An FPGA Drop-In Replacement for Universal Matrix-Vector Multiplication

Eric S. Chung‡, John D. Davis‡, Srinidhi Kestur†
‡ *Microsoft Research Silicon Valley*
† *Dept. of Computer Science and Engineering, The Pennsylvania State University*
{*joda, erchung*}*@microsoft.com, kesturvy@cse.psu.edu*

***Abstract*—We present the design and implementation of a universal, single-bitstream library for accelerating matrix-vector multiplication using FPGAs. Our library handles multiple matrix encodings ranging from dense to multiple sparse formats. A key novelty in our approach is the introduction of a hardware-optimized sparse matrix representation called *Compressed Variable-Length Bit Vector* (CVBV), which reduces the storage and bandwidth requirements up to 43% (on average 25%) compared to compressed sparse row (CSR) across all the matrices from the University of Florida Sparse Matrix Collection. Our hardware incorporates a runtime-programmable decoder that performs on-the-fly decoding of various formats such as Dense, COO, CSR, DIA, and ELL. The flexibility and scalability of our design is demonstrated across two FPGA platforms: (1) the BEE3 (Virtex-5 LX155T with 16GB of DRAM) and (2) ML605 (Virtex-6 LX240T with 2GB of DRAM). For dense matrices, our approach scales to large data sets with over 1 billion elements, and achieves robust performance independent of the matrix aspect ratio. For sparse matrices, our approach using a compressed representation reduces the overall bandwidth while also achieving comparable efficiency relative to state-of-the-art approaches. Note: this work was published in FCCM'12 [1].**

***Keywords*-FPGA; dense matrix; sparse matrix; spMV; reconfigurable computing**

I. Introduction

Heterogeneous systems that incorporate GPGPUs and FPGAs are potent candidates for improving energy efficiency while satisfying the ever-increasing demands for performance. In the space of heterogeneous computing, FPGAs offer unique advantages over coarse-grained architectures such as CPUs and GPGPUs. Aside from vastly lower power requirements (tens of watts vs. hundreds on the GPU), FPGAs can generalize more efficiently across application domains through careful exploitation of fine-grained parallelism and communication.

In this work[1], we investigate how FPGAs can be used to accelerate matrix-vector multiplication (MVM), a frequently-exercised computation that remains highly relevant in many scientific domains. A major objective in our effort is to use the FPGA as a drop-in replacement for existing tuned MVM software libraries such as MKL [2], cuBLAS [3], and cuSPARSE [4], which support a variety of matrix inputs, ranging from dense to multiple sparse encodings (e.g., Compressed Sparse Row). Another objective is to exploit the fine-grained capabilities of the FPGA to improve performance, storage, and bandwidth efficiency beyond what is possible with conventional, processor-centric matrix formats and implementations.

To use the FPGA effectively, we concentrate our efforts on specific optimizations that best exploit the flexibility of reconfigurable logic. Our design (1) only requires a single bitstream with a merged datapath to handle both dense *and* sparse formats, (2) can support arbitrary matrix sizes up to the memory capacity of the system, and (3) is agnostic to the matrix aspect ratio. Furthermore, a key novelty in our approach is a hardware-optimized format that minimizes the storage overhead used to encode matrices of any type. Compared to the traditional Coordinate (COO) or Compressed Sparse Row (CSR) formats, our Compressed Variable-Length Bit Vector (CVBV) approach substantially reduces the storage and memory bandwidth needed to handle large matrices, on average, saving 25% in bandwidth and storage across 2547 matrices [5]. This substantially increases the ceiling of performance beyond what is possible in CPUs and GPGPUs, given the bandwidth-limited nature of MVM.

The flexibility and scalability of our design is demonstrated across two FPGA platforms: (1) the BEE3 (Virtex-5 LX155T with 16GB of DRAM) and (2) ML605 (Virtex-6 LX240T with 2GB of DRAM). For dense matrices, our approach scales to large data sets with over 1 billion elements, and achieves robust performance independent of the matrix aspect ratio. For sparse matrices, our approach using a compressed representation reduces the overall bandwidth while also achieving comparable efficiency relative to state-of-the-art approaches.

