An FPGA Implementation for Solving the Large Single-Source-Shortest-Path Problem

Guoqing Lei, Yong Dou, Rongchun Li, and Fei Xia

***Abstract*—Single source shortest path (SSSP) is a fundamental problem in graph theory. However, the existing SSSP implementations on field-programmable gate arrays (FPGAs) are incapable of processing large graphs by storing the graph and results in internal memories. In this brief, we propose a parallel FPGA implementation to solve the SSSP problem, which is derived from a variant of the "eager" Dijkstra algorithm. In order to process a large graph problem, an extended systolic array priority queue called ExSAPQ is proposed to allow large-scale priority queue processing. The experimental results on the full United States road network show that our SSSP implementation on FPGA can achieve a speedup of 5× over the CPU implementation and the power consumption is only 1/4 of the latter.**

***Index Terms*—Field-programmable gate arrays (FPGAs), single source shortest path (SSSP), systolic array priority queue (SAPQ).**

I. Introduction

SINGLE source shortest path (SSSP) is a fundamental problem in graph theory. It is widely used in network routing, urban transportation, and many other fields. Two classical approaches for the SSSP problem are attributable to Dijkstra [1] and Bellman–Ford [2], [3]. Dijkstra's algorithm is the most efficient sequential implementation in terms of processing speed since it runs in time linear in the number of edges. However, the algorithm requires many iterations and is difficult to parallelize. In contrast, the Bellman–Ford algorithm involves fewer iterations, and each iteration is highly parallelizable. However, the algorithm may process each edge multiple times [4]. Recently, a few types of parallelizable SSSP algorithms have been proposed. Meyer and Sanders proposed the Δ-stepping algorithm [5], a tradeoff between the two extremes of the Dijkstra and Bellman–Ford algorithms. The "eager" Dijkstra and Crauser *et al.* algorithms use different heuristics to determine which vertices should be removed from the priority queue; then, all the vertices are processed in parallel [6], [7].

As graph problems grow in size, efficient parallel shortest path processing becomes important as computational and memory requirements increase. Recently, modern high-capacity field-programmable gate arrays (FPGAs) have been becoming an attractive alternative to accelerate scientific and engineering computing kernels [8]. There has been abundant research to implement SSSP algorithms on FPGA. Most of these works use internal RAMs to store graph and results, and the graph scale is limited by the resources of the FPGA chip. In this brief, we focus on developing an FPGA architecture for the SSSP problem over large-scale graphs. The contributions of this brief can be summarized as follows.

1) We propose a parallel SSSP implementation on FPGA derived from the variant of the "eager" Dijkstra algorithm. In each iteration, a constant number of vertices are removed from the systolic array priority queue (SAPQ) and processed in parallel.
2) An extended SAPQ (ExSAPQ) based on off-chip memory is proposed to accommodate a large-scale SSSP problem. We also propose a scheme to get the off-chip queue elements back to the internal SAPQ to guarantee the correctness of ExSAPQ.
3) Our parallel SSSP architecture was implemented on a Xilinx Virtex-7 XC7VX485T. The experimental results on real road networks show that our architecture can achieve a speedup of 5× over the CPU implementation with lower energy consumption.

The rest of this brief is organized as follows. We review the background and related work in Section II. The proposed parallel SSSP architecture is illustrated in Section III. In Section IV, the run-time complexity is analyzed and compared with related work. The experimental results are presented in Section V. Section VI concludes this brief.

II. Background and Related Work

A. Background

The SSSP problem is to find the shortest paths from a source vertex s to all vertices in a graph. The input consists of a weighted undirected graph $G = (V, E, w)$, where the weight function w assigns an integer weight $w(e) > 0$ to each edge $e \in E$. Let the number of vertices be n and the number of edges be m. The output of the SSSP problem are two arrays, namely $dis[1..n]$ and $pre[1..n]$, where $dis[v]$ denotes the shortest path distance from source to vertex v and $pre[v]$ denotes the predecessor vertex of v in the shortest path to v. The $dis[v]$ is also called the tentative distance from source to vertex v. At any stage of the algorithm, the value $dis[v]$ is guaranteed to be an upper bound on the actual shortest distance $dis^*[v]$. As the algorithm proceeds, the tentative distance $dis[v]$ will decrease until the end when $dis[v]$ equals $dis^*[v]$. The vertex v is said to be settled when the algorithm can guarantee that $dis[v] = dis^*[v]$.

