ELectronics EXpress

LETTER

IEICE Electronics Express, Vol.12, No.11, 1–10

A deeply-pipelined FPGA-based SpMV accelerator with a hardware-friendly storage scheme

Song Guo[1a)], **Yong Dou**[1], **Yuanwu Lei**[1], **and Guiming Wu**[2]

[1] *National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha, China*

[2] *State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China*

a) *songguo@nudt.edu.cn*

Abstract: This paper presents a high performance sparse matrix-vector multiplication (SpMV) accelerator on the field-programming gate array (FPGA). By exploiting a hardware-friendly storage scheme, named as Variable-Bit-Width Coordinate Block Quasi Compressed Sparse Row, the redundant computation and memory accesses can be reduced greatly through the nested block compression and variable-bit-width column-index encoding schemes. Based on the proposed compression scheme, a deeply-pipelined SpMV accelerator is implemented on a Xilinx Virtex XC7VX485T FPGA platform, which can handle sparse matrices with arbitrary size and sparsity pattern. Experimental results show that the proposed design can gain higher performance for most of the tested matrices and improve the utilization of the memory bandwidth up to 13×, compared with the previous works on the Convey platforms (HC-1 and HC-2ex) and Nvidia Tesla S1070 GPU platform.

Keywords: SpMV, FPGA, hardware-friendly compression scheme, deeply-pipelined accelerator

Classification: Electron devices, circuits, and systems

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams and K. A. Yelick: Technical Report UCB/EECS-2006-183 (2006) 9.

[2] R. W. Vuduc and H. J. Moon: HPCC (2005) 807. DOI:10.1007/11557654_91

[3] Y. Saad: Technical Report (2005) 1.

[4] J. Sun, G. D. Peterson and O. O. Storaasli: FCCM (2007) 349. DOI:10.1109/FCCM.2007.56

[5] G. Wu, X. Xie, Y. Dou and M. Wang: IEEE Trans. Circuits Syst. II, Exp. Briefs **60** (2013) 791. DOI:10.1109/TCSII.2013.2278111

[6] D. Zou, Y. Dou, S. Guo and S. Ni: IEICE Electron. Express **10** (2013) 20130529. DOI:10.1587/elex.10.20130529

[7] S. Kestur, J. D. Davis and E. S. Chung: FCCM (2012) 9. DOI:10.1109/FCCM.

DOI: 10.1587/elex.12.20150161
Received February 9, 2015
Accepted April 6, 2015
Publicized May 1, 2015
Copyedited June 10, 2015

2012.12
[8] S. Jain, R. Pottathuparambil and R. Sass: HPC (2011) 80.
[9] S. Jain-Mendon and R. Sass: ReConFig (2012) 1. DOI:10.1109/ReConFig.2012.6416788
[10] L. Zhuo and V. K. Prasanna: FPGA (2005) 63. DOI:10.1145/1046192.1046202
[11] S. Sun, M. Monga, P. H. Jones and J. Zambreno: IEEE Trans. Circuits Syst. **59** (2012) 113. DOI:10.1109/TCSI.2011.2161389
[12] K. K. Nagar and J. D. Bakos: FCCM (2011) 1. DOI:10.1109/FCCM.2011.60
[13] R. Halstead and W. Najjar: Technical Report UCR-CSE-2013-02011 (2013) 1.
[14] T. A. Davis and Y. Hu: ACM TOMS **38** [1] (2011) 1. DOI:10.1145/2049662.2049663

1 Introduction

Sparse matrix-vector multiplication (SpMV) is one of the most essential kernels in scientific computing, such as sparse linear solvers, image processing, circuit analysis, and so on. However, as one of the "seven dwarfs" [1], SpMV is notorious for sustaining low fractions (less than 10% [2]) of the peak performance on the general purpose processors, mostly due to the inefficient use of the memory bandwidth. It results from the mismatches between the memory access patterns and the compression schemes of the sparse matrix.

