A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication

Jeremy Fowers*† Kalin Ovtcharov‡ Karin Strauss‡ Eric S. Chung‡ Greg Stitt†

†Dept. of Electrical and Computer Engineering
University of Florida
Gainesville, FL, USA

‡Microsoft Research
Redmond, WA, USA

***Abstract*—Sparse matrix-vector multiplication (SMVM) is a crucial primitive used in a variety of scientific and commercial applications. Despite having significant parallelism, SMVM is a challenging kernel to optimize due to its irregular memory access characteristics. Numerous studies have proposed the use of FPGAs to accelerate SMVM implementations. However, most prior approaches focus on parallelizing multiply-accumulate operations within a single row of the matrix (which limits parallelism if rows are small) and/or make inefficient uses of the memory system when fetching matrix and vector elements. In this paper, we introduce an FPGA-optimized SMVM architecture and a novel sparse matrix encoding that explicitly exposes parallelism across rows, while keeping the hardware complexity and on-chip memory usage low. This system compares favorably with prior FPGA SMVM implementations. For the over 700 University of Florida sparse matrices we evaluated, it also performs within about two thirds of CPU SMVM performance on average, even though it has 2.4x lower DRAM memory bandwidth, and within almost one third of GPU SVMV performance on average, even at 9x lower memory bandwidth. Additionally, it consumes only 25W, for power efficiencies 2.6x and 2.3x higher than CPU and GPU, respectively, based on maximum device power.**

Keywords-sparse matrix vector multiplication, FPGA, accelerator, SPMV, SMVM, reconfigurable computing, HPC

I. INTRODUCTION

Sparse matrix-vector multiplication (SMVM) has received significant attention due to its increasingly important application in scientific and commercial applications (e.g., computational fluid dynamics, computer vision, robotics, and structural engineering, among others). Although SMVM is a highly parallelizable algorithm, the irregular memory access patterns of real-world sparse matrices often restrict realizable parallelism. To address this problem, numerous studies have introduced specialized SMVM implementations for parallel microprocessors [19] and graphics-processing units (GPUs) [1][2].

Field-programmable gate arrays (FPGAs) are a compelling substrate for SMVM due to the availability of massive parallel resources (i.e., logic gates, on-chip memories) and a flexible interconnect to support fine-grained communication. Prior work has shown that FPGAs can perform similarly to GPUs [16], even with much lower peak memory bandwidth, and can exceed GPU performance with equivalent bandwidth [20]. Furthermore, for comparable or better performance, FPGAs consume a very small fraction of the GPU's power (e.g., 25W vs. 200W), which is a critical factor for supercomputers where energy costs can approach millions of dollars per month [6][9]. FPGA performance and efficiency has been typically obtained by efficiently parallelizing multiply-accumulate operations within a single row of the matrix [21], while also leveraging FPGA-specialized matrix encodings [11] and accumulator architectures [17][18]. In this paper, we introduce a novel FPGA accelerator for SMVM that addresses two key bottlenecks of previous approaches: 1) restrictions on exploitable parallelism, and 2) limited on-chip block RAM.

While prior approaches have shown promising performance, they can be difficult to scale due to limits on exploitable parallelism. Specifically, early works on accelerating SMVM focused mostly on exploiting parallelism within a single matrix row. For example, for an accelerator with 32 multipliers, if a given row of the matrix has less than 32 unprocessed nonzero values, the remaining multipliers will be wasted due to zero padding [21]. It is possible to begin processing the next row instead of using zero padding, but supporting an arbitrary number of rows with an arbitrary number of elements increases complexity significantly, limiting the clock rate. Ideally, an accelerator should be capable of processing elements from multiple rows of the matrix to maximize parallelism. Prior works implement support for dynamic scheduling across rows but have not demonstrated scalable performance or efficient utilization of memory bandwidth [13]. Furthermore, most prior works assumed Compressed Sparse Row (CSR) matrix encodings that are cumbersome to fetch across multiple rows for parallel processing because the matrix is encoded in a sequential, row-major fashion. This requires an entire row to be read from memory and buffered on-chip before the first element of the subsequent row can be fetched.

Another bottleneck of previous SMVM approaches is the need for replicated storage of the input vector using on-chip FPGA block RAM. Previous approaches parallelize multiplications by streaming matrix values from external memory, while reading a vector value, with one vector replica implemented in FPGA block RAM per multiplier. Although a replicated memory architecture is well-suited for small vectors, it becomes a bottleneck for highly parallelized implementations using large vectors, if they are to be stored

*Work performed while employed by Microsoft Research.

entirely on-chip. For example, for an FPGA board with ~10 GB/s of external memory bandwidth and a clock of 100 MHz, an SMVM accelerator can potentially fetch 100 bytes and execute ~25 32-bit floating-point multiplications every cycle. For a vector of 100,000 elements, previous approaches would require 10 MB of block RAM, which exceeds even the largest FPGAs. Furthermore, this bottleneck is rapidly becoming more significant due to the exponential growth of SMVM problem sizes [3]. Even for smaller vectors, replicating the vector limits usage of block RAM for other common purposes (e.g., external transfer buffers, buffers between pipelined tasks).

In this paper, we introduce a new *Compressed Interleaved Sparse Row (CISR)* matrix encoding that enables simultaneous multiply-accumulate operations on multiple rows of the matrix without the need for complex schedulers or load-balancers. We also introduce a *Banked Vector Buffer (BVB)* that supplies vector data at high bandwidth without requiring expensive replication of data, i.e., a single buffer services multiple computations simultaneously.