Outline. The remainder of this paper is organized as follows. Section II describes the development of a universal format decoder designed to handle a wide variety of dense and sparse matrix formats. Section III presents the design of the universal MVM library and related results. We offer conclusions in Section IV.

II. A Universal FPGA Library For MVM

Our goal is to develop a single bitstream that can be treated as a "library" while being agnostic to matrix formats. Our design incorporates a flexible decoder that enables us

[1]Category 2: This work is published in FCCM'12 [1].

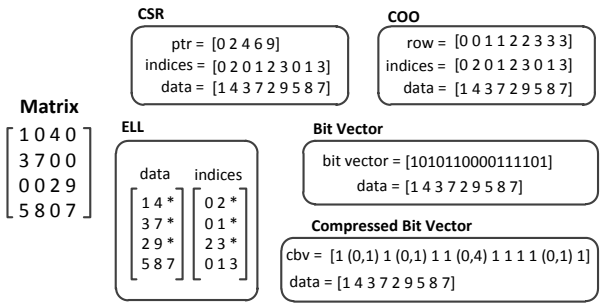

Figure 1. Various Formats Used to Encode Matrices.

to specify the characteristics of a dense or sparse matrix format at runtime. This unique feature adds a dimension of flexibility missing from other libraries or previous FPGA-based MVM kernels restricted to single formats. We first describe several popular sparse formats. We then present our hardware-optimized intermediate format, which exploits the bit-manipulation capabilities of the FPGA. Finally, we present an in-depth overview of the universal sparse format decoder.

A. Conventional Sparse Data Formats

A substantial body of literature has explored the optimization of sparse formats and algorithms for CPUs [6, 7, 2] and GPGPUs [8, 9, 10, 11, 12]. In general, these optimizations aim to minimize the irregularity of the matrix structure by selecting a format best suited for the matrix kernel. Figure 1 illustrates several well-known formats—COO, CSR, and ELL—as supported by standard sparse matrix packages like SPARSKIT [13]. We refer the reader to [1], which explains each of these formats in more detail.

A key advantage of our new format—Compressed Variable-Length Bit Vector (CVBV)—is that it can be constructed at runtime to reduce the bandwidth requirements between a host processor and the FPGA. In general, the in-memory storage requirements of our format are less than other formats, allowing larger problem sizes given a fixed amount of DRAM. Further, we can easily convert many sparse formats into our bit vector format at runtime—obviating the need for a pre-conversion step.

B. FPGA-Optimized Bit Vector Formats (*BV)

Conventional sparse matrix formats are highly processor-centric (i.e., employ word-level encodings) and do not leverage the fine-grained, bit-level manipulation capabilities of the FPGA. We propose a series of highly compact sparse matrix storage representations optimized for the FPGA, which also incur low overheads in encoding and decoding. All *BV representations are stored in a contiguous array and not constrained by word-level alignment requirements, making them ideally suited for manipulation by FPGAs.

Bit-Vector (BV). The first and simplest *bit vector* (BV) representation simply encodes the zeros and nonzeros of a matrix using a dense bitmask, where '0' denotes a zero value and '1' denotes a nonzero value in the column and row location. The storage requirement for BV is $8B \times NZ + 1bit \times M \times N$. BV has a lower sparse matrix storage requirement than CSR when the percentage of nonzero values in the matrix is greater than 3%.

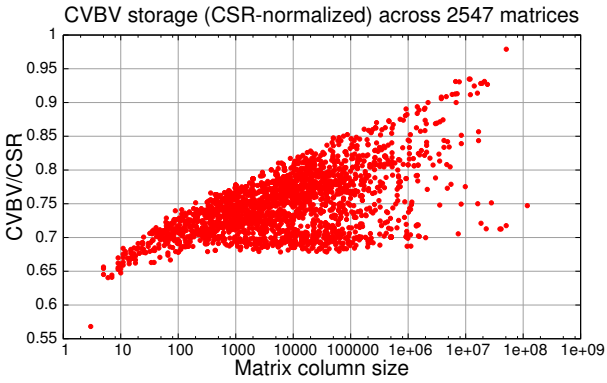

Figure 2. CVBV Storage (normalized to CSR) for all UCF Matrices [5].

