Efficient Pathfinding Co-processors for FPGAs

Alexandre S. Nery, Alexandre C. Sena and Leandro A. J. Marzulo
Dep. de Informática e Ciência da Computação
Instituto de Matemática e Estatística - IME
Universidade do Estado do Rio de Janeiro - UERJ
Rio de Janeiro, Brazil
Email: {anery, asena, leandro}@ime.uerj.br

Leandro S. Guedes
Dep. de Informática
Instituto Federal de Educação, Ciência e
Tecnologia de Mato Grosso do Sul - IFMS
Corumbá, Brazil
Email: leandro.guedes@ifms.edu.br

***Abstract*—Pathfinding algorithms are at the heart of several classes of applications, such as network appliances (routing), GPS navigation and autonomous cars, which are related to recent trends in Artificial Intelligence and Internet of Things (IoT). Moreover, advances in semiconductor miniaturization technology have enabled the design of efficient Systems-on-Chip (SoC) devices, with demanding performance requirements and energy consumption constraints. Such systems might include Field Programmable Gate Arrays (FPGAs) to allow the design of customized co-processors that yield lower power consumption and higher performance. Therefore, this work aims at designing and evaluating four efficient pathfinding co-processors, each one implementing a different well-known pathfinding algorithm: breadth-first, dijkstra, greedy and a-star. Each co-processor is designed using Xilinx High-Level Synthesis (HLS) compiler and is implemented in the programming logic of a Xilinx FPGA embedded with an ARM microprocessor, which is in charge of controlling the set of co-processors. Extensive performance, circuit-area and power consumption results shows that each co-processor can efficiently execute a pathfinding algorithm, paving the way for novel dedicated accelerators.**

I. Introduction

Many modern embedded systems, such as smartphones and tablets, are equipped with in-house customized Multi-Processing Systems-on-Chip (MPSoC), often built around ARM multi-core architectures, such as Cortex-A9. This application-specific integrated circuit design includes inside a single chip key advanced components, like GPUs and communication modules, yielding higher efficiency and reduced power 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 re-configurable chips, allowing the extension of the ARM basic functions. Together with High-Level Synthesis (HLS) compiling tools [1] 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.

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 [2]. Lately, pathfinding algorithms have an important role in the implementation of autonomous vehicles [3]. 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.

In this paper, we propose four efficient pathfinding co-processors suitable for FPGAs with embedded ARM microprocessors. They were designed, implemented and evaluated in a Zynq Field-Programmable Gate Array (FPGA) from Xilinx [4]. The Zynq architecture 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 power efficient part.

The aim of this paper is to show the feasibility of extending the hardware of embedded devices to execute pathfinding algorithms and to evaluate the efficiency of the RTL hardware produced using Xilinx HLS compiler. Furthermore, our implementations not only allow a fast execution of four distinct types of pathfinding algorithms but includes low energy consumption. More importantly, more than one path can be calculated in parallel as more co-processors can be fit into the hardware design, allowing the development of more complex applications in more than ever constrained embedded devices.

The rest of this paper is organized as follows: Section II describes related works. The pathfinding algorithms implemented in the co-processor are presented in Section III. Section IV explains the hardware architecture and the pathfinding co-processors implemented. Experimental results are presented in Section V. Finally, Section VI concludes and presents ideas for future work.

II. 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 microprocessor. They are often designed for applications with high-potential for parallelism exploitation, such as physics simulation [5], [6], [7], clothing [8], 3D-object collision [9], model-based robotics [10], [11] and others [12], [13].