We evaluate our accelerator using over 700 matrices from the widely used University of Florida Sparse Matrix Collection [3] and compare our results with other platforms. We show that the FPGA **performs within about two thirds of CPU SMVM performance, even though it has 2.4x lower DRAM memory bandwidth, and within almost one third of GPU SVMV performance, even at 9x lower memory bandwidth.** Additionally, it consumes only 25W, with power efficiencies 2.6x and 2.3x higher than the CPU and GPU, respectively, based on maximum device power.

The remainder of this paper is organized as follows: Section II provides an overview of the proposed architecture, and Sections III and IV provide more details on its key components; Section V evaluates the proposal, Section VI discusses related work, and Section VII concludes.

II. Overall Architecture

The goal of our proposed design is to accelerate sparse matrix-vector multiplications of large matrices—millions of elements or more—by vectors as large as tens of thousands of elements. Our design choices are guided by two principles: (1) to enable as much parallelism as possible while keeping hardware complexity low, and (2) to eliminate the replication of vector inputs, so that larger vectors can fit on chip.

A. Limitations of Compressed Sparse Row (CSR)

Matrix-vector multiplications consist of multiple dot product operations, one for each row in the matrix. Each dot product operation requires the addition of pair-wise multiplications between elements of a matrix row and vector elements. All rows in a densely represented matrix are the same size, so the effort of parallelizing the operation across or within dot products is roughly equivalent. However, sparsely encoded matrices using the popular Compressed Sparse Row (CSR) format encode only the non-zero values of the matrix, resulting in variable-sized rows. CSR creates a trade-off in parallelization: parallelizing within dot products introduces the complexity of controlling variably-sized addition reduction operations. Conversely, parallelizing across dot products requires either a potentially large amount of buffering to store entire rows, or that multiple memory fetching points are managed and coordinated, which also introduces complexity.

B. Proposed Architecture and CISR Encoding

The proposed architecture avoids this trade-off by introducing a specialized sparse matrix encoding that explicitly exposes parallelization across rows, which eliminates the need for additional buffering or convoluted memory access and arbitration mechanisms.

Figure 1 illustrates the proposed architecture running on a single FPGA connected to two dedicated DRAM chips delivering up to 21.3GB/s of aggregate bandwidth. The design maximizes bandwidth utilization by organizing the data coming from memory into parallel *channels,* where all elements in a matrix row are processed by the same channel. When a channel exhausts a row, it fetches a new row, which is typically already pre-fetched from memory.

Each channel requests the vector elements corresponding to the newly fetched matrix elements. These vector requests are steered to the *Banked Vector Buffer (BVB)*, a multi-ported, high bandwidth structure used to store and supply vector elements. Vector elements returned by the BVB are paired with their matrix counterparts at their channel FIFOs. Finally, a multiplier pulls each pair out of their respective FIFOs, multiplies them and feeds the results into a *fused accumulator*, which is responsible for performing additions for a *group* of channels. Each fused accumulator is fully pipelined and takes turns processing the multiplier outputs of each channel in its group, performing several additions for a channel at once. When the dot product of a row completes, the accumulator places the result in the *output buffer*.

The first step in obtaining a highly parallelized design with low hardware complexity is to minimize communication and dependencies between channels. Figure 1 shows that there is no communication across channels. A factor that may limit parallelism is how data are retrieved from memory. This is the inspiration for the newly proposed *Condensed Interleaved Sparse Representation,* or *CISR* encoding.

The principle in CISR encoding is to divide the total bit width received from memory in a single cycle into slots, each corresponding to a channel. Data to be used by a channel should only be placed in its corresponding slot. This results in a directly connected design where no crossbar is required to route data from the DRAM interface into their respective channels. In other words, when data is brought from memory, each data slot lines up with a corresponding channel. This greatly simplifies the design.

The CISR encoding, like the CSR encoding, consists of three arrays, where the first encodes non-zero values, the second encodes their corresponding columns, and the third encodes information about where rows start and end. Unlike CSR, which encodes values in row-major order, CISR encodes the first and second arrays in a modified column-major order. The third array in CISR stores the length of each row, which breaks data dependencies between rows and simplifies parallelization. Section III explains the CISR encoding in detail.

