Reduce, Reuse, Recycle (R^3): a Design Methodology for Sparse Matrix Vector Multiplication on Reconfigurable Platforms

Kevin Townsend, Joseph Zambreno
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA, USA
ktown@iastate.edu, zambreno@iastate.edu

***Abstract*—Sparse Matrix Vector Multiplication (SpMV) is an important computational kernel in many scientific computing applications. Pipelining multiply-accumulate operations shifts SpMV from a computationally bounded kernel to an I/O bounded kernel. In this paper, we propose a design methodology and hardware architecture for SpMV that seeks to utilize system memory bandwidth as efficiently as possible, by *Reducing* the matrix element storage with on-chip decompression hardware, *Reusing* the vector data by mixing row and column matrix traversal, and *Recycling* data with matrix-dependent on-chip storage. Our experimental results with a Convey HC-1/HC-2 reconfigurable computing system indicate that for certain sparse matrices, our R^3 methodology performs twice as fast as previous reconfigurable implementations, and effectively competes against other platforms.**

***Index Terms*—HPRC, High Performance Reconfigurable Computing, SpMV, Convey**

I. Introduction

Sparse Matrix Vector Multiplication (SpMV) kernels have uses ranging from web mining (eg. Google's PageRank algorithm [1]) to simulation (eg. cardiac tissue simulation [2]) to image processing (eg. editing and transformations [3]).

Many previous approaches describe reasonably efficient implementations on RISC and x86 processors [4], [5], Supercomputers [6], and GPGPUs [7]. However, comparable implementations on reconfigurable computers fail to reach the potential performance of these systems [8]–[12]. In this paper, we propose a Reduce, Reuse, Recycle (R^3) methodology and corresponding hardware implementation for SpMV, which illustrate how reconfigurable implementations can compete with other platforms. Our R^3 methodology takes advantage of the ability to design an application-specific architecture with high parallelism and bandwidth-conscientiousness.

The novelty of our R^3 approach stems from three major aspects. First, we utilize a matrix storage format (the R^3 format) that *reduces* the memory storage requirements of a given sparse matrix. Second, our matrix value ordering scheme (GRMLCM) enables *reuse* of the x vector values. Finally, our approach makes use of an Intermediator block that stores intermediate y vector values, enabling our design to *recycle* storage space before sending values to off-chip memory.

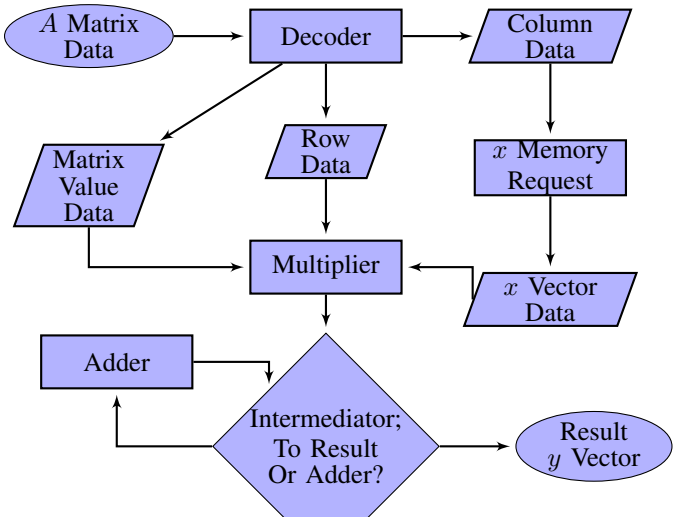


Figure 1: Data flow of an R^3 SpMV processing element. The processing element needs the column data before accessing the vector data.

The following outlines the remainder of the paper. In Section II, we cover the basics of SpMV as a computation kernel, followed by a description of our R^3 approach and implementation on the Convey HC-1/HC-2 in Section III. In Section IV, we show our results using two sets of test cases. Lastly Section V describes possible future work and implications of our work.

II. Background

SpMV or sparse-matrix (dense-)vector multiplication, has the mathematical form: $y = A \times x$, which expands to:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_i \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} A_{11} & \dots & A_{1j} & \dots & A_{1n} \\ A_{21} & \dots & A_{2j} & \dots & A_{2n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ A_{i1} & \dots & A_{ij} & \dots & A_{in} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ A_{m1} & \dots & A_{mj} & \dots & A_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_j \\ \vdots \\ x_n \end{bmatrix} \quad (1)$$

