Depth-First, Ordered, Greedy and A* algorithms on Android platforms. Results show that heuristics-based methods, such as A*, are more efficient in terms of execution time for games.

An implementation of a FPGA Bellman-Ford pathfinding algorithm can be seen in Jagadeesh et al. (2011). The architecture distributes the input graph among 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. It runs at 143 MHz 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 impossible to compare to the work presented here.

A FPGA design for large-scale graph processing is presented in Zhou et al. (2016). The design uses large external memory for storing graph data and FPGA for acceleration. The parallel architecture in FPGA is used to saturate the external memory bandwidth and concurrently process multiple input data to increase throughput. Performance and energy consumption are measured in terms of Million Traversed Edges Per Second (MTEPS). Another FPGA design is presented in Umuroglu et al. (2015), targeting the Breadth-First search algorithm acceleration using hybrid FPGA-CPU processing on a range of synthetic small-world graphs. The hybrid approach is able to perform better than software-only, also in terms of MTEPS, on the ZedBoard platform.

The work in Idris et al. (2009) 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.

The work in Menon (2017) presents challenges and open problems to move from vehicular cloud computing to vehicular fog computing. It highlights that an intermediate fog layer between the cloud and the mobile devices can provide low latency, better QoS, improve the efficiency of the network, reduce energy consumption, among other advantages.

On the other hand, the article in Lee et al. (2016) discusses the evolution of intelligent vehicle grid to an autonomous vehicular fog that are capable of making its own decisions about driving customers to their destinations. Since the urban fleet of vehicles is evolving from a collection of sensor platforms to the internet of autonomous vehicles the fog layer will be responsible to compute the huge amount of data produced in a feasible time. The article shows a vehicular fog model in detail and discusses the potential design perspective with highlights on autonomous vehicles for future research.

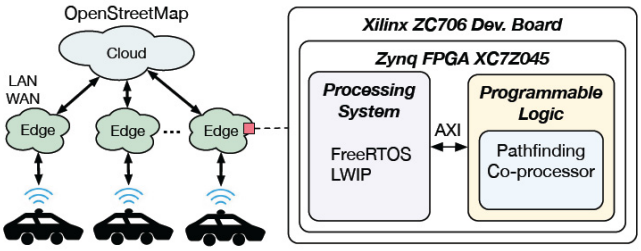
A Reference Architecture for Fog Computing was defined in Consortium (2017) by the OpenFog Consortium that was founded by ARM, Cisco, Dell, Intel, Microsoft and Princeton University. According to this document unfettered cloud computing approaches are not able to support the growth of data provided by transportation, agriculture, visual security, wind farms, among others IoT projects. Therefore, the OpenFog architecture is the baseline to develop an open architecture fog computing environment, creating standards to enable interoperability in IoT, 5G, artificial intelligence, tactile internet, virtual reality and other complex data and network intensive applications.

The work described in this paper is different from all previous works since it proposes an approach to create and evaluate a fog node for pathfinding systems in FPGA for real world map locations stored on the cloud. The computation is in the edge node itself (or in the vehicle), instead of in the cloud, avoiding data communication and improving the performance of IoT vehicle systems. Moreover, it proposes and evaluates a pathfinding co-processor highlighting its benefits such as low energy consumption and high performance.

3 The system's architecture

The pathfinding system's architecture is presented in Figure 1 together with the proposed FPGA accelerator/co-processor for edge/fog computing services.

Figure 1 The edge/fog computing system using a Zynq system-on-chip re-configurable FPGA chip augmented with the pathfinding co-processor



Each edge node may work, for instance, as a fog node in the OpenFog Reference Architecture Consortium (2017), which is a consortium intended to help create and maintain the hardware, software and system elements necessary for fog computing. More specifically, each edge/fog node is an autonomous node that can receive and send data to the cloud or to another fog node. Therefore, each node can receive a map from the cloud and compute a route on the system's co-processor or in the ARM processor.

The Zynq-FPGA architecture is split into Processing System (PS) and Programmable Logic (PL) parts. The first consists of an embedded ARM Cortex-A9 SoC, which can be

programmed in software, while the latter is the re-configurable logic, which can be programmed using hardware description languages, such as VHDL, or high-level synthesis compilers and tools which automatically produce the hardware architecture.

3.1 The pathfinding co-processor

The pathfinding problem is often described using graph theory and its related data structures. Thus, let $G = (V, E)$ be a graph such that V is the set of vertices, also known as nodes, and E the set of edges. Pathfinding can be defined as an algorithm that provides a path, i.e., a route, between a pair of nodes $(s,t) \in G$, where s denotes the start and t the target. In an autonomous car scenario, the nodes can represent specific way-points and landmarks, while the edges represent the possible paths between nodes. In general, the pathfinding algorithm begins the search from a starting node s and, at each loop iteration, expands the perimeter of the search based on the neighbors of the node that is being visited.