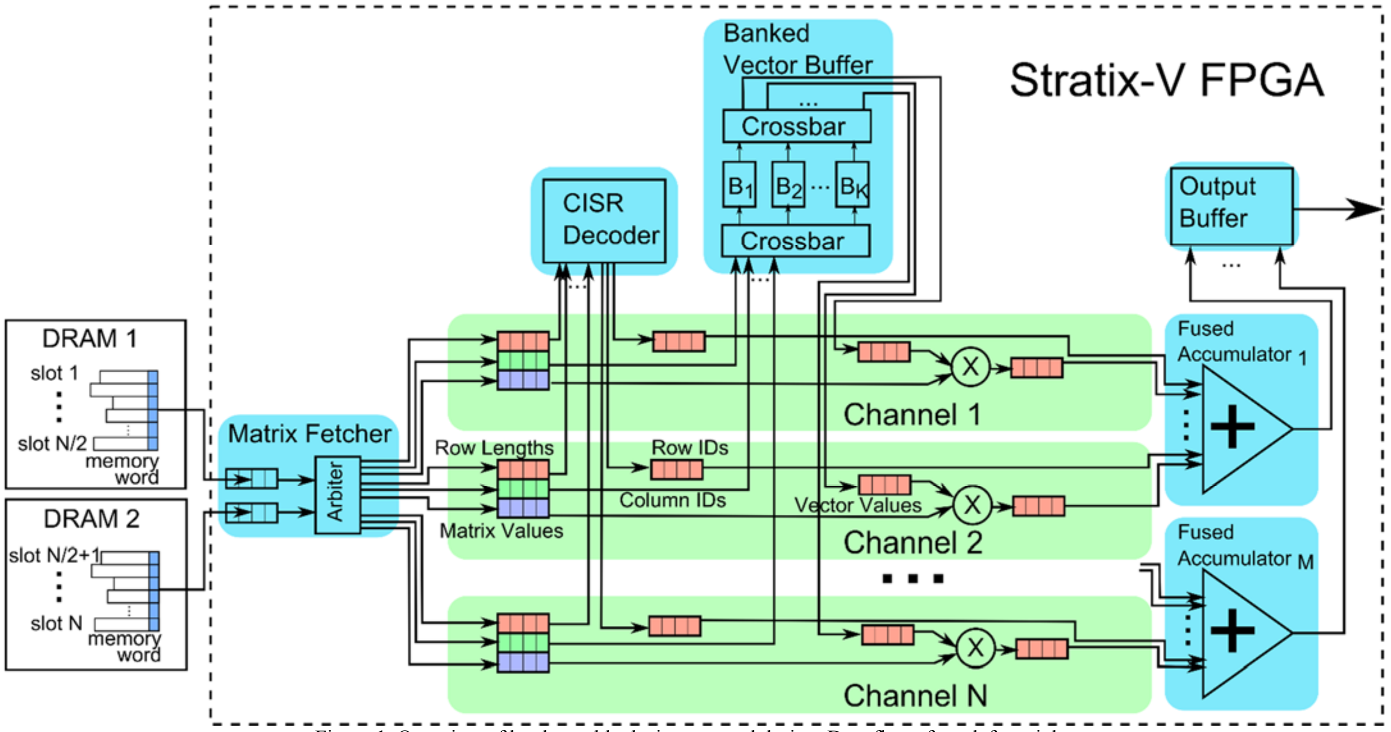


Figure 1. Overview of hardware blocks in proposed design. Data flows from left to right.

C. *De-replicating Vectors and Dynamic Fetch*

Conventional strategies for sustaining high vector access bandwidth often involve replicating input vectors across multiple sets of block RAMs. Although this strategy is simple to implement, it is difficult to scale due to limited on-chip FPGA storage. An important goal of our proposed design is to avoid the overhead and redundancy of replicated vector buffers while achieving high throughput. To meet this goal, the Banked Vector Buffer (BVB) implements a shared pool of on-chip memories connected to a highly optimized crossbar switch that services up to 32 requests every clock cycle. Section IV provides more detail on the BVB design.

III. CISR Format, Encoding and Decoding

This section compares the CISR and CSR encodings and explains why CISR leads to simpler parallel designs.

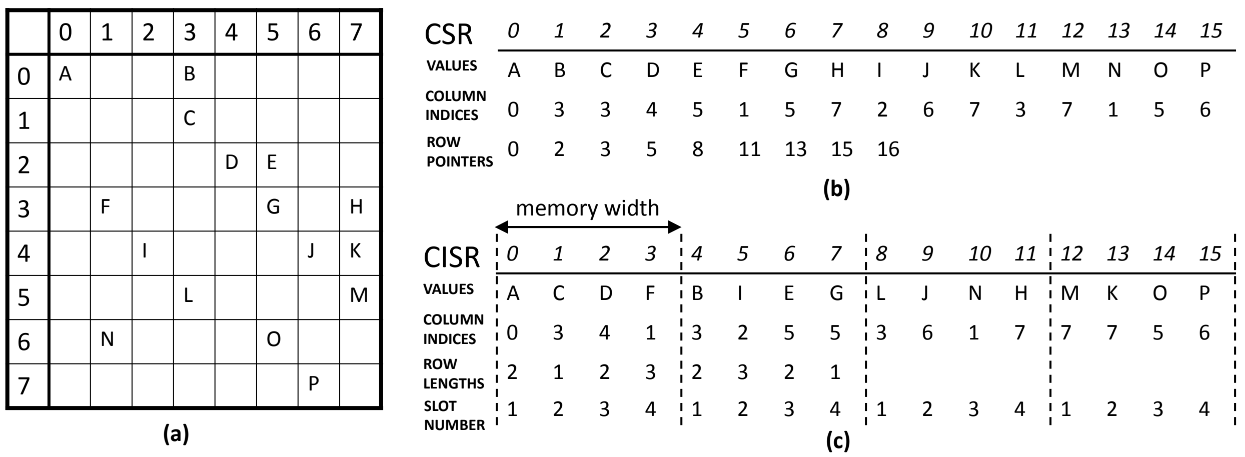
A. *CSR Encoding*

The CSR encoding uses three arrays to represent a sparse matrix. Figure 2a shows an original densely-represented sparse matrix. Letters indicate non-zero values; blanks indicate zero values. Figure 2b shows its corresponding CSR encoding. The first array (values) lists all non-zero elements in row-major order (i.e., A, B, then C, and so on). The second array (indices) lists the column index of each non-zero element and thus also follows row-major order. The ith element in the third array (row pointers) contains pointers to the position in the first array where the ith row begins. If the ith element in the third array contains the same pointer as the (i+1)th element, then the ith row contains no non-zero elements. The last element in the third array represents the number of elements in each of the first and second arrays.

