Improving SpMV Performance on FPGAs through Lossless Nonzero Compression

Paul Grigoras, Pavel Burovskiy, Eddie Hung, Wayne Luk
Department of Computing
Imperial College London
180 Queen's Gate, London SW7 2AZ, UK
Email: paul.grigoras09@imperial.ac.uk

***Abstract*—Sparse matrix vector multiplication (SpMV) is an important kernel in many areas of scientific computing, especially as a building block for iterative linear system solvers. We study how lossless nonzero compression can be used to overcome memory bandwidth limitations in FPGA-based SpMV implementations. We introduce a dictionary-based compression algorithm which reduces redundant nonzero values to improve memory bandwidth without reducing computation efficiency by making use of spare FPGA resources. We show how a sparse matrix in the CSR format can be converted to the proposed storage format on the CPU and that average compression ratios of 1.14 - 1.40 and up to 2.65 times can be achieved, over CSR, for relevant matrices in our benchmarks.**

I. INTRODUCTION

In memory-bound designs, compute resources will stall when there is insufficient memory bandwidth to supply the data required. A common technique for overcoming this issue is by compressing the input data prior to transfer, and transparently decompressing it on-chip. This can lead to higher effective bandwidth and is especially useful for common memory bound applications such as sparse matrix vector multiplication (SpMV). We show how spare FPGA resources in SpMV designs can be repurposed to perform this task in a lightweight fashion, leading to higher performance.

The main challenge of implementing compression and decompression of scientific floating point data for *improving performance* of SpMV on FPGAs, rather than just reducing storage requirements, is that high throughput rates must be achieved with minimal resource usage. Improving overall performance requires careful balancing of achieved compression ratio, compression and decompression throughput and resource utilisation. General purpose algorithms such as gzip do not perform well under these constraints since the decompression must be performed in the limits of what spare resources are available on the device, after the implementation of state of the art SpMV architectures and other blocks, without stalling the existing compute units.

We analyse the potential of using compression of double precision matrix values to improve the performance of SpMV applications on FPGAs. Our contributions are:

- A novel approach using dictionary-based compression to eliminate redundancies in nonzero values without reducing computational efficiency, by making use of spare FPGA resources.
- Implementation of the proposed approach on a Xilinx Virtex 6 and corresponding software version on an Intel Xeon processor.
- Evaluation of the proposed approach on a relevant subset of the University of Florida sparse matrix collection [1].

II. BACKGROUND

Compressing Sparse Matrices. When compressing matrix metadata the matrix sparsity pattern and order can be exploited to reduce the number of bits required for encoding the position of nonzero elements. This can be achieved either through delta coding (where the first position in each row is stored and the subsequent row entries are stored as a difference from the first position) as proposed in [2], or more elaborate pattern based compression techniques as shown in [2], [3]. Compression of matrix metadata may result in good compression ratios of up to 40%.

Generally, it is more difficult to compress the nonzero values since the data has a wider range. General techniques for compressing floating point values have been proposed and can achieve good compression rates. [4] uses a prediction based algorithm to compress a stream of double precision scientific data. The method relies on the *smoothness* of the data and the fact that successive (small) differences can be encoded with a small number of bits. The CSRVI format [5] is a simple dictionary based compression algorithm for compressing the nonzero values of a sparse matrix — it builds a table of *all* unique values and replaces duplicates with an index in this table. Both proposals achieve good compression ratio and high throughput but may require a large amount of on-chip memory making them unsuitable for FPGA as proposed.

SpMV Compression on FPGAs. [6] is one of the first to propose different compression and storage techniques to support encoding/decoding of various matrix formats directly on the FPGA. It proposes an efficient FPGA storage format — Compressed Variable-Length Bit Vector (CVBV), in which the bitmap of nonzero elements is compressed using variable width run-length encoding. The nonzero values themselves are not encoded but the overall savings achieved over a wide range of sparse matrices from the UoF collection [1] are promising: on average 1.25 and up to 1.43 over CSR. Section IV shows that by compressing nonzero values, compression ratios of on average 1.14 - 1.40 and up to 2.65 times can be achieved, over CSR, for particular matrices in our benchmarks.

III. The Bounded CSRVI Format

Since SpMV designs are primarily memory bound, compression could be used to improve performance. We consider a typical accelerator model, as shown in Figure 1: a CPU system is connected to an FPGA based accelerator. Both have large on-board DRAM available and the data is initially in the memory of the CPU system. For many iterative applications, it is feasible to perform compression of the input data before loading it into accelerator memory. This reduces traffic over the slow interconnect between the CPU and accelerator systems and from accelerator DRAM to the FPGA. By using a format which achieves good compression and decompression throughput and achieves a good compression ratio, higher effective bandwidth can be achieved on the FPGA, which leads to improved overall performance.