Listing 1 presents the C/C++ code snippet for Dijkstra's algorithm implemented in Vivado HLS compiler. Given an input graph, its size and a pair of start and goal coordinates, the algorithm outputs the shortest path, if such path exists. The code first presents static global arrays on lines 1 to 3, which are often mapped onto the FPGA's Block-RAMs. The `graph_m` array stores the adjacency matrix of the input graph coded as a uni-dimensional array. The two other arrays, `path` and `costs`, hold the resulting path and the costs associated with it, respectively. The adjacency array, in particular, is the largest one, containing 250,000 positions that represent an input graph of up to 500 nodes. The limitation on the number of nodes in the graph is due to the limited number of available BlockRAMs in the FPGA. Thus, keeping the data inside the FPGA (using BlockRAMs) is much faster than using the board's external memory resources (e.g., DDR memory) and enables parallel access to many of the available BlockRAMs. Moreover, BlockRAMs are usually scarce in FPGA architectures, limited to a couple dozens of megabits. The ZC706 FPGA board used in this work has a total amount of 19.1 Mb of BlockRAM resources. Thus, working with an input graph of 1000 integer nodes (32-bits each) is unfeasible, as it would require about 30 Mb of BlockRAM memory for a adjacency matrix representation and, thus, would not fit in the 19.1 Mb of available BlockRAMs. Besides, all the auxiliary arrays that are needed during the algorithm's execution would not fit too. Further details on the co-processor's resource utilisation are given later in Section 4. Moreover, the graph adjacency array receives its values from external memories, either connected to the ARM host processor or inside it, with the function parameter `g_addr` providing its starting base memory address, as shown in line 17. Memory functions like `memcpy` are specifically used in Vivado HLS to indicate AXI (Advanced eXtensible Interface) bus burst operation, as will be described in Sub-section 3.2.

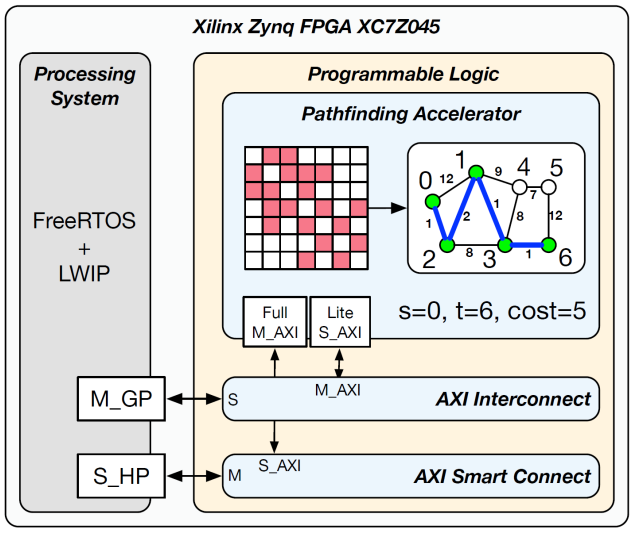
Listing 1: Dijkstra vivado HLS implementation

```
1 static int graph_m[250000]; //adjacency array
2 static int path[500]; //resulting path array
3 static int costs[500]; //path costs array

5 void dijkstra(volatile int *g_addr, volatile int *p_addr,
      int n, int s, int t)
6 {
7     #pragma HLS INTERFACE m_axi depth=250000
          port=g_addr offset=slave
8     #pragma HLS INTERFACE m_axi depth=500
          port=p_addr offset=slave
9     #pragma HLS INTERFACE s_axilite port=g_addr
          bundle=AXI_Lite1
10    #pragma HLS INTERFACE s_axilite port=p_addr
          bundle=AXI_Lite1
11    #pragma HLS INTERFACE s_axilite port=n
          bundle=AXI_Lite1
12    #pragma HLS INTERFACE s_axilite port=s
          bundle=AXI_Lite1
13    #pragma HLS INTERFACE s_axilite port=t
          bundle=AXI_Lite1
14    #pragma HLS INTERFACE s_axilite port=return
          bundle=AXI_Lite1

16    //AXI4 burst mode copy
17    memcpy(graph_m,(const int*)g_addr,n*n*sizeof(int));

19    int i,n,new_cost;

21    //perimeter and neighbors array-lists
22    list_t perim;
23    lsit_t neig;
24    init_list(&perim); //init "perim" list size to 0
25    init_list(&neig); //init "neig" list size to 0

27    init(path,costs); //init "path" and "costs" static
          arrays to -1

29    put_item_prior(&perim, s, 0);
30    costs[s] = 0;

32    //dijkstra main-loop
33    while (!is_empty(&perim)) {
34       int v = get_item_prior(&perim);
35       if (v == t) //early exit
36          break;

38       //get neighbors of v in graph_m
39       neighborhood(&neig, graph_m, v);

41       for (i = 0; i < neig.size; i++) {
42          n = neig.data[i];
43          new_cost = costs[v] + edge_w(graph_m,v,n);
44          if (costs[n] < 0 || new_cost < costs[n]) {
45             costs[n] = new_cost;
46             put_item_prior(&perim, n, new_cost);
47             path[n] = v;
48          }
49       }
50    }
51    //AXI4 burst mode copy
52    memcpy((int*)p_addr,path,n*sizeof(int));
53 }
```

Besides the static arrays, lines 22 and 23 present array-backed lists that are used by the algorithm's main loop. The first list (`perim`) stores the nodes perimeter, i.e., the ones that are still open to evaluation. The second (`neig`) stores the neighbors of the node that is currently under assessment. Each list is often mapped onto FPGA Distributed RAMs. Even though the code for the lists is not presented (for the sake of simplicity), it suffices to say that each can hold up to 500 integer elements, i.e., equivalent to the maximum number of nodes of the input graph. Furthermore, the lists allow the insertion of items with or without priorities, meaning that an item can be inserted at any position in the list depending on its priority, which can be an edge weight or some other cost information. This behavior is what makes each pathfinding algorithm more or less efficient in terms of performance and usage of FPGA resources, especially because inserting an item in the middle of the list requires more steps to first determine the position and later on insert