It is possible, but complex, to parallelize multiplications across rows with CSR. First, enough buffer space must be provided to store a memory line's worth of non-zero values and column indices. Alternatively, multiple DRAM read streams must be managed and distributed across multiple buffers. Finally, inter-row parallelization requires sequential decoding steps to determine the boundaries between rows.

B. *CISR Encoding*

CISR encoding stores rows in a format that enables more straightforward parallelization and hardware design. One can think of CISR as a static scheduling of rows to channels using *channel slots* (referred to simply as *slots*), which are illustrated in the bottom row of Figure 2c. Figure 2c makes use of a hypothetical memory width of four elements, which results in enough per-cycle bandwidth to feed 4 channels. Therefore, four slots (numbered 1 to 4) are used to statically schedule the input to each channel. The first step for encoding a matrix into a 4-slot CISR format is to allocate each of the first four rows to one of the slots. Next, the first element in each of these rows is placed in its respective slot (A, C, D, F), then the second element, and so on. The corresponding column indices are placed in the same order in the indices array. Once one of the rows runs out of elements, a new row is assigned to that slot. For example, row 1 contains only one element (C), so on the second round of allocations, row 4 is assigned to slot 1, and I is placed in this slot of the values array, right next to B (second element in row currently allocated to slot 1). The process repeats until it exhausts the rows in the original matrix and all row elements have been



(a)

	0	1	2	3	4	5	6	7
0	A			B				
1				C				
2					D	E		
3		F				G		H
4			I				J	K
5				L				M
6		N				O		
7							P	

(b)

CSR	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
VALUES	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
COLUMN INDICES	0	3	3	4	5	1	5	7	2	6	7	3	7	1	5	6
ROW POINTERS	0	2	3	5	8	11	13	15	16							

(c)

memory width

CISR	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
VALUES	A	C	D	F	B	I	E	G	L	J	N	H	M	K	O	P
COLUMN INDICES	0	3	4	1	3	2	5	5	3	6	1	7	7	7	5	6
ROW LENGTHS	2	1	2	3	2	3	2	1								
SLOT NUMBER	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4

Figure 2. (a) Original densely represented matrix, (b) its representation in CSR and (c) its representation in CISR.

placed in the sparse encoding. At the end of this process, certain slots will be empty while other slots will still have row elements to be assigned. Special-symbol padding (e.g., zero padding, not shown) is used to fill completed slots such that all slots have the same number of values. The number of elements in each row is placed in its corresponding slot in the row length array as rows are completely placed in the values and indices arrays. Note that this effectively reorders row element counts so that the counts are placed in the same slots as their non-zero elements and indices counterparts. For example, element 4 in the row length array corresponds to row number 5, the second row to be allocated in slot 1.

This static row scheduling is beneficial because it shifts the complexity of assigning rows to hardware resources from hardware to software, which is much more flexible. Given the simplicity of the CISR encoding format, we do not expect a significant increase in the encoding time relative to CSR. On the hardware side, static scheduling allows elements, indices and row lengths of different rows to be directly passed to the hardware resources responsible for processing them, thanks to the direct correlation of memory slots to hardware channels. This obviates the need to implement dynamic row scheduling in hardware and to provide a full, difficult to manage and area-intensive crossbar between the memory buffer segments and hardware resources.

C. *CISR Decoding*

The CISR encoding process was designed to statically perform as much of the work of preparing the matrix for processing as possible, so the CISR decoding process can be very simple. Matrix values and column indices are forwarded directly from slots to the channels processing them. The only on-chip decoding required is transforming row lengths into row IDs, which inform the accumulator to which row each matrix-vector value pair belongs. First, the CISR decoder initializes sequential row IDs (i.e., each channel gets its own index minus one as the initial ID). Next, for each channel, the CISR decoder reads a value from the row length FIFO and sets a counter equal to that value. The CISR decoder decrements each counter every cycle and places a copy of each channel's row ID in that channel's row ID FIFO. When a channel's counter value reaches zero, it indicates that all of the row IDs for that row have been produced and that a new row ID must be assigned. If multiple channels need new row IDs in the same cycle, the encoding guarantees that lower-indexed channels correspond to lower-indexed row IDs. This process continues until the matrix's row length array has been exhausted, indicating that the matrix has been fully decoded.

IV. BANKED VECTOR BUFFER

The purpose of the BVB is to supply vector elements at high bandwidths to a set of channels (32 or more). To mitigate the need for replicated storage while simultaneously providing high bandwidth, the BVB is internally built out of two 32x32 input-queued crossbars connected to 32 independently accessible block RAMs. The address-request crossbar accepts a column index from each channel and routes it to one of 32 banks using a simple bank hashing function (in our design, the $\log_2$(# banks) lower-order bits of the column index). When a given column index is routed to a block RAM, the index (excluding the banking bits) is used to read the vector element from the RAM in a single clock cycle. The resulting value is then issued to one of 32 input ports in the data-response crossbar, which is used to forward results back to the requesting channel. On a bank conflict, requests are back-pressured into the crossbars' input queues, and eventually back to the channel.

The BVB crossbars are highly pipelined and can operate at up to 150 MHz after place-and-route. Each of the 32 output ports of the crossbar selects among 32 input candidates on each clock cycle using a single-cycle priority encoder. The match decision is then fed along with the input channel's data into a 6-cycle pipelined multiplexer. The BVB has enough on-chip bandwidth to sustain 32 channels simultaneously.