Pathfinding is also a common operation found in artificial intelligence, networking and route planning applications, such as autonomous vehicles [3], [14]. An evaluation of pathfinding algorithms has been presented in [15], which analyzes the Breadth-first, Depth-First, Ordered, Greedy and A* algorithms on Android platforms. Results show that heuristics-based

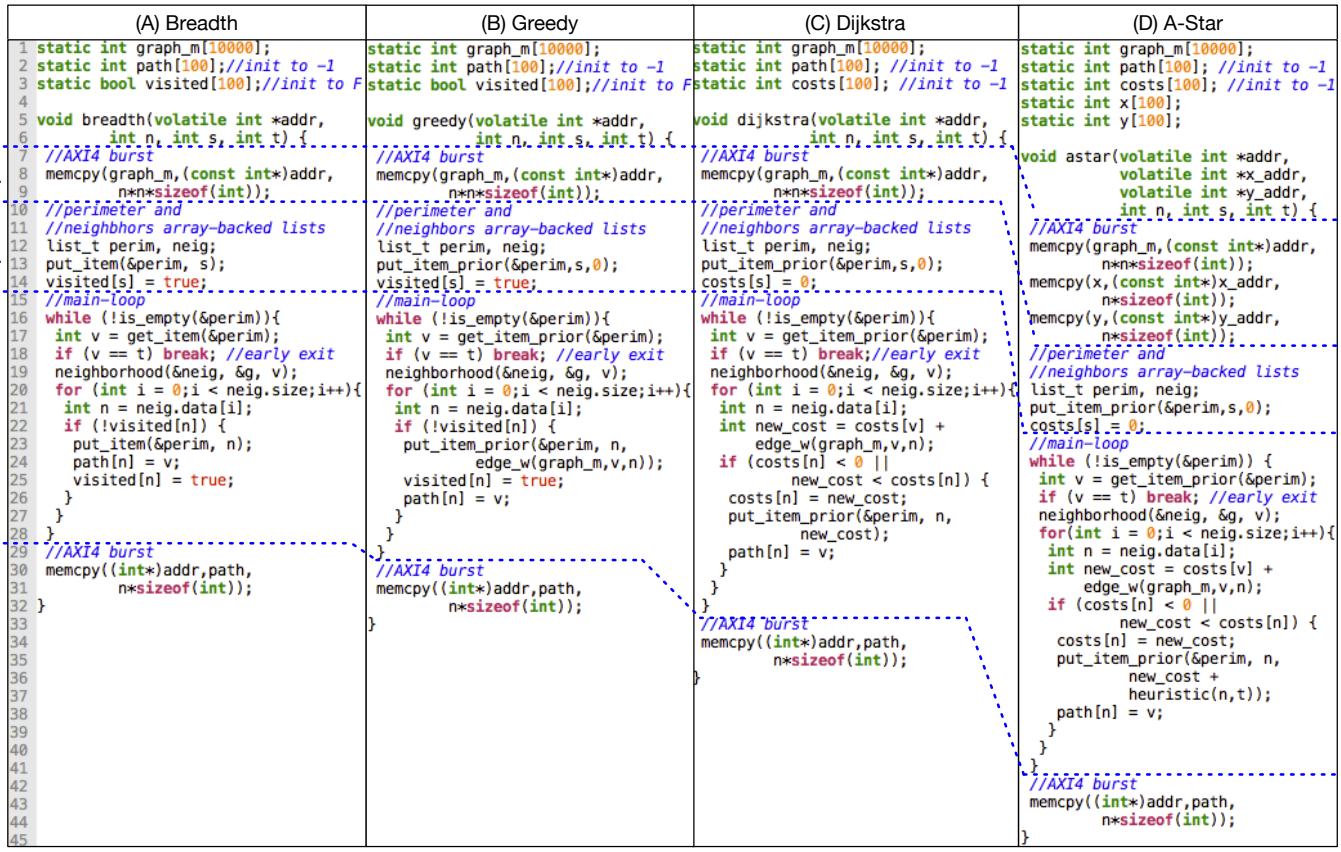

Fig. 1: Breadth-First Search (BFS), Greedy (GRD), Dijkstra (DIJ) and A-star (A*) pathfinding C++ snippets, compatible with Xilinx Vivado HLS. Interface protocol synthesis directives are shown only in Fig. 3, for the sake of organization. Memory function `memcpy` is used only to indicate AXI4 bus burst operation mode.

methods, such as A*, are more efficient in terms of execution time.