the item. On the other hand, taking elements from the beginning or from the end of each list has an impact on the search criteria of the pathfinding algorithm, often leading to a faster path search. For instance, the Breadth First search is the simplest pathfinding algorithm and does not consider edge weight or any other information to insert nodes into the lists. Thus, given a start and target nodes, the search expands equally in all directions until the given target node is reached or all nodes have been visited. The Greedy algorithm, on the other hand, simply expands its search based on the perimeter node with lowest edge weight at each loop iteration, possibly resulting in a sub-optimal path, while Dijkstra's tends to prioritise the search towards all perimeter nodes with low-cost edges. It is important to point out that, among these algorithms, Dijkstra is the only one that always produces the optimum solution (smallest path). Finally, A* operates like Dijkstra. The key difference lies in the use of a heuristic to guide the search towards the nodes which are closest to the goal, avoiding the nodes which are farthest from the goal. However, it may not produce the optimum solution, which is the main reason for using Dijkstra's algorithm in this work. Figure 2 presents the co-processor architecture and its AXI compatible buses.

Figure 2 The Zynq system-on-chip re-configurable architecture augmented with the pathfinding accelerator using AXI4-Full and AXI4-Lite interfaces. The depicted shortest path starts at node $s = 0$ and ends at node $t = 6$



A detailed analysis of these four algorithms (Breadth First, Greedy, A* and Dijkstra) was presented in our previous workshop paper (Nery et al., 2017) that designed and evaluated four efficient pathfinding co-processors for each one of these algorithms. It showed that although Breadth First and Greedy co-processors were slightly faster than Dijkstra's, most of the time they did not calculate the optimum path. On the other hand, the performance of A* algorithm was almost the same as Dijkstra's but occupied the largest area, limiting its performance (e.g., the number of co-processors that could be used at the same time).

3.2 The co-processor interface

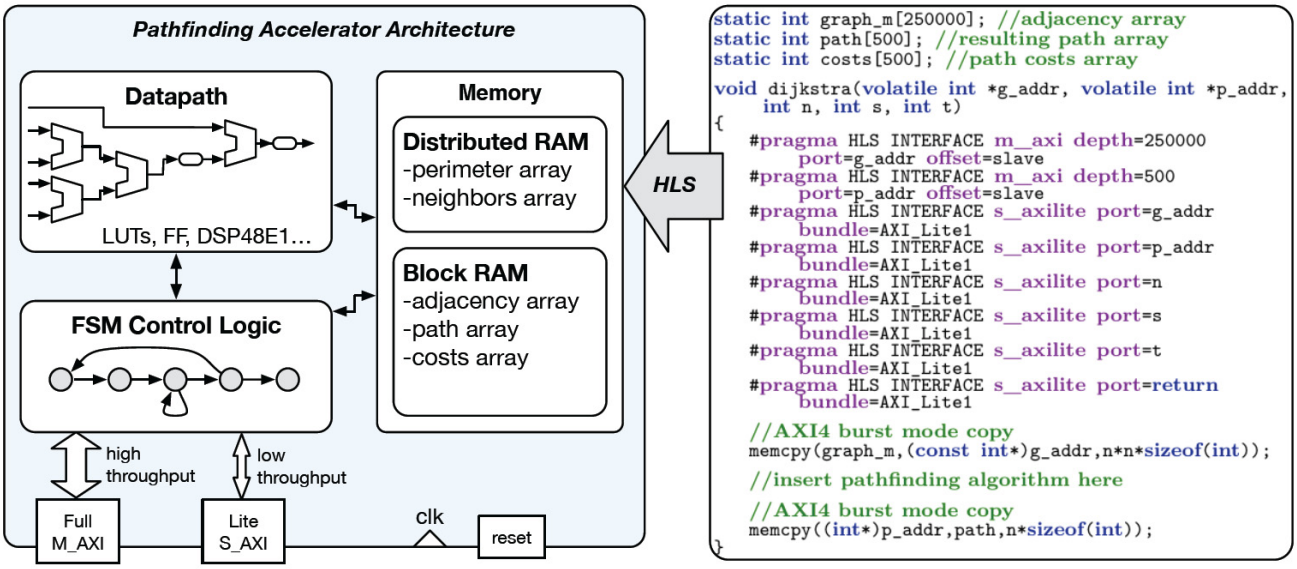
The PS-PL communication interface operates according to the AXI protocol, version 4, which is part of the AMBA-4 specification (AXI Reference Guide, 2017). The PS available interfaces are: 4× General Purpose (GP) AXI master/slave ports, 4× High-Performance (HP) AXI slave ports and one Acceleration Coherency (ACP) slave port. In general, the last two interfaces (HP and ACP) are used for high-performance burst transfers between the PS and PL, while the General Purpose (GP) interfaces are used for control signals and tolerable high-latency pieces of data. The proposed co-processor's interface connects to both GP and HP interfaces of the PS. The co-processor is designed using Xilinx HLS compiler, which transforms a C specification into an RTL implementation suitable for running into Xilinx FPGAs, as shown in Figure 3. In general, the HLS compiler maps C functions onto VHDL entities or Verilog modules. Each function is properly organised in the produced RTL hierarchy, with the top-level function arguments translated into RTL I/O ports, some arrays translated into BlockRAMs (or Distributed RAMs) and loops remaining rolled by default. The AXI4 protocol interfaces supported by the HLS compiler include the AXI4-Stream (s_axis and m_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 over a memory-mapped AXI4-Lite interface, connected to a PS general-purpose port. This protocol is not used in this work due to the need to control the DMA, which would also make the PS programming more difficult for non-experienced embedded systems programmers.

Diversely, the AXI4-Lite is the slowest and 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 nodes of the path that the core needs to search.