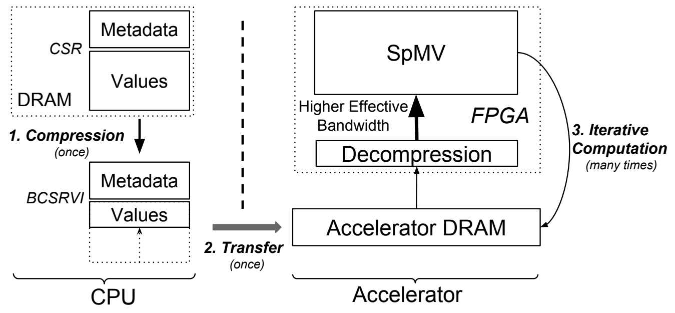

Fig. 1. Overview of the proposed approach.

Table I shows that previous SpMV designs have spare resources available which could be used for compression, in order to increase effective memory bandwidth.

TABLE I. Summary of previous SpMV work.

Ref / Device	Slices	BRAM	DSP	BRAM	PEs
	% used / # free			Size	
[7] XC2V6000	72% / 9K	20% / 120	N/A	18K	6
[8] XC2VP70	73% / 9K	N/A	N/A	18K	8
[9] X2CV6000	36% / 21K	23% / 111	N/A	18K	3
[10] XC4VLX200	20% / 71K	N/A	67% / 32	18K	8
[11] XC5VLX330	50% / 26K	73% / 78	50% / 192	36K	8
[6] XC5VLX155T	74% / 25K	87% / 28	91% / 12	36K	16
[12] XC5VSX95T	53% / 27K	65% / 84	50% / 320	36K	64
[13] EP1S80	30% / 25K	40% / 218	N/A	4K	8
[14] S3 EP3SE260	31% / 70K	84% / 8	N/A	144K	6
[15] 5SGSD5	38% / 107K	27% / 1.4K	2% / 1558	20K	32

We therefore propose a novel, dictionary based compression format to improve the effective memory bandwidth of SpMV designs. The proposed format reduces redundancies in the matrix values by replacing frequently recurring values with smaller bit-width indices into a decoding table. The proposed format is based on the widely used Compressed Sparse Row (CSR) format.

The CSR [16] format encodes a sparse matrix with three arrays: *values* (nonzero elements of the matrix), *col_ind* (column index of each nonzero element) and *row_ptr* (start and end of each row). The CSRVI [5] format can be used to remove duplicate elements from the *values* array by replacing it with a dictionary of unique nonzero values and an array of indices into this dictionary. This approach performs well when repeated values are present. However this algorithm is not suited to FPGA implementation since encoding all nonzero values by indices would require a large dictionary. This is expensive in terms of BRAMs with Table I showing that BRAM is a scarce spare resource in state of the art SpMV designs.

Starting from the CSRVI format, we propose that dictionary replacement would be restricted to only the *k* most frequently occurring values in the *values* array. We call the resulting format the Bounded CSRVI (BCSRVI) format. Higher values of k may lead to better compression ratios, but may also be impossible to implement within the available resources. Therefore the underlying assumption is that, in practice, only a small number of values in the original matrix occur frequently enough to warrant this type of compression.

To simplify the hardware implementation, the initial sparse matrix is split in two non-overlapping matrices, one of integer values, A_E, and one of floating point values, A_U, such that $A = D(A_E) + A_U$. A_E contains all indices into the decompression dictionary (obtained as above), while A_U contains all remaining floating point values. D is the decoding function implemented on the FPGA which converts the indices to the corresponding nonzero values in the original matrix.

The SpMV is performed in three stages: first matrix $D(A_E)$ is multiplied by the left hand side vector, x, then matrix A_U is multiplied by x, then the two partial results are added to give the final result. Since the matrices are kept in CSR format, the only source of storage overhead is one additional row_ptr array required for the A_E matrix. The operation of the design for performing a hypothetical iterative computation involving SpMV can be summarised by Algorithm 1.

Algorithm 1 CPU operation using BCSRVI format

```
procedure ITERATIVE_SPMV_KERNEL(A)
    $A_U, A_E$ = split_encode_matrix(A)
    while necessary do
        $x_i$ = get_vector(...) // get next vector x
        $b_1$ = SpMV_on_FPGA($A_U$, $x_i$, compressed = False)
        $b_2$ = SpMV_on_FPGA($A_E$, $x_i$, compressed = True)
        $b_i = b_{i1} + b_{i2}$
    end while
end procedure
```

The advantage of this approach is the regularity of the input data: once the design is initialised, data of equal bitwidth is read at every cycle. This means decoding of that input data within the BCSRVI decoder can be performed at the rate of one value per clock cycle (without stalling the pipeline) and with minimal resource overhead. Additionally this design is easy to vectorize since at any given time all processing elements request data of equal bitwidth.