Although we do not discuss in detail in this paper, scaling beyond supporting 32 channels in the BVB could be achieved by increasing the degree of banking further and by using a more scalable network-on-chip instead of a high-radix crossbar that scales quadratically in area with inputs ($O(n^2)$). Alternative strategies could include more scalable topologies such as a 2-D mesh ($O(n)$). We leave this investigation to future work.

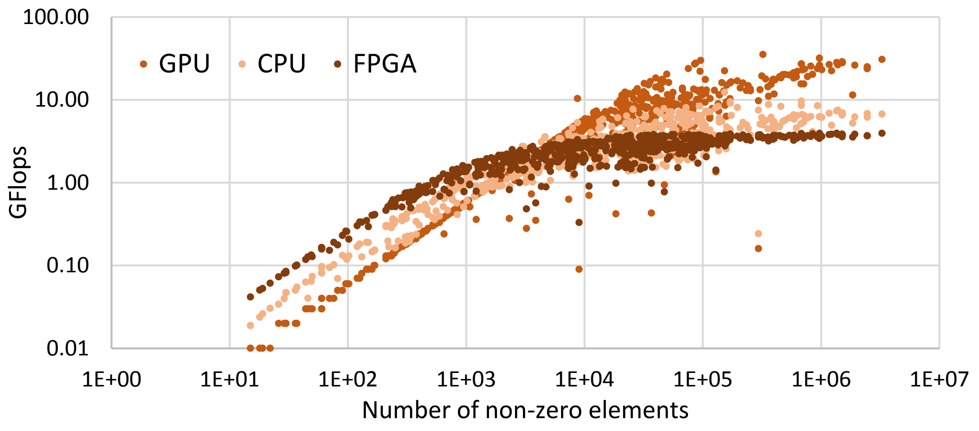


Figure 3. Overall performance comparison between FPGA, CPU, and GPU implementations of SMVM, with growing number of non-zero values. The CPU and GPU results are based on warmed (repeated) measurement runs. Axis are in logarithmic scale.

V. Evaluation

A. *Methodology*

We implemented and benchmarked the proposed SMVM accelerator on a custom FPGA PCIe card with an Altera Stratix V D5 FPGA. The selected FPGA has 172K ALMs and 39 MBits of on-chip memory. The card supports up to 8 GB of DRAM, with two dual-rank DDR3-1333 SO-DIMMs clocked at 667 MHz supporting an aggregate peak off-chip bandwidth of 21.3 GB/s. The FPGA is clocked at 150 MHz, which translates to over 1024 bits/cycle DRAM bandwidth and supplies 32 32-bit processing channels. The FPGA communicates with the host system using PCIe Gen 2x8, which supports up to 4 GB/s of bandwidth to the host memory.

We evaluate the SMVM FPGA prototype's performance using over 700 sparse matrices from the University of Florida Sparse Matrix Collection [3] with matrix dimensions that can fit into the aggregate on-chip vector buffer — currently, up to 16K elements. The number of vector elements is not a fundamental limitation of the architecture and it would be straightforward to modify the design to accommodate sizes of up to 100k on the targeted FPGA. This architecture could support even larger configurations via tiling/blocking software strategies. We leave these extensions to future work.

To demonstrate the absolute benefits of the proposed design, we compare our results to highly-optimized CPU and GPU SMVM implementations. Our CPU measurements are carried out on a quad-core Xeon E5-1620 @ 3.6 GHz running a highly tuned multithreaded CSR-format SMVM implementation from Intel's MKL library [10]. Our GPU measurements are carried out on a high-end NVidia GTX 580 running the latest version of NVidia CUSP [14]. It is worth noting that compared to the FPGA's bandwidth of 21.3 GB/s, the CPU supports up to 51.2 GB/s of DRAM bandwidth, while the GTX 580 GPU supports up to 192.2 GB/s.

In our hardware measurements, the CISR-encoded matrix is first preloaded and stored in the FPGA's DRAM. This scenario is based on the assumption that the SMVM kernel is executed iteratively on the FPGA (as is common in many use cases of SMVM, e.g., Conjugate Gradient Solver). Our performance measurements of the FPGA implementation include the time needed to stream the inputs from DRAM.

For the GPU, we measure CUSP using all supported matrix formats (e.g., CSR, ELL, HYB) and select the best performance. Our GPU results optimistically exclude the time it takes to encode sparse matrices in GPU-optimized formats (e.g., HYB or ELL), and also excludes the time it takes to load the input matrix and vector into GPU DRAM.

Our measurements include both "warm" and "cold" runs. The warm measurements involve running the GPU kernel repeatedly to include the impact of last-level caching in the GTX 580. The cold measurements ensure that all matrix and input vector data is resident only in GPU DRAM. For the CPU, we also exclude any matrix encoding time and measure performance with both warm and cold caches. The "cold" results allow us to understand the impact of off-chip memory bandwidth on the CPU and GPU, while the "warm" results give us an upper bound on CPU and GPU performance under ideal circumstances (i.e., vectors and matrix elements are cached on-chip).

Finally, we report aggregate GFlops numbers by using a time-weighted average to emphasize GFlops rates obtained for large matrices that take longer to execute.

B. *Results*

We first compare overall performance of the FPGA, CPU, and GPU, then turn our attention to a normalized memory bandwidth comparison, as well as a power efficiency comparison. Finally, we provide additional characterization of bottlenecks of our current implementation and compare to a prior FPGA implementation.

