Field programmable gate array-based acceleration of shortest-path computation

Jagadeesh, George Rosario; Srikanthan, T.; Lim, C. M.

2011

Jagadeesh, G. R., Srikanthan, T., & Lim, C. M. (2011). Field programmable gate array-based acceleration of shortest-path computation. IET Computers and Digital techniques, 5(4), 231 - 237.

<https://hdl.handle.net/10356/104263>

<https://doi.org/10.1049/iet-cdt.2009.0072>

© 2011 The Institution of Engineering and Technology. This is the author created version of a work that has been peer reviewed and accepted for publication by IET Computers and Digital techniques, Institution of Engineering and Technology. It incorporates referee's comments but changes resulting from the publishing process, such as copyediting, structural formatting, may not be reflected in this document. The published version is available at: [<http://dx.doi.org/10.1049/iet-cdt.2009.0072>].

Downloaded on 26 Apr 2023 17:55:35 SGT

FPGA-based Acceleration of Shortest Path Computation

G. R. Jagadeesh, T. Srikanthan and C. M. Lim

Centre for High Performance Embedded Systems

School of Computer Engineering

Nanyang Technological University

Singapore 639798

Email: {asgeorge, astsrikan, S8018826I} @ntu.edu.sg

Abstract

There exist several practical applications that require high-speed shortest path computations. In many situations, especially in embedded applications, an FPGA-based accelerator for computing the shortest paths can help to achieve high performance at low cost. This paper presents an FPGA-based distributed architecture for solving the single-source shortest path problem in a fast and efficient manner. The proposed architecture is based on the Bellman-Ford algorithm adapted to facilitate early termination of computation. One of the novelties of the architecture is that it does not involve any centralized control and the processing elements, which are identical in construction, operate in perfect synchronization with each other. The functional correctness of the design has been verified through simulations and also in actual hardware. It has been shown that the implementation on a Xilinx Virtex-5 FPGA is more than twice as fast as a software implementation of the algorithm on a high-end general-purpose processor that runs at an order-of-magnitude faster clock. The speed-up offered by the design can be further improved by adopting an interconnection topology that maximizes the data transfer rate among the processing elements.

Keywords: Graph algorithms, Shortest path computation, Parallel architectures, FPGA implementation.

1. Introduction

High-speed computation of shortest paths in a graph from a given source to one or more destinations is a vital requirement in many application domains such as intelligent transportation systems, robotics, VLSI computer-aided design and computer gaming. Highly compute-intensive applications such as urban traffic simulation typically utilize high-performance computing systems with powerful general-purpose processors for fast shortest path computations. However, in many other situations, especially in embedded applications, it is not feasible to utilize powerful general-purpose computing systems and cost-effective alternatives are desirable. For instance, cost and energy concerns rule out the use of high-end,

high-speed processors in portable navigation devices (PND), which nevertheless have to perform fast route computation in addition to several other compute-intensive real-time tasks.

A number of well-established algorithms [1] exist for solving the single-source shortest path problem through software running on a uniprocessor system. The demand for accelerating the shortest path computation has spawned research in many different directions including exploring efficient data structures [2] and the use of heuristics [3]. A comprehensive performance comparison of the various shortest path methods can be found in [4]. Another active area of research is the development of parallel shortest path algorithms [5,6,7] targeted towards parallel random access machines (PRAMs) where multiple sequential processors execute the same program in parallel and communicate through a shared global memory. While the PRAM is a useful abstract model for parallel computation, it is difficult to realize such machines in practice mainly due to the restrictions in accessing the global memory.

Field programmable gate array (FPGA) based application-specific hardware architectures have been shown to outperform programmable processor-based implementations in the case of computationally complex problems. The recent significant increases in the FPGA gate counts and clock rates have made it possible to realize highly parallel implementations of shortest path algorithms for relatively larger graphs. This paper presents an FPGA-based architecture that exploits the distributed nature of the Bellman-Ford single-source shortest path algorithm [1]. It is intended to serve as a hardware accelerator, whose services can be utilized by a host system for high-speed shortest path computation.