An implementation of a FPGA Bellman-Ford pathfinding algorithm can be seen in [16]. 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. It 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 impossible to compare to the work presented here.

The work in [17] describes a 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.

III. Pathfinding Techniques in HLS

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.

Fig. 1 depicts four panels (A,B,C and D), each one presenting Vivado HLS compatible C/C++ snippet codes of the aforementioned algorithms, respectively. The snippets are further organized into five parts. Part 1 presents static global arrays which are often mapped onto FPGA BlockRAMs. Most arrays are power-on initialized, except for the graph adjacency and resulting path arrays, which get their values from the external DDR memory connected to the ARM host processor, that provides the starting base memory address, as shown in Part 2. Observe that memory functions like `memcpy` are specifically used in Vivado HLS to indicate AXI4 bus burst operation, as will be shown in Section IV. Moreover, the A* specification includes two additional arrays and their external base addresses. These arrays hold the `(x,y)` position of each node and are used by the A* heuristic to compute the Manhattan distance between a pair of nodes (s, t), as follows:

$$abs(x[s] - x[t]) + abs(y[s] - y[t]) \quad (1)$$

Besides the static arrays, Part 3 presents array-backed lists that are used by the algorithm's main loop, described in Part 4. 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 brevity), it suffices to say that each can hold up to 50 integer elements. 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 the each list have an impact on the search criteria of each algorithm, often leading to a faster path search. For instance, the Breadth First Search (BFS) 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 (GRD) 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 (DIJ) tends to prioritize 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 produces the optimum solution (smallest path). Finally, A-star (A*) operates like Dijkstra. The key difference lies in the use of a heuristic to guide the search towards the nodes which are closer to the goal, avoiding the nodes which are farther from the goal.

Lastly, Part 5 presents the bus burst operation of the resulting path back to the ARM host microprocessor, starting on the same base address of the input adjacency array.

IV. The Co-Processor Architecture

The Zynq-FPGA architecture is split into Processing System (PS) and Programmable Logic (PL) parts, as shown in Fig. 2. 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.

The PS-PL communication interface operates according to the Advanced eXtensible Interface (AXI4) protocol, which is part of the AMBA4 specification [18]. The PS available interfaces are: $4\times$ General Purpose (GP) AXI master/slave ports, $4\times$ 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

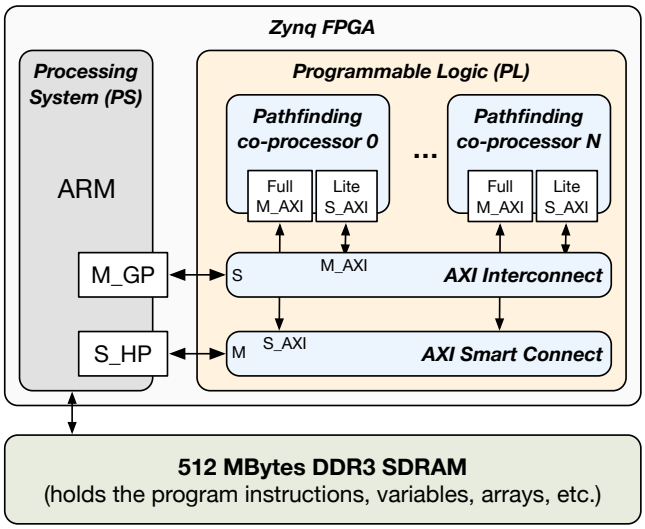

Fig. 2: The Zynq System-on-Chip re-configurable architecture augmented with pathfinding co-processors using AXI4-Full and AXI4-Lite interfaces.

(GP) interfaces are used for control signals and tolerable high-latency pieces of data. The proposed co-processors interface connects to both GP and HP interfaces of the PS.