1) *Overall Performance*

Figure 3 shows the relative performance of our FPGA implementation against the warm CPU and GPU implementations, as the number of non-zero values increases. These results show that the FPGA has equivalent performance up to a certain size, and then it saturates at around 3.9GFlops. This is lower than the ideal ~5.3GFlops because of inefficiencies in the BVB further characterized in Section V.B.4. As expected, CPU and GPU performance saturate at higher non-zero value quantities, since they have higher available memory bandwidths. Overall, the FPGA implementation achieves 65% of CPU performance and 29% of the GPU performance on average. If we only consider speedup after saturation, the FPGA implementation achieves about 32% of CPU performance and 11% of GPU performance. However, there is no reason the FPGA cannot be augmented with additional memory bandwidth. In the next section, we estimate the FPGA performance at a memory bandwidth equivalent to the CPU and GPU.

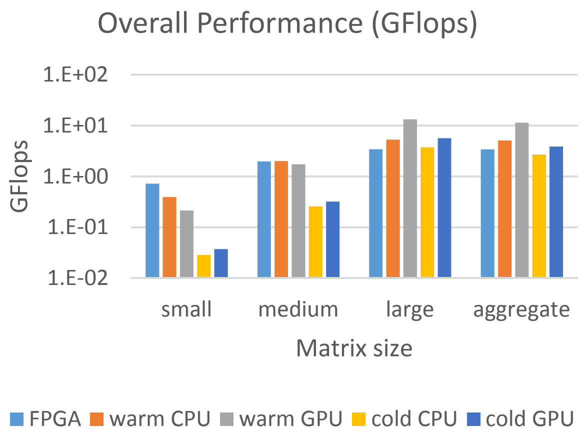


Figure 4. Overall performance for different matrix sizes.

Figure 4 reports average GFlops for matrices with small (up to 10^3), medium (10^3 to 10^5), and large number of non-zero values (more than 10^5), as well as the aggregate over all matrices. Small matrices perform poorly on the GPU due to startup overheads and an insufficient amount of available work to keep the GPU busy and to amortize memory access latency. Small matrices running on the CPU perform better than the GPU since less parallelism is needed for efficient execution on the CPU. In experiments with warm cache (warm CPU and warm GPU), the difference stems from the software that makes the call to the SMVM subroutine. Cold caches add to this overhead because they insert additional cache subsystem latencies. As sizes grow, the overheads in the former category are amortized over the SMVM execution time and CPU/GPU look more attractive. In medium matrices, sufficiently large to amortize software overheads, we can clearly see the effect of caches: matrices are cache-resident, so the effective available memory bandwidth is higher than that offered by main memory. Large matrices do not fit in caches, so the difference between cold and warm cache results do not differ much.

2) Normalized Performance

The next question we want to answer is how the FPGA implementation would compare if it had memory bandwidth equivalent to what is available to the CPU and the GPU. The FPGA we are using has unused transceivers that can be used to connect additional DIMMs, which would increase memory bandwidth proportionally. To take advantage of the increased memory bandwidth, it would be necessary to increase the number of supported channels. Based on our current FPGA capacity and area results, we optimistically estimate that with additional area optimizations applied to our current design, it would be possible to support 3x as many channels. A balanced design with 3x as much memory bandwidth (63.9GB/s) would be almost as fast as a GPU (0.9x) on average. A larger FPGA with enough resources to match the GPU's peak memory bandwidth would be 2.6x as fast as the GPU on average. At the same bandwidth as the CPU, the proposed FPGA design would achieve 1.6x of the CPU performance on average. Table 1 summarizes these results.

Table 1. Scaled memory bandwidth comparison.

Memory Bw GB/s	FPGA/CPU GFlops/GFlops	FPGA/GPU GFlops/GFlops
51.2	1.6x	--
63.9	--	0.9x
192.2	--	2.7x

3) Power Efficiency Comparison

While performance is an important metric to compare, power efficiency should not be left out. Unfortunately, we do not currently have a setup that we could use to measure actual power, so we use maximum power rating as a proxy. The CPU we used for our measurements is specified at 100W maximum power (with the unfair advantage that we are not counting DRAM power for it), while the GPU board is rated at 195W and the FPGA board at 25W, including the two DIMMs used for this evaluation. The FPGA, if augmented to have 3x as much memory bandwidth, would consume 45W, including additional memory controllers and DIMMs.

Table 2 shows measured and scaled MFlops/W numbers. When comparing measured performance numbers, the FPGA implementation performs 2.6x as well as the CPU and 2.3x as well as the GPU. When comparing the scaled FPGA, these numbers grow to 4.4x and 3.8x, respectively.

TABLE 2. POWER EFFICIENCY COMPARISON: FPGA, CPU AND GPU.

Absolute	**FPGA**	**CPU**	**GPU**
Measured	132.9MFlops/W	50.8MFlops/W	58.3MFlops/W
Scaled 3x	221.5MFlops/W	--	--
Ratios	--	**FPGA/CPU**	**FPGA/GPU**
Measured	--	2.6	2.3
Scaled 3x	--	4.4	3.8

4) Characterization

We characterize the main sources of FPGA inefficiency to explain why our implementation does not reach its ideal performance and to identify improvement opportunities. Figure 5 plots the percentage of clock cycles spent stalling on DRAM fetches, ordered by increasing number of non-zero elements. Unsurprisingly, small matrices offer an insufficient amount of work to overlap computation and DRAM accesses.

Increasing the total number of non-zeros reduces this effect, although some large matrices still experience significant stall times (shown on the right of Figure 5). These underperforming matrices are limited by inefficiencies in our current memory controller IP block. Nevertheless, even with current limitations, the FPGA still offers competitive performance relative to CPUs and GPUs.

