High-Performance Architecture for the Conjugate Gradient Solver on FPGAs

Guiming Wu, Xianghui Xie, *Member, IEEE*, Yong Dou, *Member, IEEE*, and Miao Wang

***Abstract*—The conjugate gradient (CG) solver is an important algorithm for solving the symmetric positive define systems. However, existing CG architectures on field-programmable gate arrays (FPGAs) either need aggressive zero padding or can only be applied for small matrices and particular matrix sparsity patterns. This brief proposes a high-performance architecture for the CG solver on FPGAs, which can handle sparse linear systems with arbitrary size and sparsity pattern. Furthermore, it does not need aggressive zero padding. Our CG architecture mainly consists of a high-throughput sparse matrix-vector multiplication design including a multi-output adder tree, a reduction circuit, and a sum sequencer. Our experimental results demonstrate that our CG architecture can achieve speedup of 4.62X–9.24X on a Virtex5-330 FPGA, relative to a software implementation.**

***Index Terms*—Conjugate gradient (CG) solver, field-programmable gate array (FPGA), sparse matrix-vector multiplication (SpMV).**

I. Introduction

LINEAR-SYSTEM solvers are widely used in scientific and engineering computing. There are two categories of methods for solving linear systems: direct method, where the solution is computed through lower/upper triangular decomposition and solving triangular system, and iterative method, where the solution is approximated by performing iterations from an initial vector. Direct method is only feasibly applied for small systems. Iterative method can be used to solve larger systems. The conjugate gradient (CG) method, which is dominated by sparse matrix-vector multiplication (SpMV), is one of the most important iterative methods used to solve large sparse linear systems.

Modern high-capacity field-programmable gate arrays (FPGAs) are an attractive alternative to accelerate scientific and engineering applications [1]. There has been abundant research to implement SpMV and CG on FPGAs. Some of the existing designs are aimed at implementing CG for dense linear systems [2], [3]. Some have focused on sparse linear systems from particular applications such as finite element method [4]. There has been little work in the area of solving large and general sparse linear systems. Most of the FPGA-based solutions proposed for sparse systems assume that the input sparse matrix is stored in the internal RAM (i.e., BRAM). The size of the problem that can be solved is limited by the available internal RAM [5]. To solve these problems, in this brief, we propose a high-performance architecture for the CG solver on FPGAs, which can handle sparse linear systems with arbitrary size and sparsity pattern. The contributions of this brief can be summarized as follows.

1) A data blocking method is proposed to partition large sparse matrices into square blocks. SpMV is decomposed into fine-grained computational tasks that operate these square blocks and can be directly implemented on FPGAs.
2) We present a pipelined and high-throughput SpMV architecture without the need of zero padding for each row in matrices. In our CG design, the SpMV architecture can perform a dot product in addition to SpMV.
3) Our proposed CG architecture was implemented on a Xilinx Virtex-5 XC5VLX330 device. The experimental results show that our architecture can achieve speedup of 4.62X–9.24X, compared to a software implementation.

The rest of this brief is organized as follows. We review the background and related work in Section II. The proposed CG architecture is illustrated in Section III. The experimental results are presented in Section IV. Section V concludes this brief.

Manuscript received March 31, 2013; revised May 28, 2013; accepted July 21, 2013. Date of publication August 29, 2013; date of current version November 14, 2013. This work was supported in part by China Postdoctoral Science Foundation, by the Hi-Tech Research and Development Program of China under Grant 2013AA010105, and by the Natural Science Foundation of China under Grant 61125201. This brief was recommended by Associate Editor P. Li.

G. Wu, X. Xie, and M. Wang are with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214125, China (e-mail: wu.guiming@meac-skl.cn).

Y. Dou is with the National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha 410073, China (e-mail: yongdou@nudt.edu.cn).

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.2013.2278111

1: $r_0 = b - Ax_0$
2: $p_0 = r_0$
3: **for** $i = 1, 2, 3,..$ **do**
4: $q_{i-1} = Ap_{i-1}$
5: $\rho_{i-1} = r_{i-1}^T r_{i-1}$
6: $\alpha = \rho_{i-1} / p_{i-1}^T q_{i-1}$
7: $x_i = x_{i-1} + \alpha p_{i-1}$ /* new solution*/
8: $r_i = r_{i-1} - \alpha q_{i-1}$ /* new residual*/
9: $\beta = r_i^T r_i / \rho_{i-1}$
10: $p_i = r_i + \beta p_{i-1}$ /* new direction*/
11: check convergence and exit if done
12: **end for**

Fig. 1. CG solver.