Manuscript received September 4, 2015; revised October 29, 2015; accepted November 21, 2015. Date of publication December 4, 2015; date of current version April 28, 2016. This work was supported by the National Natural Science Foundation of China under Grants 61125201, 61202127, and 61572515. This brief was recommended by Associate Editor Z. Zhang.

G. Lei, Y. Dou, and R. Li are with the National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha 410073, China (e-mail: guoqinglei@nudt.edu.cn).

F. Xia is with the Electronic Engineering College, Naval University of Engineering, Wuhan 430033, China.

Color versions of one or more of the figures in this brief are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSII.2015.2505998

As the algorithm begins, the distance and predecessor of the source vertex are initialized to 0 and -1, respectively. For all other vertices, the distance and predecessor are set to ∞. The priority queue Q is initialized with the source s with $dis[s] = 0$. Dijkstra's algorithm begins by declaring all the vertices to be unsettled and proceeds in multiple iterations. In each iteration, the unsettled vertex u having the minimum tentative distance is removed from the priority queue Q. The vertex u is called the active vertex of this iteration and declared to be settled. Then, for each neighbor v of u given by an edge $e = \langle u, v\rangle$, the operation $Relax(u, v)$ is performed, which is defined as

$$dis(v) \leftarrow \min\left\{dis(v), dis(u) + w\left(\langle u, v\rangle\right)\right\}.$$

Dijkstra's algorithm terminates when there are no more unsettled vertices. If $dis[v]$ is decreased in the $Relax(u, v)$ operation, the priority queue must be modified in either of the following two ways: 1) insert $(v, dis[v])$ to Q if the neighbor $v \notin Q$ and 2) replace the previous $(v, dis^p[v])$ with the latest $(v, dis[v])$ if the neighbor $v \in Q$; this operation is also called decrease key.

An obvious way to parallelize the above Dijkstra algorithm is to remove more vertices from the priority queue in each iteration and relax all the outgoing edges from all these vertices in parallel. The "eager" Dijkstra and Crauser *et al.* algorithms [6] are based on this idea. The "eager" Dijkstra algorithm uses a simple heuristic to determine which vertices should be removed in a given iteration. It used a constant look-ahead factor λ and, in each iteration, the processor removes every vertex u such that $dis(u) \leq \mu + \lambda$, where μ is the global minimum distance of all queued elements. The optimum value for λ depends on the graph density, shape, and weight distribution. For different types of graph data, the "eager" Dijkstra needs to tune the parameter λ. The Crauser *et al.* algorithm uses more precise heuristics to increase the number of vertices removed in each iteration without causing any reinsertions. This algorithm uses two separate criteria in conjunction to determine which vertices should be removed in a given iteration. In our SSSP implementation, we propose a variant that is similar to the "eager" Dijkstra algorithm. We also define a simple constant look-ahead integer value λ, which determines the number of vertices to be removed from the queue in a given iteration.

B. Related Work

A large amount of work has focused on implementing Dijkstra's algorithm for FPGAs. Tommiska and Skyttä [9] store the network structure in the internal ROM block and temporary results in the internal RAM blocks. A comparator bank is used to select the vertex with the smallest distance. They did not implement the priority queue and the graph scale is severely limited by the internal memories. Sun *et al.* [10] design a SAPQ processor to accelerate Dijkstra's algorithm. The SAPQ consists of an array of identical processing elements (PEs) with each PE holding a single queue element. However, the queue size is constrained by the logic resources of FPGA. Their design cannot support a large scale. Lam and Srikanthan [11] adopt a variation of the binary heap called Bucket-Heap for hardware implementation. This design does exploit the parallelism and has not been tested in the real device using a large-scale graph.