The last protocol, 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 need 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 adjacency array that represents the input graph nodes and edges, as well as to transfer back to the host (ARM) the resulting path. Using this protocol, the programmer just needs to specify the base addresses of the adjacency array and the resulting path to the core.

Figure 3 The pathfinding general co-processor architecture and its HLS interface specification



Thus, given the base addresses and the compiler directives (i.e., pragmas), as shown in the code snippet in Figure 3, the HLS compiler implements an AXI4-Full interface port which is used to transfer the adjacency array data into the accelerator and the resulting path back to the pointed memory address, possibly starting at a different base address. Also, an AXI4-Lite interface port is implemented to transfer the base addresses and the other parameters of the function, such as the number of nodes, the start and target nodes. The `memcpy` function call indicates that the graph adjacency array (`g_addr`) and the resulting path (`p_addr`) should be transferred in burst mode, whenever possible. The HLS compiler automatically implements the protocol handshaking signals, which greatly simplifies the design process of each co-processor. 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 the accelerator core (e.g., set start node, target node, adjacency array base address, start execution, etc.).

4 Experimental analysis

The design process of the accelerator 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 synthesisable 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 protocols (AXI4-Lite, AXI4-Full and/or AXI4-Stream). The last stage is the processing system development, which builds on top the designed architecture.

Performance, circuit-area and energy consumption results were analysed in a Xilinx Zynq-7000 FPGA (XC7Z045-FFG900–2 SoC) within Xilinx ZC706 Development Kit. The input graphs are based on popular locations in Rio de Janeiro gathered from the OpenStreeMaps database (OpenStreet Map contributors, 2017) using OSMNX python library (Boeing, 2016). Each graph has been simplified to its corresponding undirected graph representation, because OpenStreetMap not only include intersections, but also they include all the points along a single street segment where the street curves. Although the curves and the direction of the streets are important aspects for autonomous vehicles such as cars and buses, the undirected streets provide important and relevant circuit-area, performance and energy-consumption results, especially because every edge of each node must be analysed, while in the directed version only the outgoing edges would need to be analysed. Thus, the undirected analysis provides worst case results in terms of execution time. Besides, each graph is limited to 500 nodes, based on a radius parameter given a pair of latitude and longitude points. This limitation is due to the FPGA's finite number of resources, such as Block-RAMs and Distributed-RAMs. The vertices representing curves have an impact on the number of required nodes and, thus, BlockRAM resources. Despite that, considering curves vertices is just a matter of input representation and would require no modification of the co-processor design. The number of edges in the graph is not limited, which means that the input graph can be a complete graph, i.e., when every pair of distinct vertices is connected by a unique edge. On the other hand, the more edges the graph contains the wider the array-backed lists need to be in order to accommodate longer paths.

4.1 Performance analysis

In the first experiment set, we evaluate the performance of the proposed accelerator when executing in the FPGA in

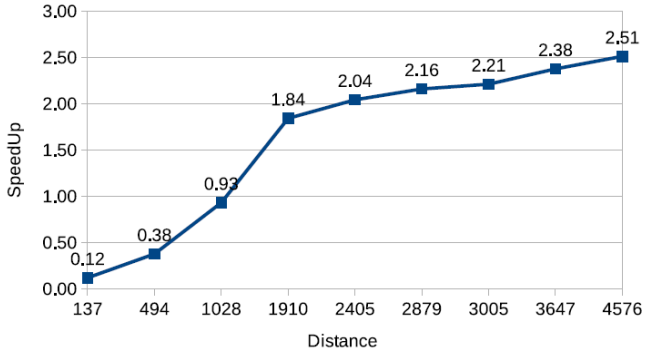
contrast to executing the same algorithm using the ARM microprocessor, while also varying the start and end nodes in order to produce paths with increasingly higher costs. The route cost is the distance (in metres) from one node to another. This is already provided by the street maps data format and is used by Dijkstra's algorithm to determine the cheapest route in terms of distance. Higher cost paths are produced given start and end nodes which are farther away from each other. Such high cost paths are expected to require more processing and energy than lower cost paths. Table 1 presents the elapsed cycle count results regarding both implementations. The number of cycles is assessed by the ARM's global counter, which increments a counting register every two cycles. Thus, the values are adjusted to twice the measured value. Moreover, the input graph corresponds to `Botafogo's` map Figure 8(c), with 481 nodes.

Table 1 Cycle counts measured from the ARM's global timer perspective

Cost	*ARM*	*Co-Proc.*
4576	13,118,026	5,224,018
3647	8,560,954	3,603,540
3005	6,067,660	2,745,106
2879	5,496,766	2,545,894
2405	4,380,572	2,145,530
1910	3,089,364	1,677,786
1028	864,742	929,478
494	277,200	737,352
137	79,696	673,076

The speedups of the co-processor execution against the ARM can be seen in Figure 4. It is possible to observe that the ARM microprocessor executes Dijkstra's shortest path algorithm faster than co-processor does for small distances, i.e., nodes which are located close to each other (around 1km or less). On the other hand, as the distance increases, the co-processor is faster than the micro-processor. This indicates that the more processing needs to be done, the better is the co-processor's speedup in comparison to the ARM, which is expected in most hardware accelerators due to the cost of transferring data in and out the co-processor. Such costs are mitigated by the AXI4-Full burst mode operation, which enables the transfer of bursts of data between the ARM and its co-processor. Otherwise, the communication costs become prohibitive.