Each pathfinding co-processor is designed using Xilinx HLS compiler, which transforms a C specification into a RTL implementation suitable for running into Xilinx FPGAs, as shown in Fig. 3. In general, the HLS compiler synthesizes C functions into blocks in the RTL hiearchy, 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 (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,

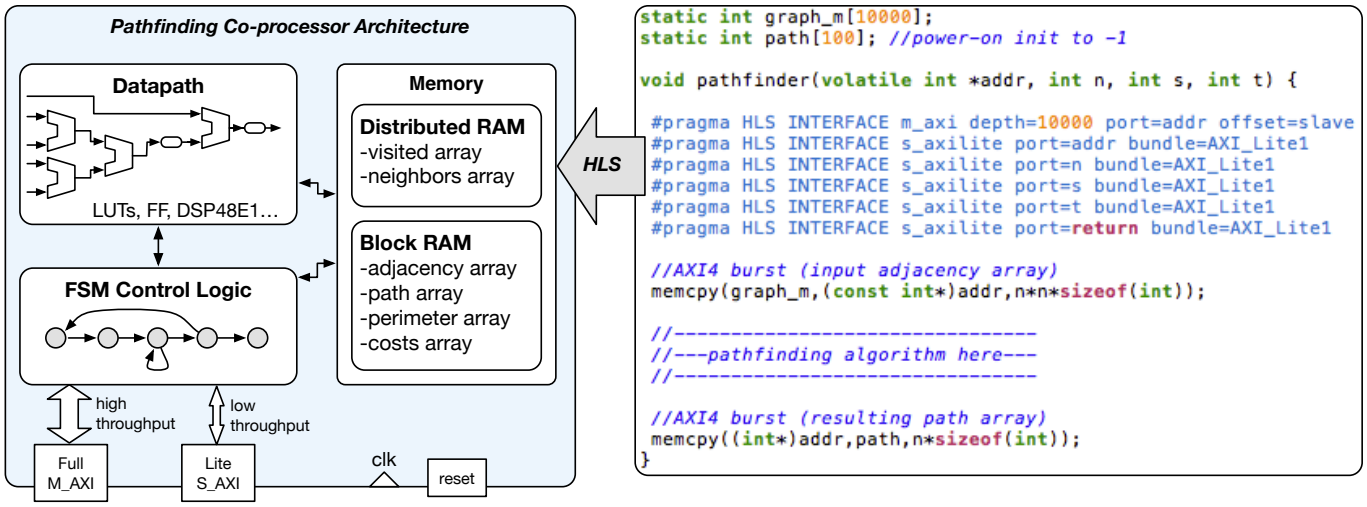

Fig. 3: The Pathfinding General Co-Processor Architecture and its HLS interface specification.

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 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 address of the adjacency array to the core.

Thus, given the base address `*addr` and the compiler directives (*i.e.*, pragmas), as shown in the snippet code in Fig. 3, the HLS compiler implements an AXI4-Full interface port which is used to transfer the adjacency array data and the resulting path back to the same address. Also, an AXI4-Lite interface port is implemented to transfer the address itself and to transfer 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 (`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 PS-PL co-processors. 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.).

V. Experimental Analysis

The design process of each 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 protocols (AXI4-Lite, AXI4-Full and/or AXI4-Stream). The last stage is the processing system development, which builds on top the designed architecture. All co-processors presented in this work had to go through all the previous stages of development. All the experiments were executed in a Xilinx Zynq-7000 FPGA (XC7Z010-1CLG400C) within Digilentinc Zybo board [19]. The input graph is randomly generated and limited to 100 nodes. This limitation is to ensure that more than one accelerator can be fit into the FPGA, enabling the evaluation of more than one path in parallel. The number of edges 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. However, the more edges the graph contains the wider the array-backed lists needs to be to accommodate longer paths.