Compressed Bit-Vector (CBV). An optimization of BV is the *Compressed Bit Vector* (CBV), where contiguous sequences of zeros and nonzeros are encoded using a fixed-width run-length encoding. For example, a single bit can be used to encode either a '0' (zero) or '1' (nonzero), followed by a 31-bit field that encodes the number of contiguous zeros or nonzeros. In the run-length encoding scheme, the 31-bit field must be large enough to store the largest number of contiguous values.

Compressed Variable-Length Bit-Vector (CVBV). The CBV format can be further improved using the Compressed Variable-Length Bit Vector (CVBV) scheme. In CVBV, the run-length encoding can be stored using a variable-length data field, which offers increased storage savings if cluster sizes tend to be small. In the CVBV scheme, a 4-bit header is attached to the beginning of a cluster of zeros or nonzeros. The first bit indicates a zero (0) or nonzero (1) value. The next three bits indicate how many nibbles (4-bits) are used to store the count, or up to 4 bytes.

C. Analysis of *BV

To characterize the effectiveness of the various bit-vector formats, we exhaustively profiled all 2547 matrices from the University of Florida sparse matrix collection [5]. Table I compares the storage requirements across all formats for commonly cited matrices [8, 14] and the entire collection.

A notable result of Table I is that CVBV achieves on average the lowest storage overhead (75% of CSR) and never exceeds CSR for any matrix. Across 2547 matrices, the low standard deviation of CVBV indicates the robustness of the format. Our analysis found that in many real-world matrices, the zero and nonzero elements tend to cluster—further, in all cases, cluster sizes can be encoded with a 31-bit data

	matrix	rows	cols	nonzeros	% nonzeros	CSR	COO	ELL	BV	CBV	CVBV
Selected inputs used in [8, 14]	conf5_0-4x4-10	3072	3072	119808	1.27%	1.0	1.32	0.99	1.47	0.70	0.76
	dw8192	8192	8192	41746	0.06%	1.0	1.25	1.47	16	0.69	0.83
	psmigr_2	3140	3140	540022	5.48%	1.0	1.33	13.31	0.86	0.77	0.92
	scircuit	170998	170998	958936	0.00%	1.0	1.26	59.42	300	0.71	0.83
	t2d_q9	9801	9801	87025	0.09%	1.0	1.29	0.98	12	0.68	0.76
	epb1	14734	14734	95053	0.04%	1.0	1.27	1.03	23	0.68	0.77
	raefsky1	3242	3242	294276	2.80%	1.0	1.33	1.19	1.03	0.69	0.71
	torso2	115967	115967	1033473	0.01%	1.0	1.29	1.08	131	0.70	0.76
Statistics for all matrices [5]	Mean	361922	361132	4448472	1.82%	1.0	1.26	390	928	0.89	0.75
	Median	4182	5300	40424	0.23%	1.0	1.27	2.77	5.03	0.91	0.74
	Stdev	3339878	3333546	55947269	6.10%	1.0	0.06	6092	10526	0.09	0.04
	Min	2	3	3	0.000002%	1.0	0.85	0.55	0.55	0.64	0.57
	Max	118142142	118142155	1949412601	76.0%	1.0	1.33	250108	380151	1.01	0.98

Table I
CHARACTERIZATION OF ALL SPARSE INPUT MATRICES FROM THE UNIVERSITY OF FLORIDA SPARSE MATRIX COLLECTION [5]. STORAGE VALUES NORMALIZED TO CSR FORMAT.