We implement the decompression and a simple sparse matrix vector multiplication kernel as outlined in [8] on the Maxeler dataflow system [17] which is a typical host-accelerator configuration. The compression is performed on the CPU of the host system. This results in an encoding table which is used to encode the original data and a decoding table which is copied into the BRAMs of the accelerator for decompression. During the first stage of execution (computing $D(A_E)x$) the stream of values is used as an index into the decoding table which is stored in on-chip memory. The value read from the decoding table is forwarded to the SpMV kernel. During the second stage of execution (computing A_Ux) the decoding table is not used, and the matrix values read from DRAM are simply forwarded to the SpMV kernel. The logic required is minimal: one BRAM to store the decoding table, one multiplexer, and one casting block to convert the floating point values to integer indices when required. Furthermore, the dual port configuration of the Xilinx BRAM can be used to share decoding tables between two SpMV processing elements (since, as can be seen from Table I, all state of the art implementations use multiple processing elements to increase the level of parallelism).

Compression on the host CPU involves two steps: *1)* build encoding and decoding tables, which involves constructing a frequency count for each unique nonzero value of the original matrix, as shown in Algorithm 2; *2)* use the encoding table to split the original matrix in the A_U and A_E components.

Algorithm 2 Construct encoding/decoding dictionaries

```
procedure BUILD_ENCODING_DECODING(A, dict_size)
    priority_queue ← k_most_frequent_values(A, $k = 2^{\text{dict\_size}}$)
    while priority_queue is not empty do
        (index, value) ← priority_queue.extract_max()
        decoding_table[value] ← index
        encoding_table[index] ← value
    end while
return (encoding_table, decoding_table)
end procedure
```

IV. Evaluation

We evaluate the proposed approach on the same subset of the University of Florida sparse matrix (UoF) collection used in [5]. Out of the matrices used in that work the following are no longer available on UoF: `dense, e40r2000, fdif, fidap011, fidapm11, random` and are therefore not included in our benchmark. We also eliminate from this set any matrices which have maxand integer data types (as shown on UoF): `gupta1, gupta2, bcsstk32, 3dtube, jpwh_991`. The remaining benchmark comprises 86 matrices. The most significant parameters of our benchmark set are shown in Table II, where **MC(x)** is used to refer to the percent of values which can be compressed using x bits with respect to the total number of nonzeros.

TABLE II. Representative benchmark parameters.

	Average	Min	Max	Median	Quartile 2	Quartile 3
Order	270K	767	4M	41K	14K	321K
Nonzeros	7.945M	6027	77M	1M	304K	7479K
Unique	2.124M	1	37M	284K	11K	1.2M
Sparsity	0.25	9E-5	4.14	0.08	0.0073	0.28
MC(2)	0.25	0	1	0.05	0.0004	0.44
MC(5)	0.31	0	1	0.08	0.0017	0.71
MC(8)	0.35	1E-5	1	0.11	0.0048	0.76
MC(10)	0.38	5E-5	1	0.21	0.0087	0.82
MC(13)	0.46	4.2E-4	1	0.38	0.0355	0.98

Resource Usage. We implemented the proposed decoder architecture using MaxCompiler 2014.1 on a Xilinx Virtex 6 (40 nm silicon technology). We use ISE 13.3 to place and route the design. We found that the BCSRVI is easy to vectorize using the Maxeler dataflow system. This is because, when implementing multiple processing pipelines, all processing pipelines request the same amount of data (when the workload is uneven, null padding is added on the CPU). This allows us to group all data in contiguous locations in memory and address it linearly, thus achieving peak memory performance.

TABLE III. BRAM Usage for BCSRVI decoder, per processing pipeline (double precision), storage only.

log_2k	1 – 10	11	12	13	14	15
BRAMs	1	2	4	8	16	32

Table III shows that the BCSRVI decoder can be implemented with a small number of BRAMs. The Xilinx RAMB36 memories BRAMs support a 512 deep x 72 bits wide configuration. The small number of resources used shows that our proposal to use spare resources for decompressing sparse matrix values is feasible. As expected resource usage becomes prohibitive when larger dictionary tags are used to encode unique values.