In the proposed architecture, where computations relating to a node in the graph are performed in parallel with those of all other nodes, the size of the FPGA limits the maximum size of the graph that can be handled. However, it has been established that path search problems in large networks can be broken down into multiple path searches in smaller sub networks using the concept of hierarchical abstraction [8, 9, 10]. For instance, our previous work with transportation networks [10] has shown that it is possible to compute near-optimal paths in a network with over 11000 nodes by reducing it to a series of shortest path computations in sub networks, whose size is limited to 128 nodes. Therefore, in the case of large networks, the FPGA-based accelerator can be used in conjunction with an appropriate hierarchical path-finding algorithm.

In the proposed setup, the FPGA-based circuit for shortest path computation is generic enough to handle any input graph provided the number of nodes in it do not exceed the maximum size limit. This is in contrast to the approach adopted in [11], where a unique

hardware circuit is generated for each input graph instance. Such approaches require the FPGA to be reconfigured with a newly generated circuit if the input graph changes. This in turn requires relatively more time for recompilation and configuring the FPGA. In our proposed framework, the FPGA needs to be configured only once and the same circuit can be used repeatedly for computing the shortest paths in different input graphs.

In the following section we examine the various FPGA-based architectures for the single-source shortest path problem proposed in the literature and contrast our approach with them. Subsequently in Section III, we describe a variation of the Bellman-Ford shortest path algorithm, which is an ideal candidate for hardware implementation due to its inherent parallelism. In Section IV, we describe the proposed distributed architecture and its operation. Section V presents the details of the FPGA implementation along with the area and timing results. Performance comparisons are made with a software implementation on a state-of-the-art general-purpose processor as well as with a comparable FPGA-based architecture reported in the literature. Finally concluding remarks are made in Section VI.

2. Related Work

Most FPGA implementations of shortest path algorithms reported in the literature conform to one of the following approaches. The first approach [11,12,13] is based on the idea that since graphs naturally correspond to circuits, it is beneficial to construct a circuit that resembles the graph topology. Babb *et al.* [11] describe a compilation technique that accepts a topological representation of a specific user input graph instance and generates a circuit that resembles the graph such that nodes correspond to logic and links correspond to wires. The circuit facilitates the computation of shortest paths by implementing logic based on the Bellman-Ford algorithm. However, this circuit representation of the graph is static and as stated earlier, any change in the input graph will require expensive recompilation and reconfiguration.

Huelsbergen [12] sought to introduce some flexibility into this method of circuit construction by providing for the insertion and deletion of nodes/links on the graph without having to regenerate a new circuit. The adjacency matrix of the graph used to represent the connectivity information is stored in the cells within the FPGA. Changes to the graph topology are achieved by selectively rewriting new values to the cells. For this, the FPGA needs to be partially reconfigurable. This work was further extended by Mencer *et al.* [13] who proposed that each cell corresponding to an entry in the adjacency matrix be made to drive a tri-state

buffer to make or break a connection. This method of circuit representation requires hardware space quadratic in size of the set of nodes. Mencer *et al.* also introduced a scaling scheme for realizing circuits corresponding to large graphs by splitting them into multiple contexts and using the lookup tables (LUTs) within the FPGAs as context memories. A major limitation of the architectures [12,13] described above is that they can only compute the shortest unit path. In other words, all the links are assumed to be of unit weight and the ‘shortest path’ found using this circuit is synonymous with the minimum-hop path.

The second approach [14] adopted for hardware implementation of shortest path algorithms employ multiple processing elements (PEs) such that each PE corresponds to a node in the graph. Information about the links is stored in an external memory such that each link is represented using three values namely the index of the start node, the index of the end node and the weight of the link. A completely different graph topology can be supplied to the FPGA-based accelerator as input by changing the contents of the external memory. The FPGA’s configuration does not need to be modified as long as the number of nodes in the graph does not exceed the number of PEs. Our proposed architectural scheme is based on this approach as it offers maximum flexibility.