II. Background and Related Work

A. Background

The CG solver is an important algorithm for solving the symmetric positive define systems $Ax = b$. If the coefficient matrix A is not ill-conditioned, the CG solver can converge rapidly. The algorithm of the CG method is shown in Fig. 1.

The algorithm consists of two phases. It initializes the residual r_0 with an initial vector x_0, followed by iterations which terminate when the residual r_i is sufficiently small. In the algorithm, ρ, α, and β are scalars, and x, r, p, and q are vectors. One iteration includes one SpMV (Ap_{i-1}), two dot products ($r_i^T r_i$ and $p_{i-1}^T q_{i-1}$), three vector updating operations (for new solution, residual, and direction), and two scalar divisions. The rate of convergence for the CG solver depends on the condition number of A. It decreases as the condition number increases.

B. Related Work

A large amount of work has focused on implementing SpMV for FPGAs. The execution time of SpMV dominates the CG solver. A variety of FPGA-based accelerators can be found in the recent research literature [6]–[8]. The architecture proposed by Zhuo and Prasanna [6] consists of a binary adder tree, the output of which is fed into a reduction circuit that accumulates partial dot products. Their results show that their design can achieve over 350 million floating-point operations per second. However, the overhead of zero padding limits the performance. The design in [7] consists of multiple processing elements (PEs), with each PE being assigned a data block. In their design, the sparse matrix is partitioned into stripes along the rows, and then the stripes are divided into blocks. The partial results from all PEs are sent to a unique reduction circuit, which would be the performance bottleneck. This design has the same overhead of zero padding as that in [6]. In [5], Sun *et al.* propose a scalable and efficient SpMV architecture that can handle arbitrary matrix sparsity patterns without zero padding. However, their design needs more hardware overhead such as distributed memories of FPGA, which are used to keep the map table generated offline. Moreover, their design cannot handle large matrices.

Some research has been conducted on implementing the complete CG method on FPGAs. Roldao and Constantinides [2] propose a deeply pipelined FPGA-based CG implementation, which targets at accelerating multiple small-to-medium-sized dense systems of linear equations. Their design cannot solve sparse systems that are widely used in scientific applications. In [9], Roldao *et al.* present a CG implementation only for band structured linear equations. This design can take advantage of the banded structure. It cannot be applied for general sparse systems. In [3], Roldao *et al.* exploit the tradeoff between precision and iteration count in the dense CG solver. A speedup of 26X on average can be achieved through exploring this tradeoff. Their technique was applied to model predictive control. Strzodka and Göddeke [10] propose pipelined mixed precision algorithms on FPGAs for CG in the partial differential equation solvers. Bakos and Nagar [11] exploit sparse matrix symmetry to nearly double the ratio of computation to communication. Other designs can be found in the research literature [12], [13].

III. Proposed Architecture

For efficiently implementing the CG solver on FPGAs, which can handle symmetric positive define systems of arbitrary size, we must partition sparse matrices to fit in the limited internal RAM. In this brief, we assume that sparse matrix and vectors in the SpMV $y = Ax$ are stored in off-chip memory. It is different from some previous work, where only the internal RAM is used to store the matrix and vectors. In our design, a block of x is transferred to the internal RAM before computing, and then a block of A is streamed into our architecture to drive the computation. After all the blocks from the same row are finished, a block of y will be obtained and stored back to the off-chip memory.

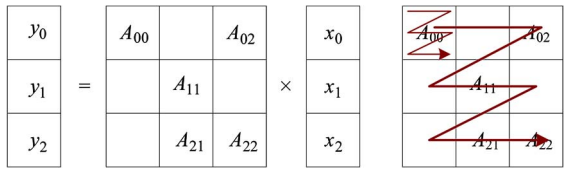

Fig. 2. Proposed sparse matrix blocking method.