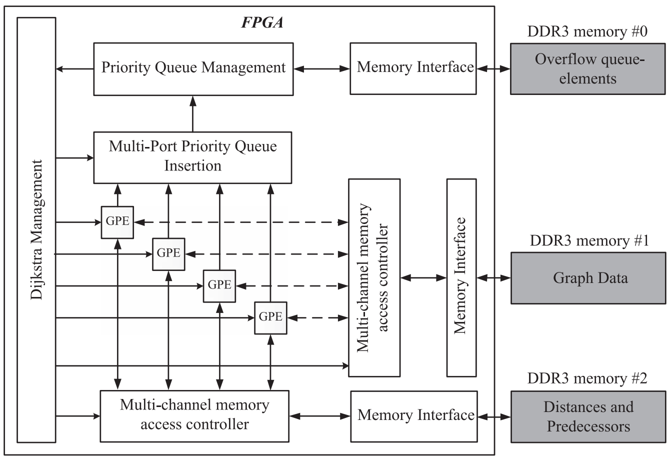
FPGA
Dijkstra Management
Priority Queue Management
Memory Interface
DDR3 memory #0
Overflow queue-elements
Multi-Port Priority Queue Insertion
GPE
GPE
GPE
GPE
Multi-channel memory access controller
Memory Interface
DDR3 memory #1
Graph Data
Multi-channel memory access controller
Memory Interface
DDR3 memory #2
Distances and Predecessors

Fig. 1. Block diagram of our proposed SSSP implementation.

Some research works target the implementations of the Bellman–Ford algorithm [12]–[14]. Dandalis *et al.* [12] propose a pipelined implementation that is composed of a set of PEs. Each PE corresponds to a vertex. The number of vertices of the graph must be smaller than the number of PEs. Jagadeesh *et al.* [13] also use a PE for each vertex. The incoming edges of each vertex are stored in a RAM corresponding to the PE. This design cannot support a large-scale graph either. Recently, Zhou *et al.* [14] store the entire graph in the DRAM for large graph processing. In each clock cycle, p memory words, each corresponding to an edge, are streamed into FPGA from DRAM. The architecture is fully pipelined and the high throughput has been reported for a graph with 2^{20} vertices. However, the external memory bandwidth is assumed to be sufficient to support the maximum data parallelism and the architecture has not been tested using real road networks.

III. Proposed Architecture

For efficiently implementing the SSSP algorithm on FPGAs, which can process a large-scale graph, the graph data and results must be stored in off-chip memory. It is different from the previous work because for most only the internal RAM is used to store the graph data and results. Fig. 1 shows the block diagram of our proposed SSSP implementation. The off-chip memory consists of three independent DDR3 chips, which are used to store the overflow queue elements (DDR3 #0), graph data (DDR3 #1), and output results (DDR3 #2), respectively. Our FPGA-based SSSP implementation is composed of several modules, including the Dijkstra Management (DM), Graph Processing Engine (GPE), Multichannel Memory Access Controller (MAC), Multiport Priority Queue Insertion (PQI), and Priority Queue Management (PQM). The look-ahead integer λ is equal to the number of GPEs in our implementation. In each iteration, the DM module extracts λ vertices from the priority queue. Each GPE is assigned a vertex and relaxes all the outgoing edges of this vertex. MAC is used to multiplex the memory requests from all the GPEs to a single DDR3 memory controller. PQI works in the same way as MAC to multiplex insert operations from the GPEs to a single insert port of the SAPQ module. The PQM module allows an ExSAPQ for large-scale processing.

As for the graph representation, we used the popular Compressed Sparse Row format, which merges the adjacency lists of all vertices into a single O(m)-sized array adj, with the beginning location of each vertex's adjacency list stored in a separate O(n)-sized array $offset$. The weight of each edge is stored in a separate O(m)-sized array w corresponding to the adj array. The distance and predecessor of each vertex v are stored together in an O(n)-sized array $dispre$, of which each element is a tuple $(dis[v], pre[v])$.

The proposed parallel SSSP implementation on FPGA includes three parts running on the different modules (DM module, GPEs, and PQM) (see Fig. 3).

A. DM Design

The DM design consists of three main steps.

1) **Initialization of the GPEs**. The DM initializes the GPEs by removing vertices from the SAPQ. Each GPE is assigned a vertex.
2) **Concurrent computation on GPEs**. Asynchronous execution takes place on each GPE for a given vertex. Once a GPE has relaxed all the outgoing edges of the vertex assigned, it sends a termination signal to the DM, indicating that it is waiting for the next iteration to be started.
3) **Synchronization of the GPEs**. DM waits for all GPEs to finish the relaxing of all outgoing edges of their assigned vertices. The termination of our SSSP implementation depends on the consensus between all GPEs and is reached when there are no vertices removed from the SAPQ.