The architecture proposed by Dandalis *et al.* [14], which solves the single-source shortest path problem using the Bellman-Ford algorithm, is the closest to our proposed architecture. In this design, the PEs corresponding to the nodes in the graph are arranged as a pipeline within the FPGA. The shortest path computation is performed by feeding each link data through the pipeline. The Bellman-Ford algorithm described in the next section involves an iterative process where the status of all the nodes in the graph are updated during each iteration. Since the design proposed in [14] requires all the links to be passed serially through the pipeline, the number of clock cycles required to perform one iteration is at least equal to the number of links in the graph. In contrast to this pipelining scheme, we propose a novel design where each PE receives and processes data in parallel such that the number of clock cycles required to perform one iteration in the Bellman-Ford algorithm is just one more than the number of nodes. Since the number of links in a typical graph is several times the number of nodes, our design computes the shortest paths in significantly less clock cycles.

An FPGA-based architecture, which does not conform to any of the two broad approaches described above, has been proposed by Tommiska and Skytta [15]. This design is based on Dijkstra’s shortest path algorithm [1], which does not permit the status of all the nodes in the graph to be updated in parallel. Therefore, in this architecture the parallelism is limited to

processing all the links connected to a given node using a parallel comparator bank. Baker and Gokhale [16] presented a parallel architecture in which several shortest path cores based on the A* algorithm are implemented on the FPGA such that each core is responsible for one shortest path calculation. However, unlike the Bellman-Ford algorithm, the A* algorithm is incapable of finding the shortest paths from a given source to multiple destinations, as required by some applications. For example, a car navigation system may need to compute the travel distance from the car's position to several fuel stations before presenting the multiple options to the driver. With the A* algorithm, this would entail several shortest path computations to be performed sequentially.

3. The Bellman-Ford Algorithm and its Adaptation

The choice of a suitable shortest path algorithm for porting onto hardware is mainly dictated by the ability to parallelize its operations. Most attempts at hardware implementation of the single-source shortest path problem have been based on variations of the Bellman-Ford algorithm. Given a weighted, directed graph and a source node, it finds a shortest path from the source node to every other node. For a graph $G(V,E)$ with n nodes and e links, the pseudocode of the Bellman-Ford shortest path algorithm is presented below.

Bellman-Ford algorithm:

1. for each node $k \in V$
2. cost $[k] = \infty$
3. predecessor $[k] = \text{nil}$
4. cost $[s] = 0$
5. for $j = 1$ to $n-1$
6. for each link $(u,v) \in E$
7. temp = cost $[u] + w[u,v]$
8. if temp < cost $[v]$
9. cost $[v] = \text{temp}$
10. predecessor $[v] = u$

In the pseudocode above, s denotes the source node and the weight of a link from node u to node v is represented as $w[u,v]$. Each node in the graph maintains the cost of the shortest path from the source node to itself along with the index of the node that precedes it in the shortest path. During each iteration, the cost and predecessor node index of all the nodes with an incoming link are updated. This results in a shortest path tree with the source node as its root

being expanded such that during each iteration, its height is incremented by one hop. The runtime complexity of the Bellman-Ford algorithm is $O(ne)$ where n is the number of nodes and e is the number of links.

Line 5 in the pseudocode indicates that the algorithm involves $n-1$ iterations. This is based on the fact that a shortest path in an n -node graph can theoretically have $n-1$ hops. However, in actual practice, the maximum number of hops in any shortest path in a graph is far less than n . This observation has been confirmed through extensive software simulations [14]. Therefore, the iterations can be terminated as soon the algorithm converges resulting in a significant degree of speed-up. For graphs with no negative-weight links, the convergence of the algorithm can be confirmed if the status of none of the nodes is modified during an iteration. Other single-source shortest path algorithms such as Dijkstra's algorithm do not permit this property of graphs to be exploited to improve performance.