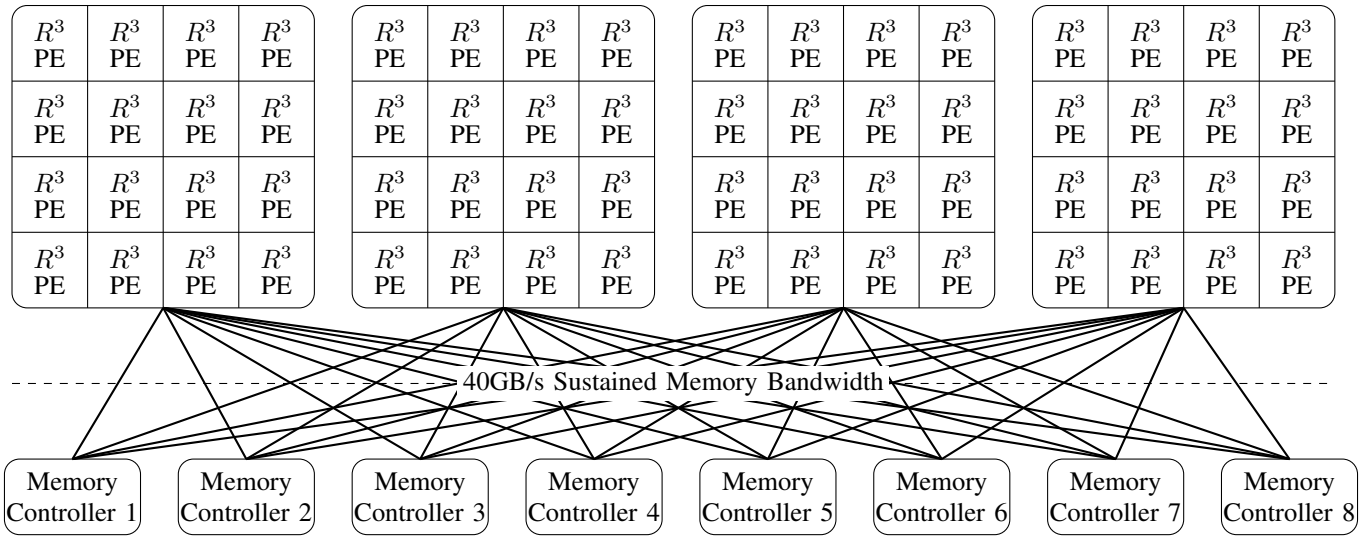


Figure 2: R^3 implementation on the Convey HC-1/HC-2 co-processor: 4 Virtex-5 LX330 tiled with 16 R^3 SpMV processing elements each. Each Virtex-5 chip connects to 8 memory controllers, which enables each chip to have access to all of the co-processor's memory.

As an example of SpMV as a computational kernel, let us look closer at the PageRank algorithm [1]. PageRank starts with a list of web pages and the links to and from each page. Using that information it creates a square matrix A. Then it computes the eigenvector of A, or more precisely an approximate eigenvector of A. It does this by an iterative method. First, it initializes a vector x_1, of length equal to the number of pages, to all ones. Second, it computes $x_2 = A * x_1$. Third, it repeats the second part until the vector meets some convergence criteria. Due to this repetition, the SpMV kernel strongly influences the overall runtime.

The most common parallelization approach for SpMV partitions the matrix with horizontal cuts. R^3 does this by splitting the matrix into 64 chunks. For example, assume rows A_1 to A_{48} contain $1/64^{th}$ of the non-zero (nnz) elements in A, then the first R^3 SpMV processing element would use the matrix elements between rows A_1 and A_{48}, and the entire x vector, and would store the y vector values between y_1 and y_{48}.

SpMV computation kernels use matrix values once and use the values in the vector only a few times (4 to 172 in our test matrices). Unlike matrix matrix multiplication, SpMV requires a significant amount of I/O bandwidth compared to the amount of floating point computations required. Most architectures have more computational power than I/O bandwidth (FPGAs included). Therefore, the I/O bound of SpMV has more importance. Even though the Convey HC-1/HC-2 (the target platform we utilize) has less I/O bandwidth than most GPUs, it can use it more efficiently.

III. Approach and Implementation

Our R^3 approach has a simple high-level philosophy: create an SpMV architecture block and replicate it as many times as possible on the target chip, in our case the Xilinx Virtex-5 LX330 [13] chip. Each SpMV block contains one double precision floating-point multiplier and one double precision floating-point adder. In this section, we present the approach and implementation together because the implementation provides an example of the approach and the motivation for the approach.

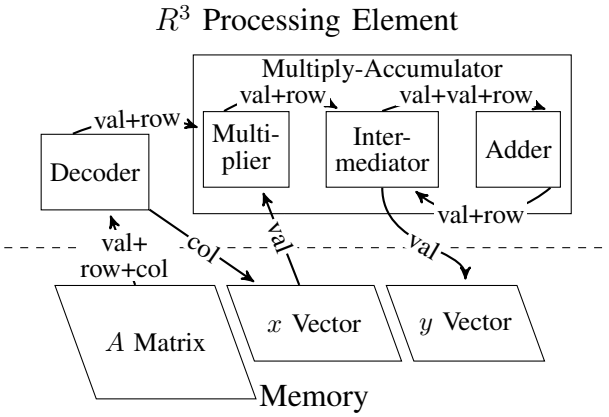


Figure 3: A single R^3 processing element. The arrows show the flow of data through the processing element. Although this diagram shows the memory access to each of the 3 places in memory as separate, they share one memory port. The diagram also does not show the FIFOs that help keep the pipeline full.