B. GPE Design

The GPE in our design is responsible for relaxing all the outgoing edges of the assigned vertex, updating the tentative distance and predecessor of the neighbors, inserting neighbors with updated distance to the SAPQ, and writing back the updated tentative distance and predecessor to the off-chip memory. The GPE consists of five stages almost serially. In order to deal with a vertex with a large number of neighbors, q neighbors are loaded from off-chip memory and processed each time. A detailed description of each stage follows.

1) **Read neighbors of** u_i. This stage first reads the $offset$ array of the given vertex u_i to get the start and end position of neighbors in the adj. After this, the neighbors of u_i are retrieved from the off-chip memory #1 and stored in local neighbor memory array Nid.
2) **Read weights of edges**. This stage reads the weights of outgoing edges of the given vertex from memory #1 according to the start and end position in the w array. After that, the weights are stored in the local weight memory array Nw.
3) **Read distances of neighbors**. In this stage, the tentative distances of all neighbors are read from the off-chip memory #2 and stored in the local distance memory array $Ndis$. This stage starts as soon as stage 1 has been finished and can process simultaneously with stage 2.

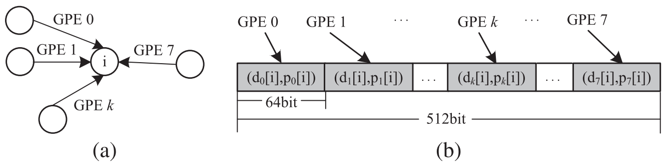

Fig. 2. Duplicate storage schema to avoid the conflict of GPEs.

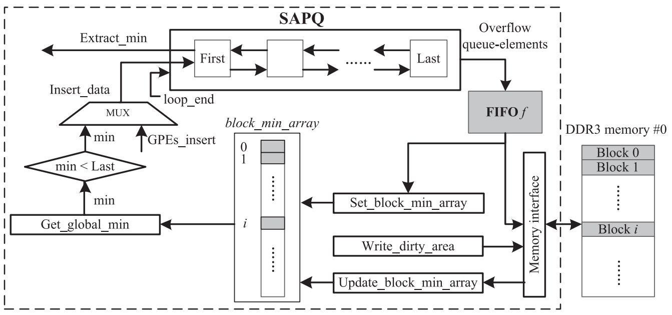

Fig. 3. Dataflow of our ExSAPQ implementation.

A problem of parallel computations on GPEs is that the conflict may happen when several GPEs are relaxing the edges pointing to the same vertex i, as Fig. 2(a) shows. In order to eliminate the conflict, we duplicate the memory space of distance and predecessor of vertex i λ times. In this way, all GPEs write the independent memory spaces to avoid the conflict [see Fig. 2(b)]. When the GPE is relaxing the edge pointing to vertex i, it reads all copies and selects the one with minimal distance. The duplicate storage schema wastes off-chip memory spaces of over $\lambda - 1$ times. Considering that the volume of the DDR3 memory chip today is over several gigabytes and eight 64-b values can be fetched per memory request by setting the data width of the DDR3 memory interface to 512 b, we declare that our schema is feasible to avoid the GPEs' conflict.

4) **Relax all outgoing edges of** u_i. In this stage, the outgoing edges of the given vertex u_i are relaxed in pipeline mode, which leads to an area-efficient and high-performance design. The neighbors with updated distances will be inserted to the priority queue.
5) **Update the distances and predecessors**. In this stage, the vertices with updated distances and predecessors will be written back to off-chip memory #2.