Figure 4 Speedups of the co-processor with respect to the same algorithm executed by the ARM alone



Further performance results are presented in Table 2, varying the map region (latitude and longitude) and using random start/goal locations. The radius parameter is used to control the number of nodes to be considered by the pathfinding algorithm, with the corresponding paths being presented in Figure 8. It is possible to observe that the speedup varies as the considered location changes, achieving from 2.41 × to 2.9 × speedup for the `Centro` and `Maracana` input maps, depicted in Figure 8(e) and Figure 8(f), respectively. Furthermore, notice that despite `Copacabana's` highest path cost, its speedup is not among the highest. This is because the path cost does not necessarily mean that the algorithm had to execute more steps, except for the results presented previously in Figure 4, as the start/goal points were indeed selected in a manner to produce results with more nodes along the path as the distance increases. Thus, the cost indicates the path's length in meters, meaning in fact that one path is longer or shorter than another, but not actually more or less costly in terms of clock cycles.

Table 2 Performance results for different locations in Rio de Janeiro, thus varying the map location coordinates (latitude and longitude), as well as the considered region radius

Name	*Lat.*	*Long.*	*Start*	*Goal*	*Nodes*	*Radius*	*Cost*	*ARM*	*Co-Proc.*	*Speedup*
Copacabana	–22.9672	–43.1874	193	134	442	1500	4538	13,097,424	5,214,976	2.51
Lagoa	–22.9719	–43.2119	262	211	451	1400	3288	12,761,352	4,889,296	2.61
Botafogo	–22.9502	–43.1844	211	421	481	1400	4044	16,989,890	6,330,332	2.68
Gávea	–22.98169	–43.23947	11	300	490	2000	4019	16,229,548	6,027,752	2.69
Centro	–22.9033	–43.1862	23	430	432	1000	2146	11,577,490	4,800,368	2.41
Maracana	–22.9120	–43.2301	344	24	482	1200	3807	16,852,114	5,818,882	2.90

4.2 Energy requirement analysis

One of the main advantages of using a co-processor oriented design is its lower power requirements compared to general-purpose processors in hardware, as can be seen in Figure 5. The processing system (named PS7) is the most power-hungry part, being responsible for almost half of the system's power requirements. On the other hand, the programmable logic (FPGA) is responsible for about 32% of the required power. Figure 6 presents the dynamic energy consumption analysis for the Processing System (PS7) and programmable logic (Clocks, Signals, Logic, BRAM and DSP). The energy consumption is roughly estimated from the power report and the clock cycles presented on Table 1, based on 5 ns clock period for the programmable logic and 1.5 ns clock period for the processing system. Besides, the clock cycles of the co-processor had to be adjusted, because they represent the elapsed clock cycles from the ARM's global timer perspective. Thus, the number of clock cycles of the co-processor is about $\frac{10}{3}$ of the ones measured by the ARM. The results show that the co-processor is much more efficient in terms of energy consumption in comparison to using the ARM's processing system of the Zynq FPGA, especially for distant points. Actually, the energy consumption of ARM's processing monotonically increases with the distance, while the energy consumption of the co-processor remained almost constant.

Figure 5 FPGA power report resulting from the system's synthesis with 5 ns timing constraint, default toggle rate of 12.5 and 0.5 static probability

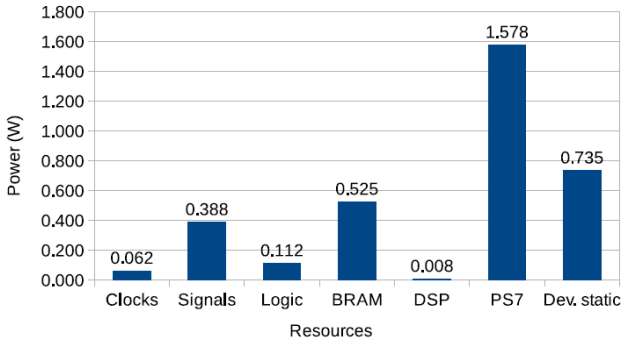
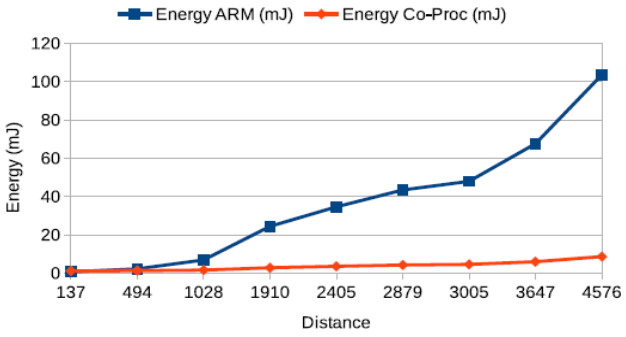


Figure 6 FPGA dynamic energy consumption ($10^{-3}J$)



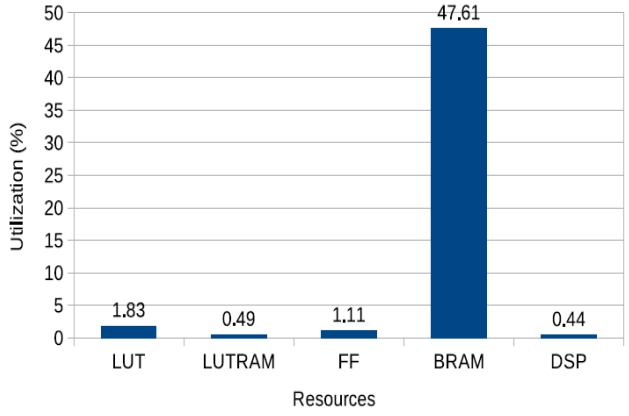
4.3 Circuit-area analysis

When considering the different resources of the programmable logic (FPGA), the BRAM was by far the most used resource, as presented in Table 3 and Figure 7. This is due to the fact that 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 arrays should be implemented based on performance, circuit-area and energy consumption trade-offs, which is a well-known difficult multi-objective optimisation problem. Most electronic design automation tools, such as the ones used in this work, must rely on heuristics to overcome the VLSI design complexity.