A. Data Blocking

We propose a data blocking method, which is referred to as 2-D uniform blocking, to partition sparse matrices. As shown in Fig. 2, the sparse matrix A is divided into square blocks, where the block sizes along the row and the column are equal. The vectors x and y have the same block size as the sparse matrix. We take advantage of 2-D uniform blocking in two ways. First, it can enable storing sparse matrix implicitly by using fewer block information, compared with other methods [6], [7]. Second, blocked x and y can be stored in the internal RAM, and data reuse can be exploited.

After blocking, each block of the sparse matrix is represented by two arrays: val for the nonzeros in row-major order and col_ind for the column indices of the nonzeros. All blocks are stored in the off-chip memory in row-major order.

B. SpMV Architecture

Our SpMV architecture mainly consists of a front end and a back end.

1) Front End: There is one output in a simple k-input binary adder tree. It can only compute the product summation of one row in the matrix A with the vector x in one clock cycle. If the number of nonzeros from one row is not a multiple of k, there will be zero padding, which can cause useless computation. To avoid useless computation, we propose a high-throughput multi-output adder tree, which can handle the nonzero elements from different rows in one clock cycle.

Fig. 3 presents the architecture of the front end with the proposed multi-output adder tree. The front end is composed of k RAMs, k multipliers, and a configurable multi-output adder tree. The parameters of our design include k, the number of the multipliers, and b, the data block size. The value of k is determined by the off-chip memory bandwidth, and can be calculated as

$$k \approx \frac{\text{memory bandwidth}}{\text{bit size of val and col\_ind} \times \text{clock frequency}}. \quad (1)$$

The block size b represents the depth of each x RAM. In each clock cycle, k nonzero elements, which may come from the same or different rows in one block, are multiplied with the elements in the vector x. The outputs from the multipliers are

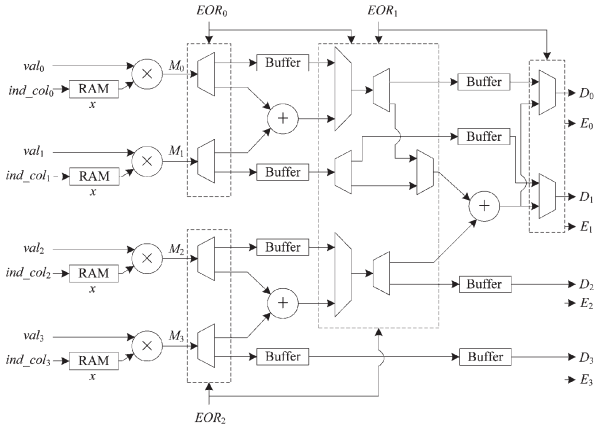

Fig. 3. Front-end architecture with proposed multi-output adder tree ($k = 4$).

TABLE I
RULE OF ADDER OUTPUT

$EOR_0EOR_1EOR_2$	D_0	D_1	D_2	D_3
0 0 0	$M_0+M_1+M_2+M_3$	-	-	-
0 0 1	$M_0+M_1+M_2$	M_3	-	-
0 1 0	M_0+M_1	M_2+M_3	-	-
0 1 1	M_0+M_1	M_2	M_3	-
1 0 0	M_0	$M_1+M_2+M_3$	-	-
1 0 1	M_0	M_1+M_2	M_3	-
1 1 0	M_0	M_1	M_2+M_3	-
1 1 1	M_0	M_1	M_2	M_3

then sent to the adder tree. If the number of nonzeros from one block is not a multiple of k, zero padding will happen. In the simple adder tree, zero padding may happen for each row of a block. Compared to the simple adder tree, the complexity of zero padding in our proposed design is reduced from $O(m^2/b)$ to $O((m/b)^2)$, where m is the size of the matrix.