Figure 6 similarly plots the percentage of clock cycles spent stalling on the Banked Vector Buffer. The BVB stalls when column addresses of a sparse matrix are skewed non-uniformly in banks. These bank conflicts contribute as much as 30% of the total stalls in the worst-case. Nevertheless, excluding a few outliers, the vast majority of matrices experience 15% or less time spent stalling on the BVB.

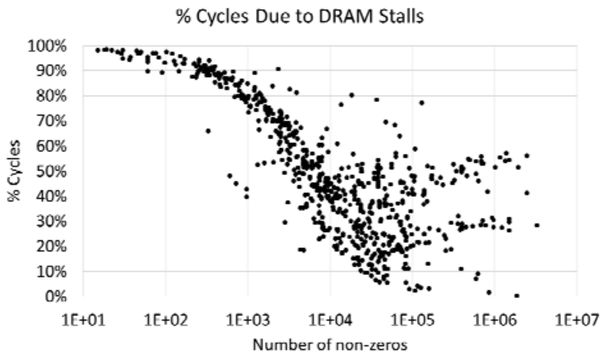


Figure 5. Percentage of cycles stalled due to DRAM. Smaller matrices have insufficient work to overlap DRAM access time and computation.

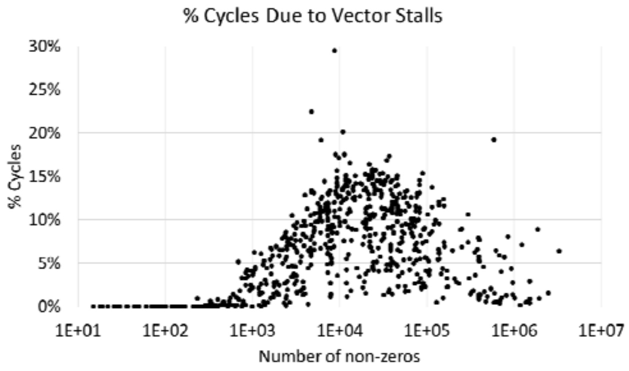


Figure 6. Percentage of cycles stalled due to the Banked Vector Buffer. Stalls occur when there are bank conflicts in the BVB, due to non-uniform column addresses in a given sparse matrix.

5) *Comparison to Previous Work on Convey HC-1*

We briefly compare the proposed design against previous work on accelerating SMVM on Convey HC-1 [13]. The Convey design employs four FPGAs and offers an aggregate off-chip memory bandwidth of 80 GB/s. Our comparison is at best an approximation because the Convey design operates on double-precision operands, while our current design only supports single-precision. We optimistically double the reported throughput of the Convey system for a fair comparison. Table 3 reports on two input matrices measured on both systems. The Convey system achieves better absolute performance, but our per-FPGA performance is significantly higher, by a factor of 1.9X to 2.6X.

6) *FPGA Area*

Table 4 reports the overall FPGA area consumption of various components of the proposed design, including the processing array responsible for 32 channels, the BVB, and the matrix fetcher. Without significant optimization effort, our design consumes a modest 38% area of a mid-end FPGA.

Table 3. Estimate of Relative Performance to SMVM on Convey HC-1 [13].

	Convey HC-1 [13]			This work
	DP GFLOPS	SP (projected GFLOPS)	SP per FPGA (GFLOPS)	SP per FPGA (GFLOPs)
dw8192	1.71	3.42	0.855	2.27
epb1	2.56	5.12	1.28	2.45

Table 4. SMVM Area Consumption on Stratix V D5

	Resources			% Area (Stratix V D5)		
	ALM	M20K	DSP	ALM	M20K	DSP
Total Area	65506	540	32	38	27	2
Proc. Array	26438	160	32	15.3	7.9	2
Mul-Accum	20705	64	32	12	3.2	2
Matrix Fetcher	11225	156	0	6.5	7.7	0
CISR Decoder	3813	32	0	2.2	1.6	0
BVB	25713	128	0	14.9	6.4	0
Address Xbar	9015	0	0	5.2	0	0
Vector Xbar	16242	0	0	9.4	0	0

VI. Related Work

Most previous FPGA accelerators [11][20][21] for SMVM maintain a separate replicated copy of the input vector for every multiplier, resulting in the block RAM bottleneck described in the previous section. Shan et al. [16] addressed this bottleneck by storing the vector in external SRAM, which saved block RAM at the expense of increased latency, reduced access bandwidth and reduced exploitable parallelism. Nagar and Bakos [13] extended this approach by using block RAMs as vector caches to improve memory bandwidth utilization. In this paper, we also avoid replicated copies of the input vector while enabling a much larger input vector to be resident on chip by implementing a highly banked vector buffer.

Multiple sparse matrix encodings have been proposed and used over the years. Compressed Sparse Row (CSR) is a commonly used encoding for CPUs [19], GPUs [1][2], and FPGA implementations [4][11][21]. Prior work introduced FPGA-optimized sparse matrix encodings to improve SMVM performance. Kestur et al. recently proposed an FPGA-specialized encoding called Compressed Variable-Length Bit Vector (CVBV) that reduced matrix storage and bandwidth requirements of CSR by an average of 25% [11]. The CISR encoding proposed in this paper is complementary to CVBV and could potentially be combined to further improve bandwidth, while reducing block RAM requirements. Dickov et al. introduced a row-interleaved compressed row storage encoding that multiplexes dot products from different matrix rows onto a single floating-point adder in order to save resources [4]. In contrast, the presented CISR encoding adopts a different optimization goal of maximizing parallelism while reducing block RAM requirements, which is appropriate for the common situation of memory bandwidth becoming a bottleneck before exhausting FPGA resources.