In recent years, FPGA has become an attractive platform to accelerate SpMV. The increased on-chip memory capacity can lessen the requirement of the memory bandwidth. The customizable feature of FPGAs can be used to design the application-specific memory structures and processing elements to match the compression schemes, which can increase the utilization of memory bandwidth. Both factors are essential to improve the performance of SpMV due to the memory-bound characteristic of SpMV.

There has been a substantial body of works to implement SpMV on FPGAs. However, lots of the prior works impose many constraints, such as excessive zero-paddings, word-level index data, and pipeline stall, which degrade the utilization of the memory bandwidth and degrade the performance of SpMV.

In this paper, a new hardware-friendly compression scheme is proposed to exploit the bit capacity of FPGA. Based on this scheme, a deeply-pipelined SpMV accelerator on an Xilinx Virtex XC7VX485T FPGA platform. The main contributions can be summarized as follows, (1) A hardware-friendly compression scheme, named as variable-bit-width coordinate block quasi compressed sparse row (VBW-CBQCSR), is proposed to reduce the redundant computations and memory accesses by exploiting nested block compression and column indices compression. (2) Based on the proposed compression scheme, a deeply-pipelined SpMV accelerator is implemented on an Xilinx Virtex XC7VX485T FPGA platform, which exploits the parallelism across multiple rows and the deep pipeline in the computational units. (3) Compared with the previous works on the Convey platforms (HC-1 and HC-2ex) and Nvidia Tesla S1070 GPU platform, the proposed design can obtain higher performance for most of tested matrices and improve the utilization of the memory bandwidth up to 13×.

DOI: 10.1587/elex.12.20150161
Received February 9, 2015
Accepted April 6, 2015
Publicized May 1, 2015
Copyedited June 10, 2015

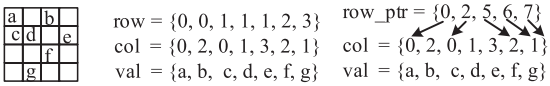
The remainder of this paper is organized as follows. Section 2 presents the background of SpMV and the related works on FPGA. Section 3 is devoted to detailing the proposed compression scheme. Section 4 illustrates in detail the deeply-pipelined accelerator of SpMV on FPGA. In Section 5, the performance is evaluated and compared. Finally, the main conclusion is drawn in Section 6.

2 Background and related work

2.1 Background

Sparse matrix-vector multiplication performs the operation $y = Ax$, where A is a large and sparse matrix, and x and y are dense vectors. In order to achieve higher performance, it is required that designing the proper compression scheme of the sparse matrix to fully utilize the underlying system architecture. There have been many compression schemes proposed in the literature, such as Coordinate (COO), Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), ELLPACK-ITPACK (ELLPACK), and so on.

For convenience of understanding the VBW-CBQCSR format, the basic storage formats, COO and CSR, are presented here. Taking the sparse matrix in Fig. 1(a) for an example, the COO format, as shown in Fig. 1(b), consists of three *nnz*-items arrays, *val*, *row* and *col*, where *nnz* is the number of nonzero elements in the sparse matrix. The *val* array stores the nonzero elements in the row-major order, and the responding row and column indices are stored in the *row* and *col* respectively in the one-to-one way. The CSR format is illustrated in Fig. 1(c), which consists of three arrays, *val*, *col*, and *row_ptr*. As the COO format, the nonzero elements and corresponding column indices are stored in the *val* and *col* in the row-major order, respectively. The *row_ptr* stores the start position of each row in the *val* and *col*.



(a) sparse matrix (b) COO Format (c) CSR Format

Fig. 1. Conventional COO and CSR format

2.2 Related work

There has been a substantial body of works to implement the SpMV on FPGA, which can be divided into two categories, the novel compression schemes and pipelined architecture based on the conventional compression schemes.