A. Performance analysis

In the first experiment set we evaluate the speedups of the algorithms when executing in the FPGA. As can be seen in Fig. 4, we did not only compare the speed of the algorithms in the FPGA with respect to the execution in the ARM processor but also executed up to three co-processors in the FPGA. When executed with only one co-processor, the results were very similar, with the Greedy algorithm being slightly better. The Dijkstra algorithm is more computing intensive because it calculates the optimum path, while the A* needs to compute a heuristic (Manhattan distance) cost function to guide the search, which harms its performance. More importantly, all co-processors are almost twice faster when compared to the ARM alone.

The performance gain is even better when we analyze the results using two or three co-processors. For instance, with two co-processors, all algorithms doubled performance, except the Breadth co-processor, which increased two and a half times the pathfinding performance. More interesting, when executing with 3 co-processors, the performance of the algorithms quadrupled. This performance leap possibly occurs because the paths which are computed by the two extra co-processors are smaller than the path computed by the first co-processor. Moreover, it was not possible to fit a third A* with

co-processor, limiting its execution to a pair of co-processors.

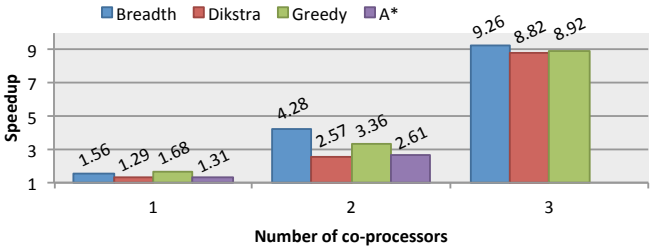

Fig. 4: Speedups of each co-processor with respect to the same algorithm executed by the ARM alone. Notice that the third A* co-processor could not be fit into the FPGA.

B. Circuit-area analysis

When considering the different resources of the programmable logic (FPGA), the BRAM was the most used resource, as presented in Fig. 5. 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 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.

Followed by the BRAM, the Look-Up Table (LUT) was the second most used resource, while the Flip-Flops were the third. 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.

Regarding the circuit-area occupied by each algorithm, the BFS co-processor clearly occupies the smallest area, while A* occupies the largest area. Due to the size of the area occupied by the algorithm A*, it was only possible to fit up to two A* co-processors, while for the others, it was possible to fit up to three co-processors.

C. Power requirement analysis

One of the main advantages of using a co-processor oriented design is its low energy cost. As can be seen in Fig. 6, for all implementations the processing system (ARM) is the most energy-consuming part (approximately 1.561 Watts for all algorithms). On the other hand, the programmable logic (FPGA) is responsible for less than $1/4$ of the energy consumed for all scenarios. The most efficient design was breadth, consuming 2.03, when considering only one co-processor. The main reason for such efficiency is because of the array-backed lists `perim` and `neig`, which are used to store the list of nodes that are being visited and the list of neighbors of each node being visited, do not need to be sorted. Thus, inserting and removing an item into and from theses lists takes O(1), while sorted lists would have a worst-case time complexity of O(n).

An interesting characteristic of the pathfinding co-processors is that their energy consumption does not increase proportionally with the increase of their use. For example, the execution of three Dijkstra co-processors only increased the energy consumption in 65%. More important, as was shown in Fig. 4, the performance was four times better than with one co-processor.

VI. Conclusion & Ideas for Future Work

This paper presented four efficient pathfinding co-processors suitable for FPGAs with embedded ARM microprocessors using the Advanced Microcontroller Bus Architecture (AMBA4) specification. The co-processors were designed, implemented and evaluated in a Zynq Field-Programmable Gate Array (FPGA) from Xilinx. Also, up to three co-processors could fit into the FPGA, enabling the computation of up three paths in parallel. Moreover, this paper analyzed 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 energy consumption (bellow 2.5 Watts) but mainly due to its high performance, being up to $9\times$ faster than the ARM microprocessor alone, for an input graph of up to 100 nodes. Although the co-processors 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, especially for embedded systems based on ARM microprocessor 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.