The input data are streamed into the multipliers in the order shown on the right side of Fig. 2. One triple $\{$val, col_ind and $EOR\}$ corresponds to one nonzero element. In the adder tree shown in Fig. 3, there are four EOR signals, each indicating the last nonzero element of one row. These EOR signals are similar to the input pattern vector in [5]. Vector x has been already stored in the RAMs. col_ind is used as the address to read the corresponding element of x. The multiplier multiplies the nonzero element from the sparse matrix with the corresponding element from x. Ms (M_0, M_1, M_2, and M_3) are the results of the front-end multipliers. The products from the same row should be summed up by the adder tree. If two products are not from the same row, they cannot be summed up. The TRUE valued EOR signals are the boundaries of the products that need to be added together. These EOR signals are used to reconfigure the connections between the adders in the adder tree. As shown in Fig. 3, EOR_0, EOR_1, and EOR_2 control several MUXs to generate the partial sums according to the rule shown in Table I. Ds (D_0, D_1, D_2, and D_3) are four sums or partial sums for four different rows. In Table I, "–" means invalidation of the corresponding Ds. When $EOR_0EOR_1EOR_2 = 000$, the four input nonzero elements come from the same row. The four Ms will be added up to generate D_0, which is the only valid partial sum.

In Fig. 3, Es (E_0, E_1, E_2, and E_3) are Ds' flags indicating the last partial sum of one row. Assuming that the sums among

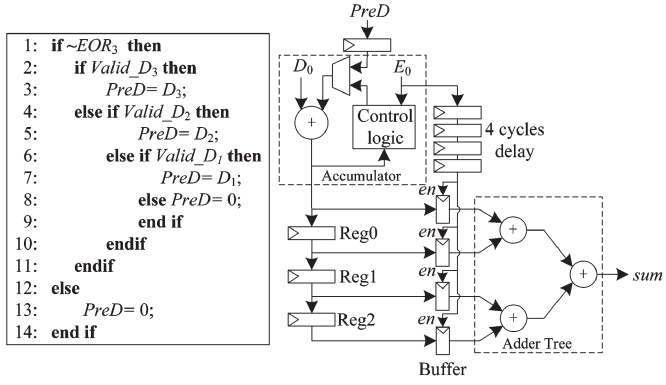

Fig. 4. Reduction circuit.

Ds are from NR rows, then NR can be obtained by arithmetically adding the four EORs

$$NR = EOR_0 + EOR_1 + EOR_2 + EOR_3. \tag{2}$$

$E_i (0 \le i \le 3)$ can be obtained as follows:

$$E_i = \begin{cases} 1, & NR > i \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

2) Back End: When $EOR_3 = 0$, the last valid partial sum $PreD$ has to be accumulated with the first partial sum D_0 of the next clock cycle. The generation of $PreD$ is described by the algorithm on the left side of Fig. 4. In the algorithm, Valid_Ds are the validation signals of the Ds. For "–" in Table I, the corresponding Valid_D signal is FALSE. When only D_0 is valid, $PreD = 0$, and the current D_0 may be accumulated with the next D_0 from the same row. The reduction circuit is shown on the right side of Fig. 4. It is responsible for accumulating the last partial sum ($PreD$) with the first partial sum (D_0) of the next clock cycle, and summing up all partial sums (D_0) from the same row.

In Fig. 4, the floating-point adder in the accumulator has a four-stage pipeline. We assume that there are $n (n \ge 4)$ partial sums from the same row. After n clock cycles, four intermediate results are obtained and distributed over the different pipeline stages of the adder. The four intermediate results need to be added up to compute the final result. In the reduction circuit, three shift registers are used to keep the first three intermediate results when they are dropped out of the adder. Then, the four intermediate results are added together by a back-end adder tree. We utilize this back-end adder tree to enable a deeply pipelined and high-throughput architecture. The back-end adder tree has four inputs. When there are less than four partial sums from the same row, we must forbid these partial sums from adding with the partial sums from the previous row. To implement this, the three shift registers should be reset after their values are latched to the buffers.

In addition to the reduction circuit, the back end in our SpMV architecture also includes a sum sequencer. Our architecture may generate more than one sums for different rows at one time. If the four input nonzero elements are from four different rows, there will be four sums generated at the same clock cycle. Due to blocking, these sums need to be accumulated with the sums from the previous data block of the same rows. For example, in Fig. 2, the sums from block A_{02} should be accumulated with

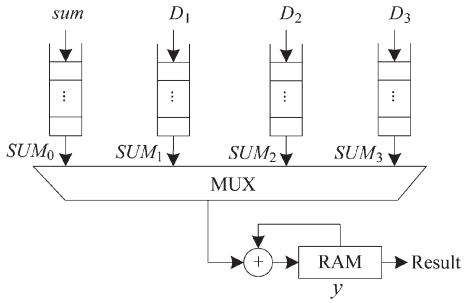

Fig. 5. Sum sequencer.