From the perspective of the novel compression schemes, due to the limited capacity of the on-chip Block RAM, block is one of the most used ways to compress the sparse matrices. Given the disposal of the submatrices, the block scheme can be divided into two groups. In the first group, such as BCSR [3] and Row Blocked CSR [4], the submatrices are considered as dense matrices, and the index data is reduced by only recording the indices of the nonzero blocks, instead of each nonzero element. However, the excessive zero paddings to construct the dense submatrices degrade the performance. In the second group, the submatrices are taken as sparse matrices, and compressed with specific scheme to decrease the

DOI: 10.1587/elex.12.20150161
Received February 9, 2015
Accepted April 6, 2015
Publicized May 1, 2015
Copyedited June 10, 2015

redundant computation and memory requirement by reducing the zero paddings [5, 6]. However, the overhead of the word-level-encoded index data of each nonzero element limits the performance improvement. As the works in [7, 8, 9], the overhead can be reduced by replacing the indices with bitmap, and the indices are retrieved through the decoding before the computing. However, the performance of these works is restricted by the idle cycles in the index decoding and the zero fillings in the bitmap.

The works employing the architecture-specific optimization focus on the design of pipelined multiply-accumulate units to reduce the computational time. Zhuo, et al., employ a tree-based multiply accumulate unit and a reduction unit to perform multiple operations in parallel [10]. However, the structure of the reduction unit depends on the sparsity pattern and there are a large number of zero paddings to meet the alignment requirement of the adder tree. The reduction is redesigned in [4], and exclusively shared among multiple processing elements (PEs), which results in the pipeline stalls, when more than one partial sum become available at one clock cycle. A similar design is proposed in [5], which employs the multiple-input-multiple-output multiply-accumulate unit and a reduction unit to process multiple rows at one clock cycle, however the serial reduction limits the performance. Song Sun, et al. make use of the input pattern vector (IPV) and map table to implement SpMV without pipeline stall and excessive zero-paddings [11], however, the storage of IPV and map table limits the dimension of the sparse matrix. K. Nagar, et al. [12] implemented SpMV for large-scale sparse matrices on the Convey HC-1 with a novel streaming multiply-accumulator and local vector cache. Further, A hardware multithreaded implementation of SpMV on the Convey HC-2ex, which makes use of multiple outstanding memory requests to mask the long latencies and multiple Computation Engines to process multiple rows in parallel [13]. However, the performance improvement of the above two implementations mainly depend on the high bandwidth and multiple memory controllers, which are greatly excessive of other platforms.

3 The proposed VBW-CBQCSR compression scheme

In order to fully utilize the bit capacity of FPGA to improve the performance of SpMV, a hardware-friendly compression scheme, named as VBW-CBQCSR, is proposed. The VBW-CBQCSR scheme consists of two parts, CBQCSR and VBW, which are used to compress the sparse matrix and the column indices of the nonzero elements, respectively.

As shown in Fig. 2, the CBQCSR scheme partitions the sparse matrix in the 2D uniform way, and stores the submatrices in a two-level storage format. For the first level storage scheme, a quasi-COO format is used to store the indices of the nonzero submatrices. The row and column indices of the submatrices are stored in array *brow* and *bcol* in a row-wise order. The array *bval* is used to store the start position of each submatrix in the second level storage scheme.

For the second-level storage scheme, a quasi compressed sparse row (QCSR) scheme is proposed to reduce the memory access and redundant computation. The QCSR scheme contains three 1D arrays: *val*, *col* and *EOR*. The values and column

DOI: 10.1587/elex.12.20150161
Received February 9, 2015
Accepted April 6, 2015
Publicized May 1, 2015
Copyedited June 10, 2015