4. The Architecture

In the proposed setup shown in Figure 1, the FPGA-based design serves as a shortest path computation unit (SPCU), whose services are utilized by a host system when necessary. The connectivity information of the input graph is represented in the form of links in the host system's memory in any random order. Data corresponding to each link consists of the index of the start node, the index of the end node and the weight of the link. Since the SPCU serves as a slave co-processor, the host issues instructions to the SPCU through a 2-bit-wide instruction bus. Data transfer between the host and the SPCU occur through a bi-directional data bus, whose width depends on the number of bits required to represent a node (which in turn depends on the maximum number of nodes) and the number of bits required to represent a link's weight. In the present implementation, we use 8 bits to represent the weight of a link.

The host issues one of the four instructions listed in Table 1 to the SPCU through the instruction bus. Before invoking the SPCU, the host has to first supply each link in the input graph along with the instruction code '00'. Subsequently, it places the index of a source node on the data bus and tasks the SPCU to perform the shortest path computation by issuing the instruction code '01'. After this instruction, the host releases the data bus (by placing its data outputs in high-impedance state) though it continues to monitor (read) the bus' contents. The SPCU signals the completion of the shortest path computation by outputting a special value on the data bus. The host can then read the result of the computation by repeatedly issuing the instruction code '10'. The last instruction code '11' indicates that the host has released its

control of the bi-directional data bus for the internal use of the SPCU. The host also uses this instruction code as a separator between any two instructions.

4.1 The Shortest Path Computation Unit (SPCU)

As shown in Figure 1, the SPCU consists of multiple PEs such that each PE corresponds to a node in the graph. Each PE parallelly receives the instruction code from the host and the input data from the bi-directional data bus. However, at any given time only one PE can output data into the data bus. The rest of the PEs place their outputs in high-impedance state using internal tri-state buffers. The proposed architecture of the SPCU is completely distributed without any centralized control. All the PEs have identical architecture and each one operates independently in complete synchronization with the rest. Synchronization is made possible by ensuring that each PE starts the execution of each step at the same time. Although the same step is executed by all the PEs in parallel, they may perform different actions based on the result of a condition evaluation.

Before describing the internal architecture of a PE, a high-level description of the operation of the SPCU may be in order here. Upon receiving data corresponding to a link from the host, each PE checks if the index of the end node of the link is the same as the index of the node it represents. If so, the index of the start node of the link and the link's weight are stored in a RAM within the PE referred as the adjacency RAM. Thus, the input graph representation is distributed among the PEs such that each PE stores its incoming links in its adjacency RAM. Upon receiving the instruction for computing the shortest path, the status of all the PEs are initialized such that all the PEs other than the one which represents the source node have their cost value set to infinity (in practice, however, it is set to a very large value). The cost value of the source node is set to zero. As a one-time step, the contents of the adjacency RAM in each PE are sorted based on the index of the link's start node. Subsequently, the computation of the shortest path tree with the source node as its root is performed in an iterative manner.

Each iteration lasts $n + 1$ clock cycles, where n is the number of nodes in the graph (i.e., the number of PE's). Each PE maintains an internal counter to keep track of the clock cycle count during each iteration. During the i^{th} clock cycle, the PE corresponding to the i^{th} node broadcasts its cost to all other PEs through the data bus. Each PE checks if the index of the PE that transmitted the data is the same as that of the start node in its adjacency RAM location currently addressed. If so, it adds the weight of the corresponding link to the received cost and compares the result with its previously stored cost value. If the newly

computed cost is lesser, the cost of the PE is updated. Also the index of the PE, which transmitted the data, is assigned as the predecessor node. The PEs receive the data transmitted by another PE after a latency of one clock cycle and hence $n + 1$ clock cycles are required for each iteration instead of n . At the end of a sufficient number of iterations, the cost of the shortest path from the source node to each node and the predecessor node in the path are stored in the PE corresponding to the node.