Table 3 FPGA resource utilisation

Resource	*Utilisation*	*Available*	*Utilisation (%)*
LUT	4004	218,600	1.83
LUTRAM	342	70400	0.49
FF	4854	437,200	1.11
BRAM	259.5	545	47.61
DSP	4	900	0.44

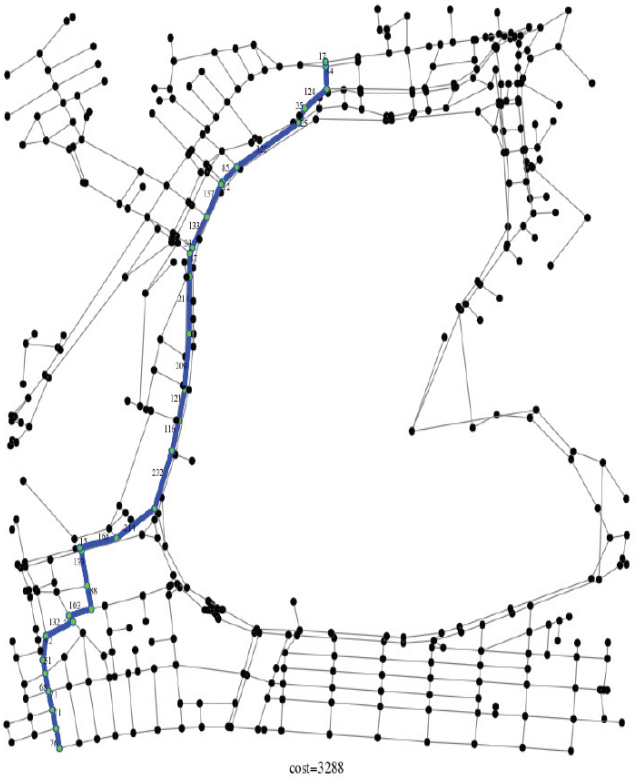
Figure 7 FPGA occupancy results, with the BRAM being the most used resource



Following the BRAM, the Look-Up Table (LUT) was the second most used resource, while the Flip-Flops (FFs) were the third. The LUT can be used to implement any logic function required by the design, which explains its high FPGA occupancy. The FFs are basic components of 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.