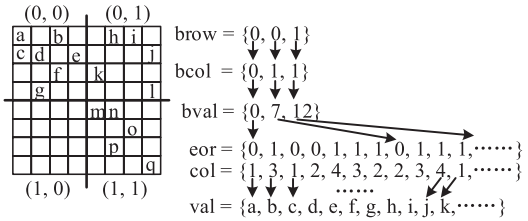


Fig. 2. Structure of CBQCSR format

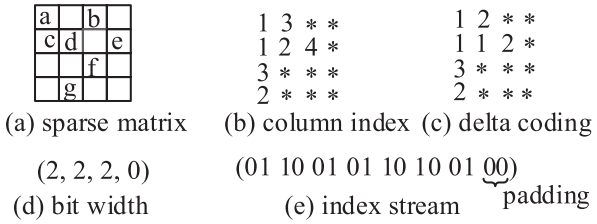


Fig. 3. Block-wise variable-bit-width compression scheme

indices of the nonzero elements are stored in row-wise order in the *val* array and *col* array in a one-to-one manner. Instead of the *row_ptr* array in the conventional CSR scheme, the EOR (end-of-row) flags are introduced to mark the termination of each row of the nonzero submatrices. The EOR flags are stored in the *EOR* array. When the value of *EOR[i]* is one, the corresponding *val[i]* and *col[i]* are the last items of one row. Through the *EOR* flag, the parallelism across multiple rows can be exploited to improve the performance.

The main idea of the VBW part is to make use of the variable bit width encoding scheme to reduce the number of bits required to store the column indices. Taking the sparse matrix in Fig. 3(a) as an example, the column indices of the nonzero elements are listed in the row-wise order in Fig. 3(b), where the asterisk $*$ marks the invalid data and is not stored. Next, the column indices are encoded row-by-row using the delta encoding scheme (Fig. 3(c)), which only stores increment relative to the previous column index to reduce the bits required. Instead of storing the bit-width of each nonzero element, the bit-width is stored column by column to reduce the memory space. The bit-width of each column is calculated by the Equation (1) shown in Fig. 3(d), where b_j represents the bit width of the j-th column, and λ_j represents the maximum value of the delta-encoded index data in the j-th column. Finally, the delta indices are encoded according to the bit-width required for each column and compressed into the bit streams by the unit of the memory bandwidth *mem_len*. Assuming that the *mem_len* is 16 and the block size is 1024, the bits required for the column indices is 16 bits, as shown in Fig. 3(e), compared with the original *col* array which requires 70 bits.

$$b_j = \begin{cases} \lceil \log_2 \lambda_j \rceil & \lambda_j \neq 2^i \\ (\log_2 \lambda_j) + 1 & \lambda_j = 2^i \end{cases} (i = 0, \cdots, n) \quad (1)$$

DOI: 10.1587/elex.12.20150161
Received February 9, 2015
Accepted April 6, 2015
Publicized May 1, 2015
Copyedited June 10, 2015

4 SpMV accelerator based on the VBW-CBQCSR scheme

4.1 Overall architecture of the SpMV accelerator

Based on the proposed VBW-CBQCSR scheme, a deeply-pipelined SpMV accelerator is implemented on a self-designed FPGA platform with one Xilinx Virtex-7 FPGA and three external DRAM Memory modules, as shown in Fig. 4. The sparse matrix and vector x are all stored in the external DRAM Memory modules. The processing elements (PE[1],…, PE[n]) access the data through the Customized Memory Interface and execute the SpMV on different block rows in parallel. When the computation of one block row is finished, the results of the vector y are written back to the external DRAM Memory modules.