C. ExSAPQ Design

The priority queue is no stranger within the hardware domain and there have been numerous publications on FPGA. In this brief, we believe the SAPQ is a very attractive implementation architecture. It scales well and the complexity of both extract-min and insert operation of SAPQ is O(1) [10]. We choose the SAPQ as the base implementation and extend it to a large scale by using the off-chip memory to store the overflowed queue elements. In our implementation, a First-In-First-Out (FIFO) memory f is used to cache the queue elements that overflowed from the SAPQ. As Fig. 3 shows, the queue elements are stored sequentially in the off-chip memory #0. A constant number of consecutive queue elements are grouped in a block. Let the size of each block be b. For the kth overflowed queue element,

TABLE I
DATA TRANSFERRED IN EACH PHASE OF GPE EXECUTION DURING ONE ITERATION

DRAM#1	DRAM#2
1.1 read $offset$ (512)	-
1.2 read adj (32d)	-
2 read w (32d)	3 read dis (512d)
-	5 write dis/pre (512d)

assume that it is stored in the jth position of block i, which meets $k = i * b + j$ (i, j, k all count from zero). In order to get the global minimal element quickly, we store the queue elements with minimal distance of each block into an on-chip memory array, namely block_min_array, where block_min_array[i] stores the queue element with minimal distance in block i.

In order to get the correct result based on our ExSAPQ, we propose a schema to determine when to get the off-chip queue elements back to the on-chip SAPQ. First, we define a register $\min$ to keep the global minimal queue element of queue elements stored in the off-chip memory. Every time a new iteration starts, the DM module reinserts the $\min$ to SAPQ if $\min$ is lower than the last element of SAPQ. After the $\min$ is reinserted, the storage place of $\min$ in the off-chip memory becomes a dirty area, which will be written with an invalid value. The left queue elements of the block except the $\min$ should be read back to select a new minimal queue element, which is then written to the block_min_array. In each iteration, the signal "loop_end" is inserted to the SAPQ after all the GPEs have finished their computation. The overflowed queue elements in the f are read and sent to a block in the off-chip memory until the "loop_end" signal has been read. The queue elements from the FIFO f are also sent to a comparator to get the minimal element for the block i to which this queue element belongs. After the last queue element of each block i is written to the off-chip memory, the ith element of array block_min_array will be set with the minimal element of block i.

IV. Performance Analysis

For the large graph algorithms, the run time is mainly occupied by the access to the off-chip memories. In our design, three memory chips are accessed independently. We now present a prediction model aimed at computing the total memory access (in bits) and resultant run time (in seconds) required by our implementation.

Let the number of iterations be L, the average degree of the input graph be d, and the effective bandwidth of the off-chip memory be Bw (Gbps). For the road networks with a lower degree, the access to DRAM #0 is trivial while the overflowed queue elements are few in each iteration, so only the memory accesses to DRAM #1 and DRAM #2 are considered. Table I shows the data (in bits shown in brackets) transferred to DRAM #1 and DRAM #2 in each phase of GPE execution.

Although multiple GPEs are processing in parallel, the memory access to the same DRAM are processing sequentially. The total transferred data (in bits) for DRAM #1 and DRAM #2 in each iteration for λ GPEs are $(512+64d) * \lambda$ and $1024d * \lambda$, respectively. For all L iterations, the run time corresponding to DRAM #1 and DRAM #2 is $(512 + 64d) * L\lambda/Bw$ and $1024d * L\lambda/Bw$, respectively. Considering that the access to DRAM #1 and DRAM #2 are almost simultaneous, the total running time is determined by $\max\{(512 + 64d) * L\lambda/Bw, 1024d * L\lambda/Bw\}$, which equals $1024d * L\lambda/Bw$ while the average degree of road networks is small (< 4). It should be noted that the number of iterations L is at most $O(V)$ but may be smaller depending on the look-ahead parameter λ. For the worse case with $L = |V|$, the run time of our implementation is $1024d * |V|\lambda/Bw$, which equals $1024 * |E|\lambda/Bw$. The optimal value for the look-ahead parameter depends on the input graph. The experimental evaluation of λ should be made to choose the optimal one.

TABLE II
SUMMARY OF THE COMPARISON WITH PREVIOUS WORKS

Approach	Graph sizes	Runtime complexity	Bandwidth Speed
[12]	small	$(\tau + 1)m + 2n$	$1\ edges/cc$
[13]	small	$(\tau + 1)(n + 1)$	-
[14]	large	$(\tau + 1)m/p$	$p\ edges/cc$
[9]	small	$nlogn + m$	-
Ours	large	$1024m\lambda/Bw$	$10.6\ GB/s$

We compare our design with previous FPGA-based SSSP implementations. The comparison results are summarized in Table II. The works in [12]–[14] all target another typical SSSP algorithm, the Bellman–Ford algorithm, while the work in [9] targets Dijkstra's algorithm without implementing a priority queue. Of all these works, only that in [14] targets large graphs. With the assumption that p edges can be streamed into FPGA in each clock cycle (cc), the total clock cycle of [14] is $(\tau + 1)m/p$, while τ is the number of required iterations for a given graph. Compared with [14], our design has comparative linear run-time complexity with the number of edges of the graph. However, our design is more practical without the assumptions for the bandwidth speed.