Figure 8 Rio de Janeiro popular locations

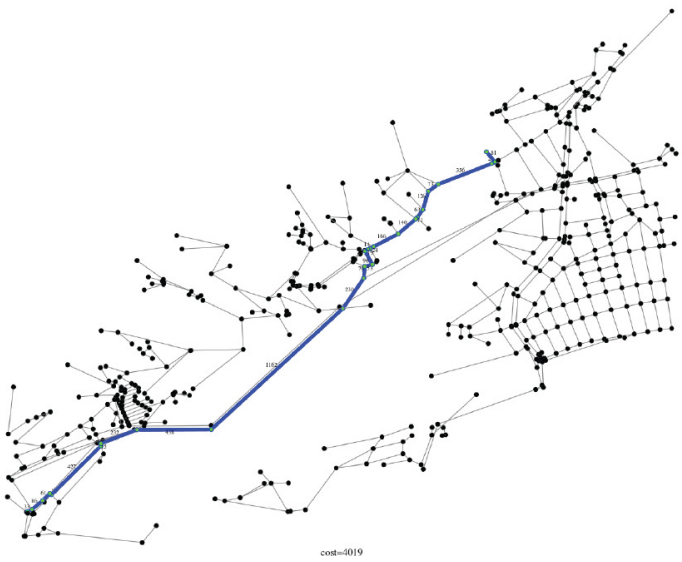
(a) *Copacabana* map, 442 nodes



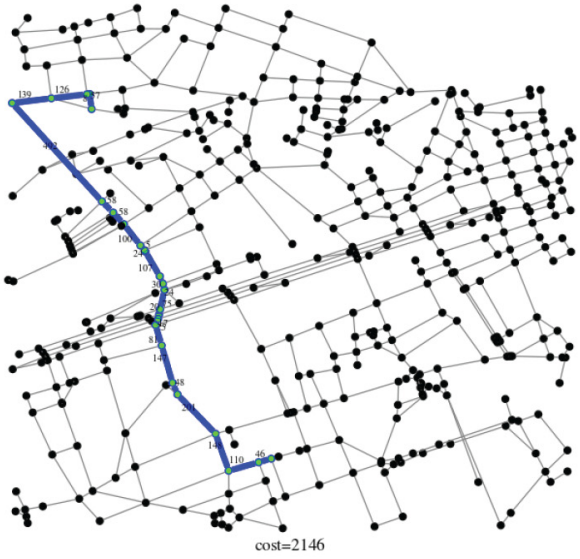
(b) Lagoa map, 451 nodes

(c) *Botafogo* map, 481 nodes

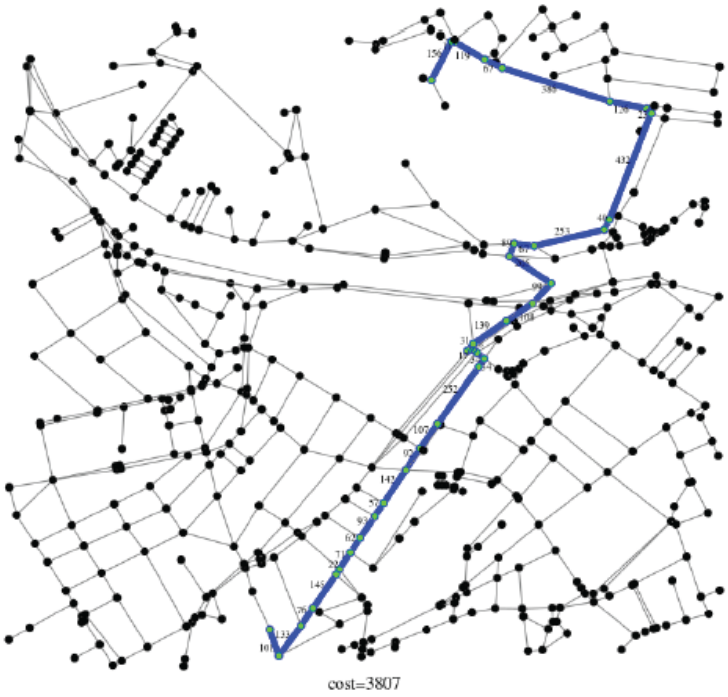
Figure 8 Rio de Janeiro popular locations (continued)



(d) *Gávea* map, 490 nodes



(e) *Centro* map, 432 nodes



(f) *Maracanã* map, 482 nodes

5 Conclusions and ideas for future work

This paper presented an efficient pathfinding system for edge/fog computing, augmented with a Dijkstra shortest path co-processor suitable for FPGAs with embedded ARM microprocessors using the Advanced Microcontroller Bus Architecture (AMBA4) specification. The co-processor was designed, implemented and evaluated in a Zynq FPGA from Xilinx. Also, this paper analysed the feasibility of extending the hardware of FPGA-based embedded systems 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 pathfinding co-processors, not only due to the low-dynamic energy consumption but mainly due to its high performance, being up to almost 3× faster than the ARM microprocessor alone. 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 usual in any integrated circuit design process. Therefore, especially for embedded systems based on ARM micro-processor architecture, the adoption of co-processors can allow the development of more complex applications 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 than three co-processors and larger graphs.

Although the input graphs are now limited to 500 nodes, the edge/fog node of the edge/fog computing architecture can be placed in a geographic region which can be fit onto the accelerator. Thus, autonomous vehicles inside such regional area would have their routes computed by the edge accelerator, closer to the vehicles, instead of computing them on distant cloud services. The pathfinding system can be embedded into the autonomous vehicle system itself, enabling them to communicate to other edge node vehicles or edge stations too. Besides, newer FPGAs can possibly fit more resources, enabling the computation of paths on graphs based on larger regions.

In the future, the co-processor will be implemented on the FPGA chip of an Alpha-Data PCI-Express board, which is equipped with SATA interfaces that allows the communication to a SSD disk, where the large-scale graphs will sit. Also, we plan to 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 master interfaces. Also, arbitrary precision data types will be introduced 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 values. Moreover, the co-processor is planned to be implemented directly in VHDL to better enable us to optimise the RTL architecture produced by the HLS compiler as well as to compare each architecture in terms of its performance, circuit-area and energy consumption.

References

Ahn, S., Gorlatova, M. and Chiang, M. (2017) 'Leveraging fog and cloud computing for efficient computational offloading', *Proceedings of the IEEE MIT Undergraduate Research Technology Conference (URTC)*, IEEE, USA, pp.1–4.

Ai, Y., Peng, M. and Zhang, K. (2017) 'Edge cloud computing technologies for internet of things: a primer', *Digital Communications and Networks*, Vol. 4, No. 2, pp.77–86.

AXI Reference Guide (2017) Available online at: `https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf` (accessed on 08 August 2017).

Boeing, G. (2016) 'Osmnx: new methods for acquiring, constructing, analyzing, and visualizing complex street networks', *Clinical Orthopaedics and Related Research*, Vol. 65, pp.126–139.

Bose, M. and Rajagopala, V. (2012) 'Physics engine on reconfigurable processor – low power optimized solution empowering next-generation graphics on embedded platforms', *Proceedings of the 17th International Conference on Computer Games (CGAMES)*, IEEE, USA, pp.138–142.

Cerina, L., Notargiacomo, S., Paccanit, M.G. and Santambrogio, M.D. (2017) 'A fog-computing architecture for preventive healthcare and assisted living in smart ambients', *Proceedings of the IEEE 3rd International Forum on Research and Technologies for Society and Industry (RTSI)*, pp.1–6.

Clow, J., Tzimpragos, G., Dangwal, D., Guo, S., McMahan, J. and Sherwood, T. (2017) 'A pythonic approach for rapid hardware prototyping and instrumentation', *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, Belgium, pp.1–7.

Consortium, O. (2017) *Openfog Reference Architecture For Fog Computing*. Available online at: www.`OpenFogConsortium.org`

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2009) *Introduction to Algorithms,* 3rd ed., The MIT Press.

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

da Silva, P.V.F. and Villela, S.M. (2016) 'Applying pathfinding techniques on the development of an android game', *Proceedings of SBGames 2016*, SBC, pp.73–80.

Dijkstra, E.W. (1959) 'A note on two problems in connexion with graphs', *Numer. Math.* Vol. 1, No. 1, pp.269–271.

Dolgov, D., Thrun, S., Montemerlo, M. and Diebel, J. (2009) *Path Planning for Autonomous Driving in Unknown Environments*, Springer, Berlin, Heidelberg, pp.55–64.