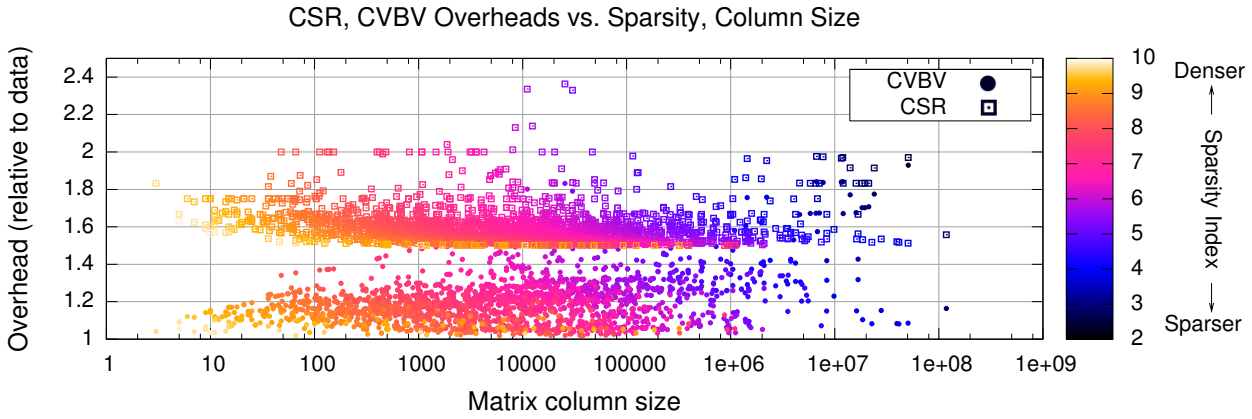

Figure 3. CSR, CVBV Overheads vs. Sparsity, Column Size (across all 2547 UCF Matrices [5]). Sparsity Index = $10 + log(nnz/total)$.

field. We found that the nonzero cluster size was relatively small—thus, the version of CVBV and CBV we employ only encodes the zeros using run-length encoding while the nonzero values are encoded as single bits.

CVBV vs. CSR. Figure 2 illustrates the storage costs (including data) of CVBV normalized to CSR across all 2547 matrices. The impact of variable-length encoding using CVBV dramatically reduces storage overheads compared to CSR by 25% on average and up to 43% in the best case. In all cases, CVBV never exceeds the storage requirements of CSR (0.98). The x-axis of Figure 2 highlights CVBV's sensitivity to the matrix column size. In general, matrices with large column dimensions and low percentage of nonzeros require more bits to encode long sequences of zeros.

Figure 3 offers a more precise characterization of the overheads of CVBV and CSR with respect to matrix sparsity and column dimension. Each matrix—represented as a point along the y-axis—quantifies the overhead of CSR or CVBV relative to the data. Each matrix in Figure 3 is color-coded according to a Sparsity Index, which characterizes the sparsity of the matrix—i.e., $log(nnz/total)$ plus a y-offset of 10. In general, darker colors indicate sparser matrices, while lighter colors indicate denser matrices.

When examining CVBV, a general trend emerges. First, matrices with small column dimensions and high density incur the lowest overheads (towards the bottom-left of Figure 3), as fewer bits are needed to encode zeros. On the other end, matrices that are sparser (darker colors) tend to incur higher overheads due to longer sequences of zeros. Finally, sparse matrices with large column dimensions incur the highest overhead as shown by several data points in the upper-right of Figure 3.

Discussion. For all *BV representations, we have opted for a simple encoding and decoding scheme. More advanced methods for encoding sparse data exist, such as arithmetic codes, but they have the drawback of being very complex to encode and decode, making them ill-suited for use at runtime, and they have much higher hardware implementation costs. The benefits of CBV and CVBV are two-fold: (1) compared to CSR, larger problem set sizes can be stored given a fixed memory capacity and (2) reduced off-chip bandwidth requirements, which raises the potential peak performance of a given MVM library. As discussed in [5], the sparse matrix size continues to grow unabated, favoring more compact sparse matrix representations.

D. Building a Universal Sparse Format Decoder

Many sparse matrix packages provide the ability to transform one sparse format into another, generally using

an intermediate representation. A unique feature of our MVM implementation is the ability to process matrices in multiple sparse formats without an explicit conversion step. Our design incorporates a runtime programmable decoder placed between the memory-to-computation datapath. The key insight behind our universal format decoder is the use of special format descriptors that enable us to programmatically convert the meta-data encoded in another sparse matrix format (i.e., COO, CSR, ELL, DIA) into CBV or CVBV.