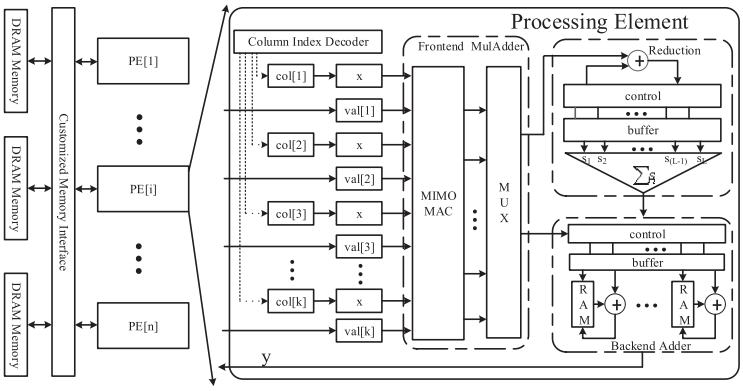


Fig. 4. Overall structure of SpMV accelerator

Each PE contains four main components, Column Index Decoder, Frontend MulAdder, Reduction and Backend Adder. The Column Index Decoder module retrieves k column indices in parallel, which are used to access the data of the vector x. The k pairs of $x[i]$ and $val[i]$ are multiplied and accumulated by the Frontend MulAdder module without the limitation of the same row, as the approach in [5]. The partial sums with $pre\_EOR[k] = 0$ are fed into the Reduction module, where $pre\_EOR[k]$ stands for the EOR flag of the last k-th partial sum. The Reduction module accumulates the partial sums of the same row as the approach in [5]. Others can be directly fed into the Backend Adder module. The Backend Adder module adopts a multi-bank architecture to accumulate the partial sums of the block being processed with the partial sums out of the same rows in the previous blocks in parallel. After the computation of one block row is finished, the results of the vector y are written back to the external DRAM Memory modules.

4.2 Structure of the Column-Index Decoder

As shown in Fig. 5(a), the Column-Index Decoder module mainly consists of the CIS RAM and k index decoders (Dec[1],…, Dec[k]), where the CIS RAM is used to store the compressed index streams and the k index decoders are used to perform the on-the-fly decoding of the column indices in parallel.

The proceeding of index decoding is shown in Fig. 5(b). Initially, the compressed index streams of address 0 and 1 are written into two registers, $cstr0$ and $cstr1$. The bit width b, required for the nonzero column index, is read from the bw

DOI: 10.1587/elex.12.20150161
Received February 9, 2015
Accepted April 6, 2015
Publicized May 1, 2015
Copyedited June 10, 2015

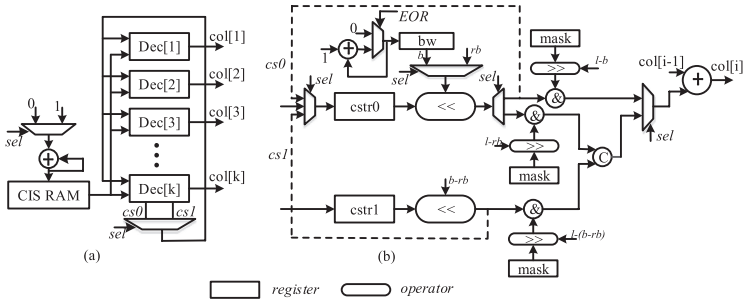


Fig. 5. Structure of the Column-Index Decoder

RAM. When b is less than the number of remaining bits (rb) in the compressed stream $cs0$, the sel signal is true and the $cstr0$ is shifted left by b bits. The most significant l bits are extracted from the shifted $cstr0$ by the shifted mask data to gain the delta result. Otherwise, the $cstr0$ is shifted by rb bits, and the most significant l bits are selected by the shifted mask data. Then the $cstr1$ is shifted left by (l-rb) bits, and the shifted result is combined with the one gained from the $cstr0$ to obtain the delta result. Finally, the column index col is gained by the addition with the previous column index $col[i-1]$. When the sel is true, the shifted $cs0$ is fed back into the $cstr0$ and the $cs1$ remains unchanged. Otherwise the shifted $cs1$ is fed back into the $cstr0$ and the compressed index stream in the next address is read into the $cstr1$.

4.3 Structure of the backend adder module

When the k nonzero inputs are not from the same row, there will be more than one partial sum generated at one clock cycle. Due to blocking, these partial sums need to be accumulated with the partial sums out of the previous nonzero submatrices of the same row, which are stored in the Block RAM (BRAM) on chip. Due to only up to two read/write ports with one BRAM, there may exist port conflicts to deal with the accumulations at one clock cycle.