In the future, each 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 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, each co-processor is planned to be implemented directly in VHDL to better enable us to optimize 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.

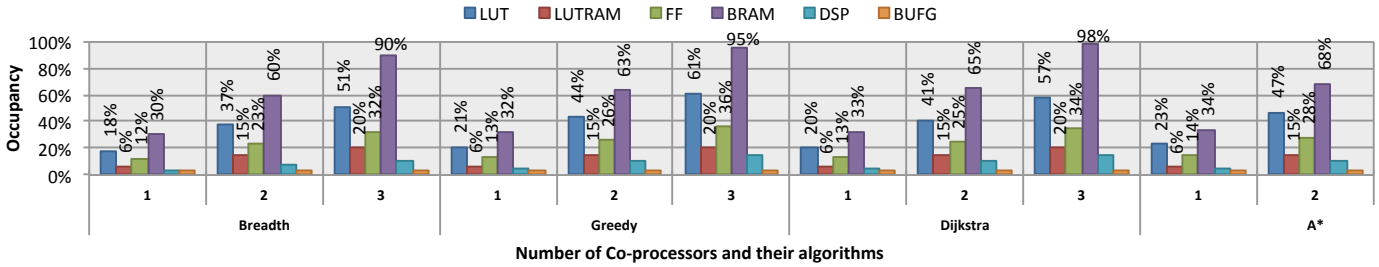

Fig. 5: FPGA occupancy results. Only the main data values are displayed.

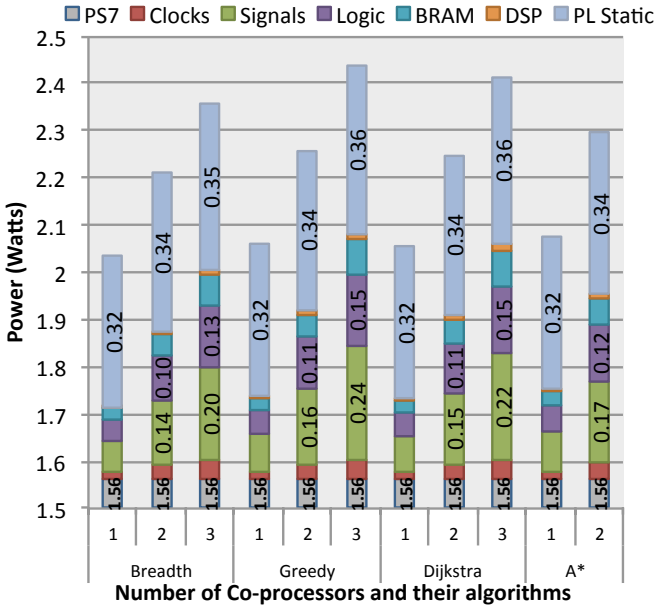

Fig. 6: Power analysis (in Watts), with the X-axis representing the number of co-processors per algorithm HLS implementation. Only data values above 0.1W are displayed.

ACKNOWLEDGMENTS

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] "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.

[2] J. Togelius and G. N. Yannakakis, "General general game ai," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, Sept 2016, pp. 1–8.

[3] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel, *Path Planning for Autonomous Driving in Unknown Environments*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 55–64. [Online]. Available: https://doi.org/10.1007/978-3-642-00196-3_8

[4] "Zynq-7000 all-programmable technical reference manual," https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, Xilinx, accessed: 08-08-2017.

[5] 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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] J. Kok, L. F. Gonzalez, and N. Kelson, "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, April 2013.

[15] 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.

[16] 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.

[17] 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.

[18] "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.

[19] "Zybo reference manual," https://reference.digilentinc.com/reference/programmable-logic/zybo/reference-manual, Xilinx, accessed: 08-09-2017.