The computation is terminated when the algorithm converges such that the status of none of the PEs is modified during an iteration. This condition is detected in the following manner. When it is the turn of a PE to broadcast its cost, it also broadcasts a signal (using one of the unused bits in the data bus) to indicate if its status was modified during the *previous* iteration. The rest of the PEs receive and process this information during the next clock cycle. Thus, at the end of the iteration, each PE would be able to decide if the status of any of the PEs was modified during the *previous* iteration. If the status of none of the PEs is modified, then the PE, which corresponds to the source node, outputs a special value on the data bus to indicate to the host that the shortest path computation has been completed. Thus, if the algorithm converges in m iterations, $m + 1$ iterations need to be performed as one extra iteration is required to detect the convergence.

The internal architecture of the PE is shown in Figure 2. It incorporates an adder, comparator and multiplexer in the data path for implementing the elementary operations in the Bellman-Ford algorithm. An adjacency RAM stores information about the incoming links of the node represented by the PE. An internal control unit, implemented as a state machine, interprets the instruction code received from the host and performs the necessary operations. It incorporates a counter for keeping track of the clock cycle count and logic for detecting the algorithm convergence. The logic required for sorting the contents of the adjacency RAM is implemented within the control unit. The control unit also causes data to be broadcast to the data bus when necessary by enabling the tri-state buffers that drive the PE's output.

5. Implementation and Results

The SPCU architecture described in the previous section was coded in VHDL and functionally verified the ModelSim simulator. Logic synthesis and physical implementation of the design were performed using the Xilinx Integrated Software Environment (ISE) design tools. Four versions of the SPCU were synthesized corresponding to graph sizes of 16, 32, 64 and 128. The designs were targeted towards the Xilinx Virtex-5 SX95T FPGA, which

contains 14720 configurable logic slices [17]. Each Virtex-5 FPGA slice contains 4 Look-Up Tables (LUT) and 4 flip-flops (FF).

The design utilizes only the configurable logic portion of the FPGA and does not make use of any on-chip dedicated resources available. The resource utilization data in terms of the number of LUT-FF pairs used and slices occupied for SPCUs of various sizes are presented in Table 2. The 128-node SPCU occupies only about 38% of the LUT-FF pairs present in the Virtex-5 SX95T FPGA and could be fitted into a smaller device. Some caution should be exercised while interpreting the FPGA slice usage as all the occupied slices may not be fully utilized (In some cases, only some of the four LUT-FF pairs within a slice may be utilized). Figure 3 presents the relationship between the SPCU size and the number of LUT-FF pairs. Since the number of PEs in the design is equal to the number of nodes in the graph, it is reasonable to expect the design to have a linear area complexity of $O(n)$. However, in practice, the area growth rate of the design is slightly superlinear. This is because the number of bits needed to represent a node is one of the factors that determine the area of an individual PE. Therefore, in a large-sized SPCU, each PE would occupy marginally more hardware area compared to a smaller-sized SPCU.

All the four versions of the SPCU have been synthesized to operate at the best possible clock frequency. The size of the SPCU appears to have an effect on the design's critical path delay such that larger-sized SPCUs result in a lower operating frequency. Referring to the architecture shown in Figure 2, the critical path of the 128-node SPCU extended from the data register to an internal register of the control unit and passed through the adder and the comparator. The maximum possible clock frequency ranged from 143 MHz for the 128-node SPCU to 180 MHz for the 16-node SPCU.

We have generated and used four two-dimensional rectangular grid graphs of the abovementioned sizes for testing the functional correctness of the SPCU architecture and to evaluate its performance. Such grid graphs, in which each node is connected to at most four of its neighbors, have been used in prior studies for evaluating shortest path algorithms [18,19] and are known to reflect the characteristics of most real-world networks. The link weights were randomly assigned in the range of 0 to 255. The size of the adjacency RAM in the SPCU limits the number of links that can be connected to a node (i.e., its in-degree). In the present implementation, we have set the maximum in-degree of a node to four. However, this should not be construed as a limitation since an increase in the maximum in-degree will require only a marginal increase in the circuit area and the computation time. In fact, the

speed-up offered by our design over other solutions increases with increase in the average in-degree.