Achieved Compression Ratio. We measure the achieved compression ratio and compare it against the standard double precision CSR and CSRVI representations. For the former, each entry uses 8 bytes for values and 4 bytes for metadata. For the latter each entry uses the minimal number of bits which is given by $N_{\text{bits}} = \lceil \log_2(\text{unique nonzero values}) \rceil$.

Table IV shows the compression ratio achieved for the nonzero values, normalised with respect to the CSR and CSRVI formats and also the overall compression ratio normalised with respect to CSR. Table IV shows that good compression ratio can be achieved with a limited number of bits (and therefore resource usage). In fact a significant proportion of the original CSRVI compression can be achieved with only a small number of values (less than 2^{10}).

TABLE IV. Achieved Compression Ratio normalised with respect to CSR Values, CSR Total and CSRVI Values.

	CSR Values			**CSR Total**			**CSRVI Values**		
log_2k	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg
1	17.96	0.93	1.63	2.65	0.96	1.14	2.57	0.14	0.41
2	15.13	0.93	1.76	2.60	0.96	1.19	2.16	0.21	0.44
3	14.38	0.93	2.10	2.58	0.96	1.24	1.92	0.25	0.49
4	12.10	0.93	2.16	2.52	0.96	1.26	1.65	0.25	0.50
5	11.67	0.93	2.22	2.54	0.96	1.28	1.42	0.25	0.51
6	9.94	0.93	2.16	2.48	0.96	1.28	1.25	0.25	0.51
7	8.60	0.93	2.09	2.42	0.96	1.29	1.14	0.25	0.50
8	7.58	0.93	2.03	2.36	0.96	1.29	1.13	0.25	0.50
9	6.78	0.94	1.98	2.30	0.96	1.29	1.09	0.25	0.50
10	6.13	0.94	1.93	2.25	0.96	1.30	1.09	0.26	0.49
11	5.59	0.94	1.91	2.20	0.96	1.31	1.11	0.26	0.50
12	5.14	0.94	1.92	2.15	0.96	1.32	1.11	0.28	0.51
13	4.76	0.94	1.93	2.10	0.96	1.34	1.11	0.30	0.52
14	4.43	0.95	1.99	2.06	0.97	1.36	1.09	0.31	0.54
15	4.14	0.95	2.08	2.01	0.97	1.40	1.07	0.32	0.57

When considering only the nonzero values (thus not including CSR metadata), we note that compression rates of up to 17.96 can be achieved on particular matrices and on average compression rates of $1.63 - 2.22$ can be achieved with up to 5 bits. It is, however, possible to achieve negative compression ratios on matrices where not many frequently recurring values exist due to the added overhead (duplicating row_ptr in BCSRVI). Such cases can be detected on the CPU early on (when creating the frequency table) and the negative impact can be avoided simply by setting $k = 0$, which gracefully degrades to the standard CSR on both CPU and FPGA. We note that some performance loss is to be expected (since we cannot avoid building the frequency table).

When compared with the full CSR (thus taking into account all metadata) we observe compression ratios of up to 2.65

times and on average between 1.14 – 1.40 when using up to 15 bits. As shown in Figure 2 the average compression ratio increases with the number of bits, but the maximum (which is achieved on a few special matrices) decreases, since more bits are used to represent values which are not duplicated (as expained below). We note however that this is assuming full, uncompressed metadata. Our approach can be applied in *addition to* existing methods for compressing CSR metadata, such as those presented in [2], [3], [6], in which case the normalised performance impact would be greater.

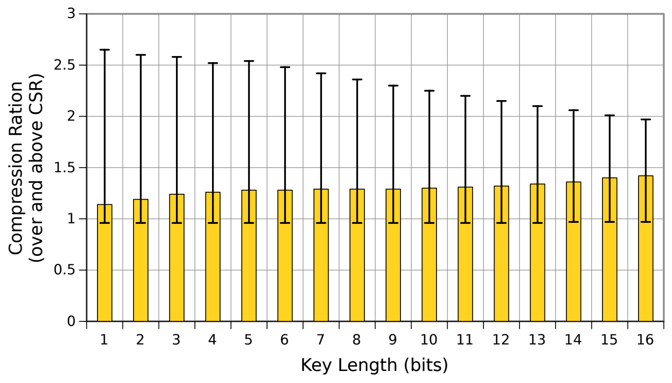

Fig. 2. Minimum/average/maximum compression ratio measured over the entire benchmark for fixed number of bits per entry versus number of bits used per entry (double precision). The bar represents average and the lower/upper tails of the error line indicate minimum/maximum compression ratio respectively.

Figure 3 shows that, if we are able to select the number of bits at runtime, based on the frequency of nonzero matrix values, better compression rates can be achieved, for a large proportion of our benchmark set.