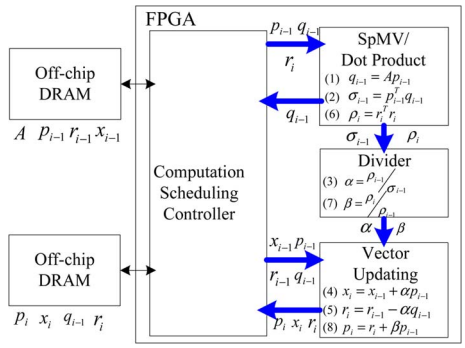

Fig. 6. Block diagram of FPGA-based CG architecture.

the sums from block A_{00}, since they come from the same rows. The previous sums are stored in the internal RAM. The internal RAM in FPGAs has up to two read/write ports. At most two sums can be added up with the according sums stored in the RAM at one time. For the sake of simplicity, we assume that the RAM has only one read/write port. Thus, a sum sequencing scheme is required. Fig. 5 shows the architecture of the sum sequencer. The final four sums are sum, D_1, D_2, and D_3. If they are valid and their Es are TRUE, they will be pushed into the FIFOs. A MUX is used to select one sum from the four FIFOs. The selected sum is sent to the accumulator that accumulates it with the sum stored in the y RAM.

C. Proposed CG Architecture

1) Architecture Overview: In addition to the SpMV architecture, we also need a dot-product unit, a vector updating unit, and a floating-point divider to finish other operations of CG. A computation scheduling controller is utilized to schedule the data between the off-chip memory and the computation modules according to the CG algorithm. It can control the operation sequence of all modules. Our proposed FPGA-based architecture is shown in Fig. 6. We consider the operations in the loop of the CG algorithm shown in Fig. 1. When $i > 1$, $\rho_{i-1}(r_{i-1}^T r_{i-1})$ in the current iteration is $\rho_i(r_i^T r_i)$ from the previous iteration. We only need to compute ρ_i. Fig. 6 gives the order of all operations in one iteration ($i > 1$) and the data flow within the modules. In the design, the input sparse matrix and the input/output vectors are stored in the off-chip DRAMs since the FPGA does not have enough local memory to hold an entire large matrix or vectors.

2) Composed SpMV and Dot-Product Architecture: The dot product has similar computing pattern as SpMV. Therefore, they can share large part of data path, reducing hardware overhead greatly. Dot-product unit has its own finite state machine. The data path of the SpMV architecture can handle dot-product operations through extending the multi-output adder tree. Before operating the dot product, the inputs of the front-end multipliers are switched to the two vectors that are ready for dot product. Initially, EOR_0, EOR_1, EOR_2, and EOR_3 are reset to "0". EOR_3 will be set to "1" when the last part of the dot product is performed. The dot product can be viewed as one row operation in SpMV. The sum in Fig. 4 will be the result of one dot product operation.

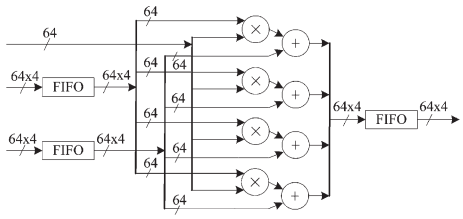

Fig. 7. Vector updating unit.

TABLE II
SYNTHESIS DATA OF OUR CG ARCHITECTURE

Resource	Slice Registers	Slice LUTs	Slice	DSP48Es	Block RAMs
Report Result	30,200	38,261	16,304	64	117
Percentage of Total Resource	14%	18%	31%	33%	40%

3) Vector Updating Unit: Vector updating unit is responsible for computing the addition/subtraction between a vector and the product of a vector and a scalar. Vector updating unit does not share floating-point multipliers and floating-point adders with the SpMV architecture for reducing the control complexity. As shown in Fig. 7, the vector updating unit is a vectorized architecture, where the two input vectors are buffered in the input FIFOs. When both are ready, the data will be sent into the vector computation unit. The output vector is buffered in the output buffer. The depth of the two FIFOs is the block size b, their width is $64 \times k$ for the 64-bit double precision floating-point arithmetic.