The test graphs were incorporated into a VHDL testbench and the functionality of the design was verified using the ModelSim simulator. Also, the design was verified in actual hardware using an FPGA development board by configuring the FPGA with a 16-node SPCU. For the sake of convenience, a circuit emulating the host's behavior was also implemented on the FPGA itself. The shortest paths computed by the FPGA-based SPCU were verified to be accurate.

5.1 Performance Comparison

In the following, we present a breakdown of the time required by the SPCU architecture for computing the shortest paths from a given source node to all other nodes in the graph. We assume that the graph topology, in the form of link details, has already been supplied to the SPCU. The computation time can be broken into the following two parts.

- (a) The time required to perform the one-time step of sorting the contents of the adjacency RAM within each PE. If we denote the maximum in-degree as d , then using a simple bubble sort, the RAM of depth d can be sorted in $d(d-1)/2$ steps. Each step involves a compare-and-swap operation that takes 3 clock cycles. Therefore, the total number of clock cycles required for sorting the adjacency RAM of depth d is $3d(d-1)/2$. Since we have chosen the value of d as 4 in the present implementation, a constant time of 18 clock cycles is required for this one-time step.
- (b) The time required to perform $m + 1$ iterations, where m is the number of iterations required by the Bellman-Ford algorithm to converge. As explained in the previous section, each iteration lasts $n + 1$ clock cycles. Therefore, $(m+1) \times (n+1)$ clock cycles are required to perform the sufficient number of iterations.

Therefore, the total computation time of the SPCU in terms of clock cycles is given as

$$T_{\text{SPCU}} = 3d(d-1)/2 + (m+1)(n+1) \quad (1)$$

The above expression has been empirically verified to be accurate through simulations using the ModelSim simulator. The average number of iterations required for algorithm

convergence in each test graph, shown in Table 3, was determined through simulations involving a number of different source nodes.

The FPGA-based SPCU architecture was compared with a software implementation running on a powerful workstation. The Bellman-Ford algorithm was coded in C and executed on an Intel Pentium-4, single-core 3400 MHz processor with 4 GB physical memory. For the sake of fair comparison, the C program was also made to terminate upon algorithm convergence. Also, the time required to load the graph topology into the memory was excluded. It can be seen from the results in Table 4 that despite running at a clock about 20 times slower compared to the CPU, the FPGA-based architecture for shortest path computation is more than twice as fast as the CPU.

We also compare the performance of the SPCU with the FPGA-based architecture proposed by Dandalis *et al.* [14] that operates under conditions most similar to ours facilitating a fair comparison. This solution is also based on the Bellman-Ford algorithm and exploits its early convergence capability. Like our approach, this work requires the links stored in the memory to be fed one by one to the FPGA-based architecture. It is the only implementation in the literature that provides an expression for accurately determining the execution time for solving the single-source shortest path problem in terms of the number of clock cycles. In [14], the number of clock cycles required to compute the shortest path tree for a given source node in a graph with n nodes and e links is given as

$$T_D = (m+1)e + 2n \quad (2)$$

where m is the number of iterations required for algorithm convergence. Each iteration lasts e clock cycles and $2n$ clock cycles correspond to the latency required for traversing the pipeline of n PEs and back. The execution time estimated in this manner also includes the time required for supplying the links in the graph to the FPGA from the host system's memory.

In our implementation, transferring a link to the FPGA takes 2 clock cycles. Therefore, a total of $2e$ clock cycles are required to transfer all the links in the graph. In Table 5, we compare the total number of clock cycles required to transfer a graph to the SPCU and to perform one shortest path computation with the corresponding number for the architecture in [14]. A comparison in terms of absolute times is not possible as the maximum achievable clock frequency for [14] on a Xilinx Virtex-5 FPGA is not known. It could be seen from Table 5 that the SPCU takes significantly less clock cycles for all the cases. The speed-up is relatively low for the lower-sized SPCUs as the time required to supply the links to the SPCU overshadows the time required to perform the computation. It is worth noting that the links, once supplied, need not be supplied again for subsequent shortest path computations with different source nodes as long as the input graph remains unchanged.