Several previous approaches have also addressed the limitation of restricting parallelism to a single matrix row. Sun et al. [17] introduced an Input Pattern Vector along with a specialized SMVM architecture that enables flexible parallelization of operations across multiple matrix rows. Our approach also parallelizes operations across rows, while additionally reducing block RAM requirements to enable sparse matrices significantly larger than those evaluated by Sun et al [17]. Dickov's SMVM implementation [4] also processed multiple rows, but with the goal of minimizing resources as opposed to maximizing parallelism.

VII. CONCLUSION

In this paper, we introduce an FPGA-optimized SMVM architecture that uses a specialized CISR encoding to efficiently process multiple rows of a matrix in parallel, coupled with a highly banked buffer design that eliminates replication of buffered vectors, enabling larger vectors to be stored on-chip. We show that the presented architecture performs within about two thirds of CPU SMVM performance, even though it has 2.4x lower DRAM memory bandwidth, and within almost one third of GPU SMVM performance, even at 9x lower memory bandwidth. Additionally, our FPGA design consumes a maximum of 25W, for power efficiencies 2.6x and 2.3x higher than CPU and GPU, respectively. When supplied with the same memory bandwidth, we predict this FPGA architecture is 1.6x faster than the CPU and 2.7x faster than the GPU, and even more power-efficient (4.4x and 3.8x, respectively).

ACKNOWLEDGMENT

We would like to thank Doug Burger and the Catapult team for their support and help with this project. We would also like to thank Adrian Macias from Altera for helping with the accumulator implementation.

REFERENCES

[1] N. Bell and M. Garl. Efficient sparse matrix-vector multiplication on CUDA. Technical report, NVIDIA, 2008.

[2] 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, SC'09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[3] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. ACM Trans. Mathematical Software, 38(1):1:1–1:25, Nov. 2011.

[4] B. Dickov, M. Pericàs, N. Navarro, E. Ayguadé, and D. D. D. Computadors. Row-interleaved streaming data flow implementation of sparse matrix vector multiplication in FPGA. In 4th Workshop on Reconfigurable Computing, WRC-2010. Vol. 104. 2010.

[5] J. Fowers and G. Stitt. Dynafuse: dynamic dependence analysis for FPGA pipeline fusion and locality optimizations. In Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA'13, pages 201–210, New York, NY, USA, February 2013. ACM.

[6] A. George, H. Lam, and G. Stitt. Novo-G: At the forefront of scalable reconfigurable supercomputing. *Computing in Science Engineering*, 13(1):82–86, Jan.-Feb. 2011.

[7] 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 Proceedings of the International Conference on Field Programmable Logic and Applications, 2007. FPL 2007, pages 786–791, 2007.

[8] P. Guo and L. Wang. Auto-tuning CUDA parameters for sparse matrix-vector multiplication on GPUs. In Proceedings of the International Conference on Computational and Information Sciences (ICCIS), 2010, pages 1154–1157, 2010.

[9] V. Hopytoff. Japanese 'K' computer is ranked most powerful. *New York Times*. June 19, 2011. http://www.nytimes.com/2011/06/20/technology/20computer.html?_r=2.

[10] Intel Math Kernel Library. 2007. http://software.intel.com/en-us/intel-mkl.

[11] S. Kestur, J. D. Davis, and E. S. Chung. Towards a universal FPGA matrix-vector multiplication architecture. In Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, FCCM'12, pages 9–16, Washington, DC, USA, 2012. IEEE Computer Society.

[12] N. McVicar, W. L. Ruzzo, and S. Hauck. Accelerating ncRNA homology search with FPGAs. In Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA'13, pages 43–52, New York, NY, USA, 2013. ACM.

[13] K. Nagar and J. Bakos. A sparse matrix personality for the Convey HC-1. In Proceedings of the IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2011, pages 1–8, 2011.

[14] NVIDIA. CUSPARSE: CUDA Toolkit Documentation, 2014, http://docs.nvidia.com/cuda/cusparse/.

[15] K. Sano, Y. Hatsuda, and S. Yamamoto. Multi-FPGA accelerator for scalable stencil computation with constant memory-bandwidth. IEEE Transactions on Parallel and Distributed Systems, 25(3):695–705, Mar. 2014.

[16] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang. FPGA and GPU implementation of large scale SPMV. In Proceedings of the IEEE 8th Symposium on Application Specific Processors (SASP), pages 64–70, 2010.

[17] S. Sun, M. Monga, P. Jones, and J. Zambreno. An I/O bandwidth-sensitive sparse matrix-vector multiplication engine on FPGAs. IEEE Transactions on Circuits and Systems I: Regular Papers, 59(1):113–123, 2012.

[18] S. Sun and J. Zambreno. A floating-point accumulator for FPGA-based high performance computing applications. In Proceedings of the International Conference on Field-Programmable Technology. FPT 2009, pages 493–499, 2009.

[19] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC'07, pages 38:1–38:12, New York, NY, USA, 2007. ACM.

[20] Y. Zhang, Y. Shalabi, R. Jain, K. Nagar, and J. Bakos. FPGA vs. GPU for sparse matrix vector multiply. In Proceedings of the International Conference on Field-Programmable Technology. FPT 2009, pages 255–262, 2009.

[21] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable gate arrays, FPGA'05, pages 63–74, New York, NY, USA, 2005. ACM.

[22] NVIDIA. https://developer.nvidia.com/cusp