Universal Matrix Format Decoder. Algorithm 1 describes the high-level procedure of the decoder. Any of the conventional sparse formats described in Section II-A can be characterized by several runtime parameters: (1) number of rows and columns of matrix A, (2) number of nonzero values, and (3) up to three streams of data in memory. Within the procedure, each of the streams are represented as FIFO objects that contiguously store data and meta-data about a given sparse matrix. By convention, stream[0] always refers to the data stream; stream[1] refers to the column stream; and stream[2], if required by a given sparse format, refers to the row stream.

Each stream is associated with a collection of bit fields populated at runtime. These metadata descriptors describe the meaning of the stream values and how to translate each nonzero of matrix A into matrix coordinates, which can then be used to encode the CBV or CVBV. The descriptors define parameters such as: (1) type of stream data (e.g., a nonzero value versus metadata), (2) fixed-length streams (e.g., Dense, ELL, or DIA) vs. variable-length (e.g., COO and CSR), (3) pointer to the given stream in memory, and (4) whether each element of the stream is a pointer into another array (e.g., row array in CSR) or a direct index into the matrix (e.g., column array in CSR).

For brevity, our procedure only considers row-major representations, but column-major duals also exist. In the case of fixed-length representations (e.g., ELL), we specify a *K* parameter, the number of elements in the row. Furthermore, in some cases, representations with fixed length rows have padded values, which must be stripped out of the FIFO stream using a pad token. Another descriptor is a pivot value, which is needed in formats such as DIA. The pivot is used to translate a relative offset into a real matrix coordinate. To illustrate how Algorithm 1 operates, we enumerate the parameter settings needed to support various dense and sparse formats.

- **All.** Number of rows, columns, and nonzero elements, and a $streams[0]$. Other streams null, unless specified.
- **Dense.** $streams[0]$ is fixed length of number of columns.
- **COO.** $streams[2:0]$ are variable length. $streams[2:1]$ provide absolute column and row addresses.
- **CSR.** Same as COO, except $streams[2]$ provides row pointers. Note that line 6 in Algorithm 1, calculates the correct row index by setting *K =1*.
- **ELL.** $streams[1:0]$ are fixed length of *K* and there is a pad symbol. $streams[1]$ provides column addresses.
- **DIA.** Same as ELL, pivot value set to -1. Column addresses are relative to the diagonal.

Algorithm 1 Universal Matrix Format Decoder.

Input: queue $streams[3]$
Output: Compressed data, rows, columns

```
1: data = streams[0].head
2: cx = streams[1].head
3: rx = streams[2].head
4: rowStream = FixedLenRows? stream[0] : stream[2]
5: for i = 0 → NNZ − 1 do
6:     r = RowAddress? rx : ((rowStream.idx-1)/K)
7:     c = cx + ((pivot == -1) ? r : pivot)
8:     stream[0].dequeue()
9:     if stream[1] then
10:        stream[1].dequeue()
11:    end if
12:    if RowAddress then
13:        stream[2].dequeue()
14:    else if (rx − streams[1].idx) > 1 then
15:        stream[2].dequeue()
16:    end if
17: end for
```

III. Matrix-Vector Multiplication Implementation

Our implementation of the universal MVM library builds upon the previous work of Kestur et al. [15] to develop a double-precision BLAS Level 1 and Level 2 kernel. In the original design, the authors developed a scalable stall-free accumulator that offered robust, sustained performance independent of the matrix aspect ratio. However, the original design could only support problems that fit within on-chip memory, making it less practical for real-world applications.

In this work, we extend the original BLAS architecture to support large dense matrix sizes up to the memory system capacity (in the case of a single BEE3 FPGA, 16GB or over billion matrix elements, see Figure 4). Implementing this feature required us to develop a specialized DMA engine that could support tiled accesses to external memory. Furthermore, the kernel was augmented with additional Block RAMs to buffer intermediate sub-matrices and vectors and the necessary control logic to orchestrate streaming of the tiled data to the functional units.