4) Divider: Our floating-point divider is based on a radix-4 SRT divider algorithm [14]. There are only two division operations in each iteration. Pipelined divider is not necessary. We implemented a low-cost serial divider.

IV. EXPERIMENTAL RESULTS

Our target FPGA is Xilinx Virtex-5 XC5VLX330 device. We used the Xilinx ISE 10.1 and ModelSim 6.2h tools to synthesize and simulate, respectively, our design. Our CG design is determined by several parameters, i.e., the block size b, the number of the front-end multipliers k, and the pipeline depth of floating-point adder h. Totally, our design needs $2k + h$ adders, $2k$ multipliers, one divider, and $64 \times b(k + 1)$-bit internal RAM. In our experiment, $b = 8192$, $k = 4$, and $h = 8$ were chosen. The synthesis result reported by the Xilinx ISE is given in Table II. Our design only utilizes 31% of slices of the XC5VLX330 device. The maximum frequency of 169 MHz can be achieved. The SpMV architecture has occupied more than half resource of the CG design. Table III shows our benchmark matrices that are from the University of Florida Sparse Matrix Collection [15].

TABLE III
BENCHMARK MATRICES

No.	Name	Dimension	Nonzero	Application
1	ted_B	10,605	144,579	thermal problem
2	bundle1	10,581	770,811	computer graphics problem
3	qa8fm	66,127	1,660,579	acoustics problem
4	raefsky4	19,779	1,316,789	structural problem
5	bcsstk25	15,439	252,241	structural problem
6	bcsstm25	15,439	15,439	structural problem

TABLE IV
CYCLE COUNTS OF SpMV FOR DIFFERENT k

Matrix	1	2	3	4	5	6
$k=2$	90,891	401,789	961,817	699,304	149,934	31,263
$k=4$	47,768	201,787	482,972	350,369	75,283	23,562
$k=8$	26,422	102,457	244,140	176,332	38,218	19,723

TABLE V
COMPUTATION TIME OF SpMV AND ONE ITERATION OF CG (ms)

Matrix	1	2	3	4	5	6
SpMV time	0.318	1.345	3.219	2.335	0.501	0.157
CG time	0.481	1.508	4.226	2.638	0.764	0.392
Percentage	66 %	89%	76%	89%	65%	40%

TABLE VI
PERFORMANCE COMPARISON (IN SECONDS)

Matrix	10^{-4}			10^{-6}		
	Hardware	Software	Speedup	Hardware	Software	Speedup
1	0.0019	0.0146	7.68	0.0178	0.1023	5.74
2	0.1237	1.0505	8.49	0.1931	1.6199	8.38
3	0.0797	0.6981	8.75	0.1563	1.3204	8.44
4	0.0395	0.3652	9.24	0.0554	0.4945	8.92
5	0.1842	1.3722	7.44	1.3761	9.9577	7.23
6	0.0624	0.2896	4.64	0.7288	3.3734	4.62

They are symmetric positive define matrices and from different applications.

Table IV shows the clock-cycle counts for one SpMV on our SpMV architecture when $k = 2, 4,$ and 8. Note that the performance improvement is approximated linearly with increasing k except the last matrix (bcsstm25). There is only one nonzero in each row of this matrix. In this extreme case, the sum sequencer in our proposed SpMV architecture may limit the performance improvement.

Table V gives the performance results and the percentage of SpMV computation time relative to the total time of CG in our CG design for $k = 4$. Our hardware design was clocked at 150 MHz. The percentage can be obtained after running one iteration of CG. Notice that the SpMV computation time still dominates that of CG.