Dunkels, A. (2001) 'Design and implementation of the lwip tcp/ip stack', *Swedish Institute of Computer Science*.

Fabry, J. and Sinclair, S. (2016) 'Interactive visualizations for testing physics engines in robotics', *Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT)*, IEEE, USA, pp.106–110.

Hamano, T., Onosato, M. and Tanaka, F. (2016) 'Performance comparison of physics engines to accelerate house-collapsing simulations', *Proceedings of the IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, IEEE, Switzerland, pp.358–363.

Hart, P.E., Nilsson, N.J. and Raphael, B. (1968) '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.

Idris, M.Y.I., Bakar, S.A., Tamil, E.M., Razak, Z. and Noor, N.M. (2009) High-speed shortest path co-processor design', *Proceedings of the 3rd Asia International Conference on Modelling Simulation*, IEEE, Indonesia, pp.626–631.

Itoh, H. (2016) 'Development of lego mindstorms model construction system on omegaspace platform with physx functions', *Proceedings of the 11th France-Japan 9th Europe-Asia Congress on Mechatronics (MECATRONICS) /17th International Conference on Research and Education in Mechatronics (REM)*, IEEE, France, pp.38–43.

Jagadeesh, G.R., Srikanthan, T. and Lim, C.M. (2011) 'Field programmable gate array-based acceleration of shortest-path computation', *IET Computers Digital Techniques*, Vol. 5, No. 4, pp.231–237.

Kok, J., Gonzalez, L.F. and Kelson, N. (2013) 'Fpga implementation of an evolutionary algorithm for autonomous unmanned aerial vehicle on-board path planning', *IEEE Transactions on Evolutionary Computation*, Vol. 17, No. 2, pp.272–281.

Koromilas, E., Stamelos, I., Kachris, C. and Soudris, D. (2017) 'Spark acceleration on fpgas: a use case on machine learning in pynq', *Proceedings of the 6th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pp.1–4.

Lee, C.Y. (1961) 'An algorithm for path connections and its applications', *IRE Transactions on Electronic Computers* Vol. EC-10, No. 3, pp.346–365.

Lee, E-K., Gerla, M., Pau, G., Lee, U. and Lim, J-H. (2016) 'Internet of vehicles: from intelligent grid to autonomous cars and vehicular fogs', *International Journal of Distributed Sensor Networks* Vol. 12, No. 9, doi: 10.1177/1550147716665500.

Liu, C., Ji, X., Cao, Y., Xu, Q. and Chen, L. (2012) 'Phusis cloth: a physics engine for real-time character cloth animation', *Proceedings of the 2nd International Conference on Computer Science and Network Technology*, IEEE, China, pp.1578–1582.

Liu, L., Chang, Z., Guo, X., Mao, S. and Ristaniemi, T. (2018) 'Multiobjective optimization for computation offloading in fog computing', *IEEE Internet of Things Journal*, Vol. 5, No. 1, pp.283–294.

Jin, J., Yuan, D. and Zhang, H. (2018) 'Virtual fog: a virtualization enabled fog computing framework for internet of things', *IEEE Internet of Things Journal*, Vol. 5, No. 1, pp.121–131.

Menon, V.G. (2017) 'Moving from vehicular cloud computing to vehicular fog computing:issues and challenges', *International Journal on Computer Science and Engineering (IJCSE)*, Vol. 9, No. 2, pp.14–18.

Millington, I. and Funge, J. (2009) *Artificial Intelligence for Games*, 2nd ed., CRC Press.

Nery, A.S., Sena, A.C. and Guedes, L.S. (2017) 'Efficient pathfinding co-processors for fpgas', *Proceedings of the International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*', IEEE, Brazil, pp.97–102.

OpenStreetMap contributors (2017) *Planet dump*. Available online at: https://planet.osm.org, https://www.openstreetmap.org

Tan, T.H., Ooi, C.Y. and Marsono, M.N. (2017) 'hpfog: a fpga-based fog computing platform', *Proceedings of the International Conference on Networking, Architecture, and Storage (NAS)*, IEEE, China, pp.1–2.

Togelius, J. and Yannakakis, G.N. (2016) 'General general game AI', *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*', IEEE, Greece, pp.1–8.

Uliana, D., Kepa, K. and Athanas, P. (2013) 'Fpga-based hpc application design for non-experts', *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*', pp.9–15.

Umuroglu, Y., Morrison, D. and Jahre, M. (2015) 'Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform', *Proceedings of the 25th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, UK, pp.1–8.

Vaughn-Nichols, S.J. (2009) 'Vendors draw up a new graphics-hardware approach', *Computer*, Vol. 42, No. 5, pp.11–13.

Vivado (2017) *Ultrafast high-level productivity design methodology guide*. Available online at: https://www.xilinx.com/support/documentation/sw_manuals/ug1197-vivado-high-level-productivity.pdf (accessed on 08 August 2017).

Yang, H. (2008) 'Floating-point reconfiguration array processor for 3d graphics physics engine', *Proceedings of the Asia and South Pacific Design Automation Conference*, IEEE Computer Society Press, USA, pp.283–283.

Yeh, T.Y., Faloutsos, P., Patel, S.J. and Reinman, G. (2007) 'Parallax: an architecture for real-time physics', *SIGARCH Computer Architecture News*, Vol. 35, No. 2, pp.232–243.

Zhou, S., Chelmis, C. and Prasanna, V.K. (2016) 'High-throughput and energy-efficient graph processing on fpga', *Proceedings of the IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp.103–110.