In addition to supporting large-scale dense matrices, we also extended the design to support sparse matrices. Beginning with the output of the universal decoder described in Section II-D, the hardware must: (1) generate and distribute the matrix rows across multiple pipelines, (2) stream in the sequential nonzero data, and (3) support random accesses to the input 'x' vector.

Our sparse architecture employs a work stealing queue and a decentralized control unit that enables decoupled

	PEs	LUT/RAM/DSP	GFLOPS		BW
	#	% Area	Peak	Sustained	% Peak
Dense V5-LX155T	16	72 / 86 / 88	3.1	0.92	64.7
Dense V6-LX240T	32	71 / 63 / 56	6.4	1.14	80
Dense+Sparse V5	16	74 / 87 / 91	3.1	Table III	

Table II
RESOURCE UTILIZATION AND DENSE PERFORMANCE.

	GFLOPS / % Peak Bandwidth Used		
	BEE3	HC-1 (32 PE) [14]	Tesla S1070 [14]
dw8192	0.10 / **10.3%**	1.7 / **13.2%**	0.5 / **3.1%**
t2d_q9	0.15 / **14.4%**	2.5 / **19.3%**	0.9 / **5.7%**
epb1	0.17 / **17.1%**	2.6 / **20.2%**	0.8 / **4.9%**
raefsky1	0.20 / **18.5%**	3.9 / **29.0%**	2.6 / **15.3%**
psmigr_2	0.20 / **18.6%**	3.9 / **29.6%**	2.8 / **16.7%**
torso2	0.04 / **4.0%**	1.2 / **9.1%**	3.0 / **18.3%**

Table III
UNIFIED MVM ARCHITECTURE PERFORMANCE AND BANDWIDTH UTILIZATION USING CSR SPARSE INPUTS (BEE3:16-PE, V5-LX155T).

random accesses to the input/output vectors to support efficient sparse matrix-vector multiplication. Because of the variability in the size and distribution of nonzero values per row, a mechanism was needed to supply memory pointers for streaming different rows into the on-chip matrix memory from contiguous off-chip memory. The work stealing queue employs three FIFOs used to store the number of nonzero values per row (NzCnt), the column indices, and the row pointer. When a pipeline dequeues the work stealing queue, it also dequeues the NzCnt from the FIFO and dequeues the corresponding number of column indices and forwards them to the consuming pipeline (see Figure 5).

The next implementation challenge involves supporting random accesses to the vector x. Due to memory port limitations and irregular access characteristics, the pipeline column address FIFO decouples the pipeline from the decoder to locally coalesce the column addresses for the corresponding entries in the matrix memory. A private vector cache per pipeline is used to request up to 4 or 8 vector x values needed to compute a dot-product. Using a cache enables us to capture the spatial locality of nonzero values observed in many of the sparse matrices.

The combination of support for large-scale dense-only MVM (DMVM), sparse MVM (SMVM), and universal decoding offers a single-bitstream solution that can be used to support a wide variety of inputs in memory—from dense matrices of arbitrary aspect ratios to sparse matrices with varying amounts of sparsity. Our design effectively provides a library comparable to the Intel's Math Kernel Library (MKL) [2] or Nvidia's cuBLAS [3] *and* cuSPARSE [4] libraries. Furthermore, we show in this work that our design is flexible and scalable across different FPGAs and platforms such as the BEE3 (16GB of DRAM, 32B DDR2 line size) and the ML605 LX240T Virtex-6 FPGA (2GB of DRAM, 64B DDR3 line size). Our design can be further mapped to newer systems with increased FPGA capacity and bandwidth (e.g., BEE4 [16]).