Now, we compare the performance between our FPGA-based design and a software implementation. In our experiment, the CG software implementation is preconditioned conjugate gradient (PCG) routine in Matlab running on an Intel Q6600 CPU (2.4 GHz, with 8 MB of L2 cache). PCG routine runs the CG solver when no preconditioner was set. The PCG routine in Matlab is highly optimized. In our experiment, we set the vector $b = Ac$, where all elements of the vector c are "1", for any benchmark matrix A. Table VI shows the speedup of our CG architecture over the software implementation in different requested tolerances. If the relative residual $\text{norm}(b - Ax)/\text{norm}(b)$ is less than the requested tolerance, it means that the CG solver converges. Compared to the software implementation, our CG architecture can achieve the speedup of 4.64X–9.24X for the tolerance of 10^{-4}, and 4.62X–8.92X for the tolerance of 10^{-6}. It is interesting to note that the speedup will be decreased if degrading the tolerance for the same matrix. A higher precision can be obtained through a lower tolerance, but more iterations are required. For cache-based processor, data reuse can be improved when performing more iterations, reducing the cache misses. As a result, latter iterations may run faster than the previous iterations. However, for our hardware design, each iteration has the same execution time.

V. CONCLUSION

In this brief, we have proposed a high-performance FPGA-based implementation for the CG solver. It mainly consists of a high-throughput SpMV architecture. Our architecture can handle sparse linear systems with arbitrary size and sparsity pattern. Our architecture does not need aggressive zero padding or preprocessing. We implemented the architecture on a Virtex5-330 FPGA. The experimental results have demonstrated that our architecture can achieve speedup of 4.62X–9.24X over a software implementation. The performance of our architecture can be improved for larger k.

REFERENCES

[1] O. Storaasli and D. Strenski, "Exploring accelerating science applications with FPGAs," in *Proc. Reconfigurable Syst. Summer Inst.*, 2007, pp. 1–30.
[2] A. Roldao and G. A. Constantinides, "A high throughput FPGA-based floating point conjugate gradient implementation for dense matrices," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 1, pp. 75–86, Jan. 2010, article 1.
[3] A. Roldao, A. Shahzad, G. A. Constantinides, and E. C. Kerrigan, "More FLOPS or more precision? Accuracy parameterizable linear equation solvers for model predictive control," in *Proc. IEEE FCCM*, 2009, pp. 209–216.
[4] J. Hu, S. F. Quigley, and A. Chan, "An element-by-element preconditioned conjugate gradient solver of 3D tetrahedral finite elements on an FPGA coprocessor," in *Proc. FPL*, 2008, pp. 575–578.
[5] S. Sun, M. Monga, P. H. Jones, and J. Zambreno, "An I/O bandwidth-sensitive sparse matrix-vector multiplication engine on FPGAs," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 59, no. 1, pp. 113–123, 2012.
[6] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proc. ACM/SIGDA FPGA*, 2005, pp. 63–74.
[7] J. Sun, G. D. Peterson, and O. O. Storaasli, "Sparse matrix-vector multiplication design on FPGAs," in *Proc. IEEE FCCM*, Apr. 2007, pp. 349–352.
[8] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal FPGA matrix-vector multiplication architecture," in *Proc. IEEE FCCM*, 2012, pp. 9–16.
[9] A. Roldao, G. A. Constantinides, and E. C. Kerrigan, "A floating-point solver for band structured linear equations," in *Proc. IEEE FPT*, 2008, pp. 353–356.
[10] R. Strzodka and D. Göddeke, "Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components," in *Proc. IEEE FCCM*, Apr. 2006, pp. 259–268.
[11] J. D. Bakos and K. K. Nagar, "Exploiting matrix symmetry to improve FPGA-accelerated conjugate gradient," in *Proc. IEEE FCCM*, 2009, pp. 223–226.
[12] D. Dubois, A. Dubois, T. Boorman, C. Connor, and S. Poole, "Sparse matrix-vector multiplication on a reconfigurable supercomputer with application," *ACM Trans. Reconfigurable Tech. and Syst.*, vol. 3, no. 1, pp. 1–31, Jan. 2010, article 2.
[13] O. Maslennikow, V. Lepekha, and A. Sergyienko, "FPGA implementation of the conjugate gradient method," in *Proc. PPAM*, vol. 3911, *LNCS*, 2006, pp. 526–533.
[14] K. Keshab and R. Hosahalli, "A fast radix-4 division algorithm and its Architecture," *IEEE Trans. Comput.*, vol. 44, no. 6, pp. 826–831, Jun. 1995.
[15] T. Davis and Y. Hu, The University of Florida Sparse Matrix Collection. [Online]. Available: www.cise.ufl.edu/research/sparse/matrices