In order to deal with the port conflicts, a k-bank linearly-addressed memory structure is adopted in the accelerator. For the sake of simplicity, Bank0 is chosen as an example to illustrate the structure shown in Fig. 6. Bank0 consists of a buffer buf, an pipelined adder, y RAM and some multiplexers. The partial sums $(S_1, \ldots, S_k)$, of which the least significant $\log_2 k$ bits are all zeros, are fed into the Bank0. The partial sum with the lowest row index is fed to the pipelined adder with the previous partial sum read from the y RAM according to the row address r. The result is written back into the y RAM in the same address r. The left partial sums, if exist, are written into buf. When there are no new inputs, the elements in buf are proceeded in the same way according to the row index. The processing of the left banks is the same.

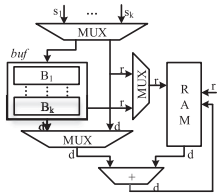


Fig. 6. Structure of the accBlock module

DOI: 10.1587/elex.12.20150161
Received February 9, 2015
Accepted April 6, 2015
Publicized May 1, 2015
Copyedited June 10, 2015

5 Experimental result

5.1 Synthesis results

For the performance evaluation and comparison, the SpMV accelerator is implemented on an Xilinx Virtex-7 XC7VX485T FPGA platform. The design is described in RTL with Verilog HDL and synthesized with ISE 14.2. Our design is determined by several parameters, i.e., the block size b, the number of PE n, the number of the frontend multipliers k, and the pipeline depth of the floating point adder h. In our experiment, $b = 1024$, $n = 3$, $k = 8$, and $h = 8$. The synthesis results reported by the Xilinx ISE are given in Table I. The maximum frequency of about 150 MHz can be achieved.

Table I. Synthesis results of the proposed SpMV accelerator

Resource	LUT	Register	DSP48E	BRAMs	Freq (MHz)
Occupy	182157	248952	144	594	151.66
Ratio	60%	41%	5%	58%	

5.2 Benchmark matrices

The experiments are conducted on sparse matrices out of the University of Florida Sparse Matrix Collection [14]. As listed in Table II, these matrices are from different applications with different characteristics.

Table II. Characteristics of the benchmark matrices

No.	Matrix	Application Domain	Dimension	non-zeros
1	dw8192	Electromagnetic	8,192	41,746
2	epb1	Thermal	14,734	95,053
3	raefsky1	Fluid Dynamics	3,242	293,409
4	psmigr_2	Economics	3,140	540,022
5	torso2	2D models of a torso	115,967	1,033,473
6	mac_*econ_fwd*500	Macroeconomic	206,500	1,273,389
7	cop20k_*A*	Accelerator cavity	121,192	1,362,087
8	cant	FEM cantilever	62,451	2,034,917
9	mc2depi	Markov	525,825	2,100,225
10	pdb1HYS	1HYS Protein Bank	36,417	2,190,591
11	consph	FEM spheres	83,334	3,046,907

5.3 Comparison of the column index overhead

Table III shows the space saving (in percentage) of the index data in the VBW-CBQCSR scheme, compared to the conventional CSR scheme. It can be seen that the space saving ranges from 67.9% to 84.2%, with an average 74.5%, which mainly results from the nested block scheme and the block-wise variable-bit-width delta encoding scheme.

DOI: 10.1587/elex.12.20150161
Received February 9, 2015
Accepted April 6, 2015
Publicized May 1, 2015
Copyedited June 10, 2015

Table III. Space saving achieved in VBW-CBQCSR format

No.	Matrix	saving (%)	No.	Matrix	saving (%)
1	dw8192	73.2	7	cop20k_*A*	73.9
2	epb1	72.7	8	cant	76.7
3	raefsky1	77.2	9	mc2depi	70.5
4	psmigr_2	71.9	10	pdb1HYS	84.2
5	torso2	78.5	11	consph	81.2
6	mac_econ_fwd500	67.9			

5.4 Performance comparison

As shown in Fig. 7, the proposed accelerator can obtain higher performance for most of the test matrices, compared with the implementations on the Convey HC-2ex platform with four Virtex-6 LX760 FPGAs [13], HC-1 [12] and Tesla S1070 [7]. With the number of the nonzero block in one block row and the density of one increasing, the performance improvement can be higher. The performance drop for some matrices mainly results from the frequent accesses to the off-chip memory for vector x due to block.