V. Experimental Results

TABLE III
ROAD NETWORKS USED TO TEST THE SSSP ARCHITECTURE

Name	Description	# Nodes	# Arcs
NW	Northwest USA	1,207,945	2,840,208
NE	Northeast USA	1,524,453	3,897,636
CAL	California and Nevada	1,890,815	4,657,742
USA	Full USA	23,947,347	58,333,344

Table III shows the road networks that are from the online repository [15]. For the road network of the entire United States, the graph contains over 23 million vertices. We are not aware of any prior study on the SSSP problem involving graphs of such scale on FPGA. The performance is measured by taking the average execution time of 16 SSSP runs from 16 different source vertices that are randomly chosen. The software implementation is based on a Fibonacci heap from the SPLIB library [16].

For the target Xilinx Virtex-7 XC7VX485T FPGA, we use the Xilinx ISE 14.2 and ModelSim SE 10.1c tools to synthesize and simulate our design, respectively. We have also evaluated the design on the in-house developed FPGA prototyping platform with 24-GB DDR3 memory (three chips each with 8 GB). Our design is determined by several parameters, i.e., the look-ahead parameter λ, the block size b, the size of the

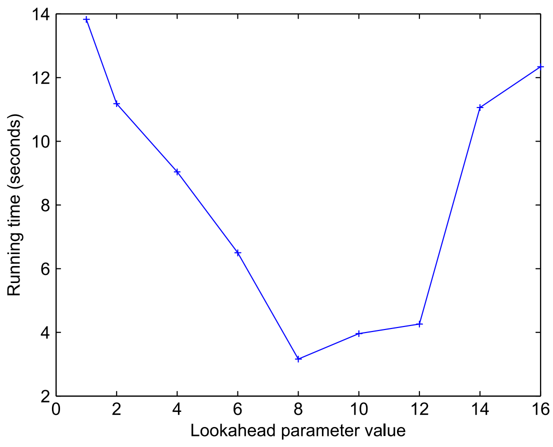

Fig. 4. Effect of look-ahead value on the performance of our implementation using the USA road network.

TABLE IV
SYNTHESIS DATA OF OUR SSSP ARCHITECTURE

Resource	Slice Registers	Slice LUTs	Slices	Block RAMs
Report result	94,323	122,748	38,820	521
Percentage of Total Resource	15%	40%	51%	50%

TABLE V
PERFORMANCE COMPARISON BETWEEN THE SOFTWARE AND OUR FPGA IMPLEMENTATION

	NW		NE		CAL		USA	
	Time	Sp.	Time	Sp.	Time	Sp.	Time	Sp.
AMD[a]	0.55	1	0.81	1	0.89	1	15.8	1
Intel[b]	0.29	1.90	0.41	1.98	0.47	1.89	8.33	1.90
FPGA[c]	0.43	1.28	0.5	1.62	0.62	1.44	3.16	5.00

[a]AMD Opteron 6376 2.3GHz, 48GB memory(12.8 GB/s)
[b]Intel Core i7 4.00GHz, 8GB memory(16 GB/s)
[c]Xilinx XC7VX485t 88MHz, 24GB memory(3 Chips, each with 10.6GB/s)

block_min_array sb, and the size of the internal SAPQ ss. In our experiment, $b = 64$, $sb = 16384$, and $ss = 256$ were chosen. In Fig. 4, we have run our implementation in a range of λ for the United States road network. Limited to the on-chip resources, the implementation with larger λ (over 16) cannot be evaluated. When tying to pick the best value for λ, we select $\lambda = 8$ in our final hardware implementation for road networks. The synthesis result reported by the Xilinx ISE is given in Table IV. Our design utilizes over 50% of Slices and Block RAMs of the XC7VX485T. The maximum frequency of 88 MHz can be achieved. The SAPQ and GPE architecture have occupied most of the resources of the hardware design.