6. Conclusions

There are a number of compute-intensive applications that strongly justify the need for high-speed shortest path computation. We have introduced an FPGA-based distributed architecture for solving the single-source shortest path problem. The proposed architecture seeks to exploit the parallelism inherent in the Bellman-Ford algorithm and its ability to facilitate early termination of computation. The architecture has been implemented on a Xilinx Virtex-5 FPGA and its functional correctness has been verified. With the help of simulations using different-sized test graphs, we have shown that the proposed architecture consistently outperforms a software implementation on a high-end general-purpose processor operating at an order-of-magnitude higher clock rate. The benefits of FPGA-based acceleration would be much more valuable in embedded applications where it is not feasible to utilize such high-end, high-speed processors. The performance of the SPCU has also been found to be superior to another comparable FPGA implementation reported in the literature.

A closer observation of the Bellman-Ford algorithm and our architectural scheme would reveal that the theoretical maximum level of parallelism in the algorithm has not been fully exploited. This is because all the PEs make use of a single shared bus such that it takes n clock cycles for all the n PEs to broadcast their cost during each iteration. A hypothetical interconnection topology where each PE has a dedicated bus to broadcast data would require just a single clock cycle for all the PEs to broadcast their cost. While such a topology is expensive and infeasible especially for large designs, it is worth exploring other feasible alternatives to the single-bus topology such that multiple PEs can transmit data simultaneously without causing bus contention problems. We believe that the performance of the proposed architecture can be further improved by adopting an interconnection topology that permits parallel data transfer among the PEs.

References

- [1] Cormen, T.H., Leiserson, C.E., and Rivest, R.L.: 'Introduction to Algorithms', (The Massachusetts Institute of Technology, 1990)
- [2] Ahuja, R.K., Mehlhorn, K., Orlin, J.B., and Tarjan, R.E.: 'Faster Algorithms for the Shortest Path Problem', *Journal of the ACM*, 1990, Vol. 37(2), pp. 213-223
- [3] Nilsson, N.J.: 'Problem Solving Methods in Artificial Intelligence' (McGraw Hill, 1971)
- [4] Cherkassky, B., Goldberg, A., and Radzik, T.: 'Shortest Paths Algorithms: Theory and Experimental Evaluation', *Mathematical Programming*, 1996, Vol. 73, pp. 129-174
- [5] Crauser, A., Mehlhorn, K., Meyer, U., and Sanders, P.: 'A Parallelization of Dijkstra's Shortest Path Algorithm', *Proceedings of the 23rd Symposium on Mathematical Foundations of Computer Science*, 1998, Vol. 1450, pp. 722-731
- [6] Meyer, U.: 'Buckets Strike Back: Improved Parallel Shortest-Paths', *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, 2002.