Area and Dense MVM Performance Results. Table II shows the resources used and dense MVM performance for several implementations: (1) dense-only MVM on a single Virtex-5 FPGA of the BEE3, (2) dense-only MVM on the Virtex-6 FPGA of the ML605, and (3) combined sparse and dense (unified) MVM on a single Virtex-5 FPGA of the BEE3. Our reported resources exclude infrastructure-related overheads such as the memory controllers. As shown in Table II, the DMVM design offers uniform sustained performance across matrix aspect ratios and for any number of matrix elements—from 66K to over 1G elements. As the results show, the sustained performance as a fraction of peak memory bandwidth is high (64.7% in the BEE3, 80% in the ML605). The ML605 experiences a slight edge in performance due to its larger burst length of 8 and increased horizontal scaling (due to the higher LUT and DSP capacity of the LX240T). Moving forward, larger Virtex-7 FPGAs or even ASICs can further enable scaling in compute resources, performance, and efficiency.

Sparse Performance Results. Table III shows the measured performance and bandwidth utilization of the BEE3 unified sparse design against previous SMVM implementations on the Convey HC-1 FPGA platform and Tesla S1070 GPGPU [14]. It is important to note that the peak performance numbers between the platforms should not be compared at face value given that the Convey HC-1 and Telsa S1070 both possess vastly superior memory systems relative to the BEE3 (80GB/s on the Convey HC-1, 102GB/s on the Tesla S1070, and 6.4GB/s on a single BEE3 FPGA). The LX330T FPGAs on the Convey HC-1 are also larger than the LX155T FPGAs on the BEE3. Because SMVM is memory-bound, a more important metric than peak performance alone is the **fraction of memory bandwidth utilized**, which captures the overall efficiency of the architecture.

Table III shows that our design achieves comparable bandwidth utilization relative to the state-of-the-art FPGA and GPGPU results. We note that because of the irregular memory accesses inherent to sparse matrices, the achieved efficiency across all platforms is highly sensitive to clustering and row length. Our current results should be treated conservatively given that there is further room for optimization and scaling in the future (e.g., adding a non-blocking, multi-banked cache, scaling to larger and/or multiple FPGAs).

From the perspective of area efficiency, Table II shows that our design does not significantly increase resource utilization over the DMVM design because the centralized control is only modified to support a work-stealing queue. The FIFOs of the work stealing queue increase the BRAM usage of our design, but do not limit the overall performance.

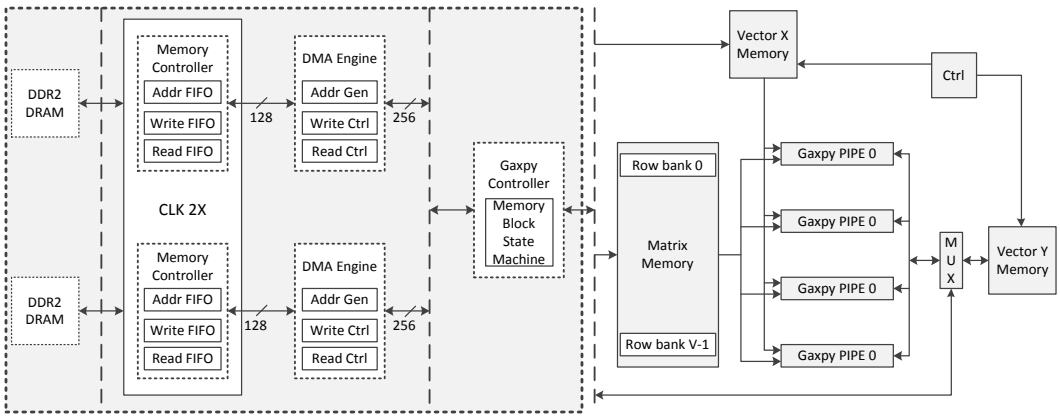

Figure 4. Large-Scale Dense Pipeline Design with New Modules (left).

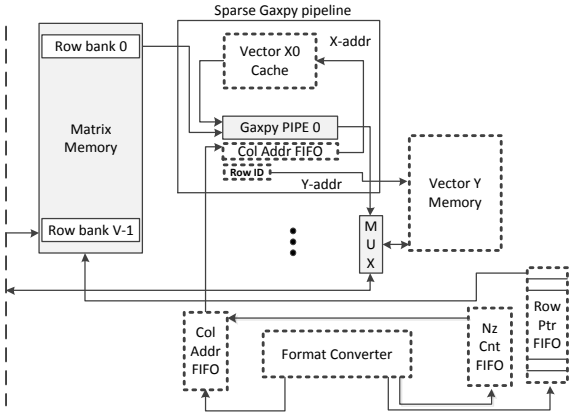

Figure 5. Large-Scale Sparse Pipeline Design.