We compare the performance between our FPGA-based design and the software implementation. Table V shows the execution time (in seconds) and speedup of our SSSP architecture over the software implementation on both AMD and Intel platforms. We can see that the Intel implementation can achieve a speedup of 1.89X-1.98X over the AMD implementation for the same SPLIB library. It can be inferred that the speedup is almost attributed to the frequency and bandwidth advantages of Intel over the AMD platform. Compared to the software implementation on the AMD platform, our FPGA implementation can achieve the speedup of 1.28X-5.00X for all four road networks. Compared to the Intel platform, our FPGA implementation can achieve comparative performance for road networks "NE," "NW," and "CAL." However, for the larger road network made up of the entire United States, the FPGA implementation can achieve a visible speedup of 2.6$\times$ over the Intel platform. A power meter is used to measure the power consumption of all these platforms while running the SSSP algorithm. The dissipation power on FPGA (26 W) is only about 1/4 of the general purpose computing platforms.

VI. CONCLUSION

In this brief, we propose an FPGA implementation based on a variant of the "eager" Dijkstra algorithm to process a large-scale SSSP problem. We have extended the SAPQ to support large priority queue processing by storing the overflowed queue elements to the off-chip memory. Experimental results show that our FPGA-based SSSP implementation can achieve a speedup of up to 5$\times$ over the CPU implementation, and the power consumption is only 1/4 of the latter.

REFERENCES

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.
[2] R. Bellman, "On a routing problem," Defense Tech. Inf. Center (DTIC), Fort Belvoir, VA, USA, Tech. Rep. 1956.
[3] L. R. Ford, "Network flow theory," RAND Corp., Santa Monica, CA, USA, 1956.
[4] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal, "Scalable single source shortest path algorithms for massively parallel systems," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 889–901.
[5] U. Meyer and P. Sanders, "Δ-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, no. 1, pp. 114–152, Oct. 2003.
[6] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine, "Single-source shortest paths with the parallel boost graph library," in *Proc. 9th DIMACS Implementation Challenge: Shortest Path Problem*, Piscataway, NJ, USA, 2006, pp. 219–248.
[7] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, "A parallelization of Dijkstra's shortest path algorithm," in *Mathematical Foundations of Computer Science*. Berlin, Germany: Springer-Verlag, 1998, pp. 722–731.
[8] G. Wu, X. Xie, Y. Dou, and M. Wang, "High-performance architecture for the conjugate gradient solver on FPGAs," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 60, no. 11, pp. 791–795, Nov. 2013.
[9] M. Tommiska and J. Skyttä, "Dijkstra's shortest path routing algorithm in reconfigurable hardware," in *Field-Programmable Logic and Applications*. Berlin, Germany: Springer-Verlag, 2001, pp. 653–657.
[10] H. Sun, C. Eik Wee, N. Shaikh-Husin, and M. K. Hani, "Accelerating graph algorithms with priority queue processor," in *Proc. Regional Postgraduate Conf. Eng. Sci.*, 2006, pp. 257–262.
[11] S.-K. Lam and T. Srikanthan, "Accelerating shortest path computations in hardware," in *Proc. IEEE CASE*, 2010, pp. 63–68.
[12] A. Dandalis, A. Mei, and V. K. Prasanna, "Domain specific mapping for solving graph problems on reconfigurable devices," in *Parallel and Distributed Processing*. Berlin, Germany: Springer-Verlag, 1999, pp. 652–660.
[13] G. R. Jagadeesh, T. Srikanthan, and C. Lim, "Field programmable gate array-based acceleration of shortest-path computation," *IET Comput. Digit. Tech.*, vol. 5, no. 4, pp. 231–237, Jul. 2011.
[14] S. Zhou, C. Chelmis, and V. K. Prasanna, "Accelerating large-scale single-source shortest path on FPGA," in *Proc. IEEE IPDPSW*, 2015, pp. 129–136.
[15] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, vol. 74. Providence, RI, USA: Amer. Math. Soc., 2009.
[16] A. Goldberg, "Andrew Goldberg's Network Optimization Library." [Online]. Available: http://www.avglab.com/andrew/soft.html