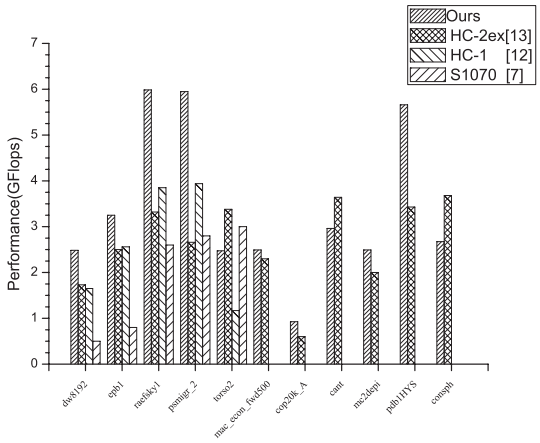


Fig. 7. Performance comparison on different platforms

Because of the memory-bound characteristic of SpMV, it is unfair that the pure comparison of the performance alone on different platforms with vastly different memory bandwidth (80 GB/s on Convey platforms, 102 GB/s on Tesla S1070, and 38.4 GB/s on our platform). A much fairer metric is the utilization of the memory bandwidth, which captures the overall efficiency of the accelerator. The ratio utilization is calculated as $R = \frac{Perf}{PB} \times 100\%$, where $Perf$ represents the computation performance measured in GFlops, and PB refers to the peak memory bandwidth.

As shown in Fig. 8, the utilization of the memory bandwidth can be improved for the tested matrices, ranging from 2.4% to 15.6%. This mainly results from the block-based nested compression scheme and variable-bit-width column-index compression scheme, which can significantly reduce the memory access requirement and redundant computation. Furthermore, the proposed accelerator can process multiple elements from different rows in parallel, which can reduce the excessive zero-padding and increase the parallelism further.

DOI: 10.1587/elex.12.20150161
Received February 9, 2015
Accepted April 6, 2015
Publicized May 1, 2015
Copyedited June 10, 2015

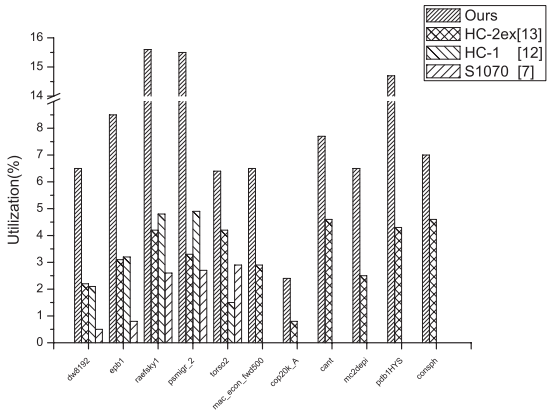


Fig. 8. Comparison of memory bandwidth utilization

6 Conclusion

This paper presents a deeply-pipelined SpMV accelerator on FPGA using a hardware-friendly compression scheme. By employing nested block compression and variable-bit-width column index compression, the compression scheme can greatly reduce the redundant computation and memory accesses. Based on this scheme, a SpMV accelerator is implemented on an Xilinx Virtex XC7VX485T FPGA platform, which can handle sparse matrices with arbitrary size and sparsity pattern. The accelerator can exploit the parallelism across multiple rows to improve the performance. Experimental results show that the proposed design can obtain higher performance for most of the tested matrices, and improve the utilization of the memory bandwidth up to 13.0×, compared with the previous works implemented on the Convey platforms and Tesla S1070 GPU platform.

Acknowledgments

This work is supported by the National High Technology Research and Development Program of China under No. 2012AA012706 and the Scientific Research Fund Project in Hunan Province under No. YB2013B007.

DOI: 10.1587/elex.12.20150161
Received February 9, 2015
Accepted April 6, 2015
Publicized May 1, 2015
Copyedited June 10, 2015