IV. Conclusion

We presented the design and implementation of a universal matrix-vector multiplication (MVM) library to accelerate matrix algebra using FPGAs. Our design scales to over a billion elements and can flexibly support a wide variety of matrix formats using a single bitstream. Our implementation further incorporates a novel, runtime reconfigurable decoder that enables us to handle matrices of all types in memory, from dense to multiple sparse formats such as COO, CSR, ELL, and DIA. The single bitstream removes the prohibitively expensive amount of time needed to configure the FPGA for different problem classes.

By translating any matrix format into a hardware-optimized format, we reduce storage requirements by 25%, on average, and up to 43% compared to the common CSR format—while still maintaining 100% of peak and up to 25% of sustained throughput (compared to DMVM) for the sparse matrix examples used in previous studies. The reduced storage requirements of our CVBV format translates directly into supporting on average 25% larger problem sizes compared to using other processor-centric matrix formats. We demonstrate our design across two platforms: the BEE3 (V5-LX155T) and ML605 (V6-LX240T). Our designs are shown to be flexible and scalable with respect to resources and performance, riding the transistor scaling curve for new generations of FPGAs. In the future, we plan to investigate ASIC implementations of our universal MVM library in future heterogeneous devices.

References

[1] S. Kestur, J. Davis, and E. Chung, "Towards a Universal FPGA Matrix-Vector Multiplication Architecture," in *Field Programmable Custom Computing Machines, 2012. FCCM'12. 17th IEEE Symposium on*, april 2012.
[2] "Intel Math Kernel library." [Online]. Available: http://software.intel.com/en-us/intel-mkl
[3] "Nvidia CUBLAS." [Online]. Available: http://developer.nvidia.com/cublas
[4] "Nvidia cuSPARSE." [Online]. Available: http://developer.nvidia.com/cusparse
[5] T. A. Davis, "University of Florida Sparse Matrix Collection," *NA Digest*, vol. 92, 1994.
[6] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply," in *Supercomputing, ACM/IEEE 2002 Conference*, nov. 2002, p. 26.
[7] M. Smith, J. Vetter, and S. Alam, "Scientific Computing Beyond CPUs: FPGA implementations of common scientific kernels," in *MAPLD*, 2005.
[8] N. Bell and M. Garland, "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC'09. New York, NY, USA: ACM, 2009, pp. 18:1–18:11.
[9] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang, "FPGA and GPU implementation of large-scale SpMV," in *Application Specific Processors (SASP), 2010 IEEE 8th Symposium on*, june 2010, pp. 64–70.
[10] Y. Zhang, Y. Shalabi, R. Jain, K. Nagar, and J. Bakos, "FPGA vs. GPU for sparse matrix vector multiply," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, dec. 2009, pp. 255–262.
[11] V. B. Kumar, S. Joshi, S. B. Patkar, and H. Narayanan, "FPGA Based High Performance Double-Precision Matrix Multiplication," in *VLSID'09: Proceedings of the International Conference on VLSI Design*. IEEE Computer Society, 2009, pp. 341–346.
[12] "Supercomputing at 1/10th the cost." [Online]. Available: http://www.nvidia.com
[13] Y. Saad, "SPARSKIT: a basic tool kit for sparse matrix computations - Version 2," 1994.
[14] K. Nagar and J. Bakos, "A Sparse Matrix Personality for the Convey HC-1," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, may 2011, pp. 1–8.
[15] S. Kestur, J. Davis, and O. Williams, "BLAS Comparison on FPGA, CPU and GPU," in *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, july 2010, pp. 288–293.
[16] "BEE4-SOC." [Online]. Available: http://www.beecube.com