- [7] Driscoll, J.R., Gabow, H.N., Shrairman, R., and Tarjan, R.E.: 'Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation', *Communications of the ACM*, 1998, 31(11), pp. 1343-1354
- [8] Korf, R.E.: 'Planning as Search: A Quantitative Approach', *Artificial Intelligence*, 1987, Vol. 33, pp. 65-88
- [9] Car, A., and Frank, A.U.: 'General Principles of Hierarchical Spatial Reasoning - The Case of Wayfinding', *Proceedings of the 6th International Symposium on Spatial Data Handling*, 1994, Vol. 2, pp. 646-664
- [10] Jagadeesh, G.R., Srikanthan, T., and Quek, K.H.: 'Heuristic Techniques for Accelerating Hierarchical Routing on Road Networks', *IEEE Transactions on Intelligent Transportation Systems*, 2002, Vol.3(4), pp.301-309
- [11] Babb, J., Frank, M., and Agarwal, A.: 'Solving Graph Problems with Dynamic Computation Structures', *Proceedings of SPIE 2914: High-Speed Computing, Digital Signal Processing, and Filtering using Reconfigurable Logic*, 1996, pp.225-236
- [12] Huelsbergen, L.: 'A Representation for Dynamic Graphs in Reconfigurable Hardware and its Application to Fundamental Graph Algorithms', *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, 2000, pp.105-115
- [13] Mencer, O., Huang, Z., and Huelsbergen, L.: 'HAGAR: Efficient Multi-Context Graph Processors', *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, Springer-Verlag, 2002, pp.915-924
- [14] Dandalis, A., Mei, A., and Prasanna, V.K.: 'Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices', *Proceedings of the 6th IEEE Reconfigurable Architectures Workshop (RAW)*, 1999
- [15] Skyttä, J., and Tommiska, M.: 'Dijkstra's Shortest Path Routing Algorithm In Reconfigurable Hardware', *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications (FPL 2001)*, Springer-Verlag, 2001, pp 653-657
- [16] Baker, Z.K., and Gokhale, M.: 'On the Acceleration of Shortest Path Calculations in Transportation Networks', *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2007, pp.23-34

- [17] Xilinx Inc., Virtex-5 Family Overview, 2009
- [18] Shekhar, S., and Kohli, A., and Coyle, M.: 'Path Computation Algorithms for Advanced Traveller Information System (ATIS)', Proceedings of the 9th International Conference on Data Engineering, 1993, pp.31-39
- [19] Hribar, M., Taylor, V., and Boyce, D.: 'Performance Study of Parallel Shortest Path Algorithms: Characteristics of Good Decompositions', Proceedings of the 13th Annual Conference on Intel Supercomputer Users Group Conference (ISUG), 1997

Figure 1: FPGA-based accelerator for shortest path computation

Figure 2: Internal architecture of the processing element

Figure 3: Area growth rate of the SPCU in terms of the number of LUT-FF pairs

Instruction code	Description	Data (On the data bus)
00	Write link.	Host supplies a link to the SPCU in the form of start node index, end node index and weight.
01	Compute shortest paths.	Host supplies the index of the source node.
10	Supply cost and predecessor node.	Host supplies the index of the node whose cost and predecessor node are required.
11	None	Host releases control of the data bus for the use of SPCU.

Table 1: Interaction between the host and the SPCU

SPCU size	No. of LUT-FF pairs used	% of LUT-FF pairs used	No. of FPGA slices occupied	% of FPGA slices occupied
16	2545	4%	974	7%
32	5803	10%	2477	17%
64	11340	19%	3536	24%
128	22534	38%	7896	57%

Table 2: Resource utilization data for different-sized SPCUs on the Virtex-5 SX95T FPGA

SPCU size	Test graph (nodes $\times$ links)	Number of iterations (average)
16	16 $\times$ 48	6
32	32 $\times$ 104	10
64	64 $\times$ 224	14
128	128 $\times$ 466	18

Table 3: Average number of iterations required for algorithm convergence

SPCU size	Clock rate of SPCU (MHz)	Test graph (nodes $\times$ links)	Average number of clock cycles (SPCU)	Execution time of SPCU (μ s)	Execution time of CPU (μ s)	Speed-up factor of SPCU
16	180	16 $\times$ 48	137	0.76	1.65	2.2
32	176	32 $\times$ 104	355	2.02	5.40	2.7
64	162	64 $\times$ 224	954	5.89	14.69	2.5
128	143	128 $\times$ 466	2418	16.91	38.56	2.3

Table 4: Execution times of the SPCU and the CPU

SPCU size	Test graph (nodes $\times$ links)	Number of clock cycles for [14]	Number of clock cycles for SPCU
16	16×48	368	233
32	32×104	1125	563
64	64×224	3354	1402
128	128×466	8924	3350

Table 5: Performance comparison of the SPCU with the solution in [14]