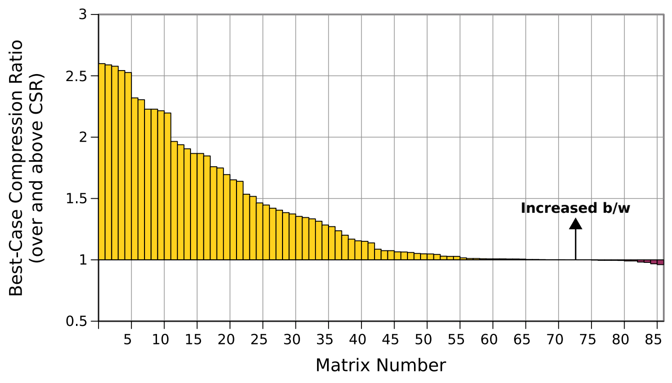

Fig. 3. Best case compression ratio for all matrices normalized with respect to CSR (double precision). For each matrix, the best number of bits (up to 12 bits) is used.

V. Conclusion

We introduce a novel format for compressing nonzero values of sparse matrices and show how the compression and decompression for the proposed format can be realised on a host CPU and FPGA respectively. We show that decompression requires few resources per processing element. It is therefore a feasible option to improve performance in many previously proposed SpMV architectures by using spare resources efficiently. The proposed format operates as an extension to the commonly used CSR format and therefore existing techniques for compressing metadata can easily be applied to further improve the achieved performance. Opportunities for future research include identifying additional optimisations compatible with our approach, and exploring methods that automate the adoption of the most effective compression technique for a given data set.

Acknowledgement

This work is supported in part by the European Union Seventh Framework Programme under grant agreement number 257906, 287804 and 318521, by the UK EPSRC, by the Maxeler University Programme, by the HiPEAC NoE, by Altera, and by Xilinx.

References

[1] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, p. 1, 2011.

[2] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," in *Proc. SC*. ACM, 2006, pp. 307–316.

[3] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "CSX: an extended compression format for SpMV on shared memory systems," in *ACM SIGPLAN Notices*, vol. 46, no. 8. ACM, 2011, pp. 247–256.

[4] P. Ratanaworabhan, J. Ke, and M. Burtscher, "Fast lossless compression of scientific floating-point data," in *Proc. DCC*. IEEE, 2006, pp. 133–142.

[5] K. Kourtis, G. Goumas, and N. Koziris, "Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression," in *Proc. ICPP*. IEEE, 2008, pp. 511–519.

[6] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal FPGA matrix-vector multiplication architecture," in *Proc. FCCM*, 2012, pp. 9–16.

[7] M. DeLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *Proc. FPGA*, 2005, pp. 75–85.

[8] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proc. FPGA*, 2005, pp. 63–74.

[9] D. Gregg, C. Mc Sweeney, C. McElroy, F. Connor, S. McGettrick, D. Moloney, and D. Geraghty, "FPGA based sparse matrix vector multiplication using commodity DRAM memory," in *Proc. FPL*. IEEE, 2007, pp. 786–791.

[10] G. Kuzmanov and M. Taouil, "Reconfigurable sparse/dense matrix-vector multiplier," in *Proc. ICFPT*. IEEE, 2009, pp. 483–488.

[11] K. K. Nagar and J. D. Bakos, "A sparse matrix personality for the convey hc-1," in *Proc. FCCM*, 2011.

[12] R. Dorrance, F. Ren, and D. Marković, "A Scalable Sparse Matrix-vector Multiplication Kernel for Energy-efficient Sparse-blas on FPGAs," in *Proc. FPGA*, 2014, pp. 161–170.

[13] Y. El-Kurdi, D. Giannacopoulos, and W. J. Gross, "Hardware acceleration for finite-element electromagnetics: Efficient sparse matrix floating-point computations with FPGAs," *IEEE Transactions on Magnetics*, vol. 43, no. 4, pp. 1525–1528, 2007.

[14] S. Sun, M. Monga, P. H. Jones, and J. Zambreno, "An I/O Bandwidth-Sensitive Sparse Matrix-Vector Multiplication Engine on FPGAs," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 59, no. 1, pp. 113–123, 2012.

[15] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," 2014.

[16] Y. Saad, *Iterative methods for sparse linear systems*. Society for Applied and Industrial Mathematics, 2003.

[17] O. Lindtjorn, R. G. Clapp, O. Pell, O. Mencer, M. J. Flynn, and H. Fu, "Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications," *Micro, IEEE*, vol. 31, no. 2, pp. 41–49, 2011.