A. Architecture

As previously mentioned, we used a Convey HC-1/HC-2 [14] to implement R^3. Figure 2 shows the layout of the Convey architecture. It has a co-processor board with 4 large FPGAs (Xilinx Virtex-5 LX330) and connects to a host board via PCI Express. The processing elements on the application engines do the heavy lifting of the platform. Figure 1 gives a high-level data-flow diagram of the processing element. We found 64

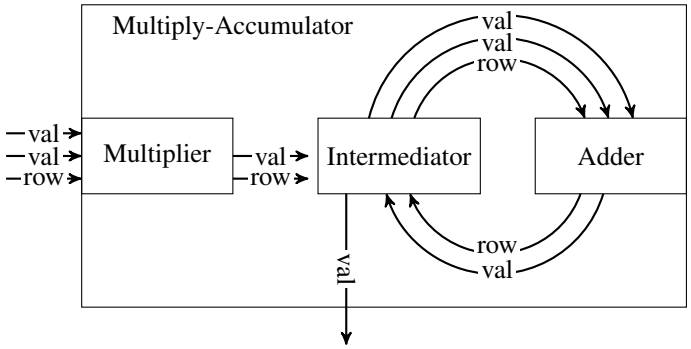


Figure 4: The no-stall multiply-accumulator block handles multiple intermediate values at a time. This allows multiple intermediate values in the adder pipeline.

blocks (16 per application engine) to work well. Our synthesis reports shows about 75% utilization of the LX330.

From here we looked at our potential performance. With 64 adders and 64 multipliers running at 150MHz, our design can compute 19 Gflops (billion floating point operations per second). Looking at I/O bandwidth, the Convey HC-1/HC-2 has 40 GB/s of sustainable I/O bandwidth. Assuming 20 bytes of I/O for each multiply-accumulate operation (4 bytes for the index, 8 bytes for the matrix value and 8 bytes for the vector value) R^3 could compute only 4 Gflops. This further motivated our exploration of ways to reduce the I/O requirements. Our R^3 approach also tries to use the multiply-accumulator as efficiently as possible (by not stalling), because it can also bound performance.

B. Designing a No-Stall Multiply Accumulator

Designing a fast SpMV implementation relies on designing a no-stall multiply accumulator (MAC). An inefficient engine stalls when a matrix and associated vector value pair arrives every or nearly every clock cycle. The long latency of floating point addition makes this hard. To solve this our approach works on multiple intermediate y vector values and does the additions out of order. For example in computing $1+2+3+4$ the MAC does $(1+2)+(3+4)$. This removes the data dependency of adding 1 and 2 before processing 3. CPUs and GPUs compute floating point addition in order (eg. $((1+2)+3)+4$). This means results may differ slightly, because changing the order of floating point addition can change the result [15].

We designed a block called an Intermediator (Section III-C) capable of storing 32 intermediate y vector values. This design had an interesting side effect that it would allow the matrix to be traversed in a loosely row major traversal and the MAC would still work correctly. More specifically, the matrix elements in one set of 16 rows can be traversed in any way just as long as all the elements are traversed before going to the next 16 rows. So we created a traversal that follows this and allows for easy reuse of x vector values, which we called GRMLCM16 (Section III-D).

C. Intermediator

The Intermediator (Figure 5) takes in two values, one from the multiplier's result and one from the adder's result and outputs a pair of values to be added. The dual-port Block RAM (middle block in Figure 5) stores intermediate values until an element in the same row appears.

For most matrices, the Block RAM cannot store the entire intermediate y vector. Also the control logic needs to remember the state of each slot in RAM (empty or full). Remembering the state of each RAM location and updating that state requires complicated logic. We approach this problem by limiting the number of active intermediate values to 32. Of these 32, we consider the bottom 16 fading because they soon leave the active window. Once a new element with a larger row index arrives, the window increases its index by 16. The Intermediator eventually stores the 16 "faded" elements no longer in the active window. The Intermediator handles each case that can occur properly:

Case 1: (Figure 5g) The trivial case, no valid input arrives. If the "to result" block has values, it outputs a value. An overflow FIFO (explained in case 6) outputs a value if it has values.

Case 2: (Figure 5d) Only one value arrives (valid) and the row corresponds to an empty cell. The value goes into the empty cell. If the "to result" window has values, it outputs a result, and if the FIFO has values it outputs a set to the adder.

Case 3: (Figure 5a) Similar to case 2 except with a full cell. It retrieves the value in the ram slot and goes to the adder with the input value. The state of the cell gets updated to empty.

Case 4: (Figure 5b, 5h) Both values have row indexes that correspond to empty cells in the Block RAM. Both values get stored in the Block RAM and both cells switch to full. If the FIFO from the Block RAM to the output has values it sends one set of values to the output.

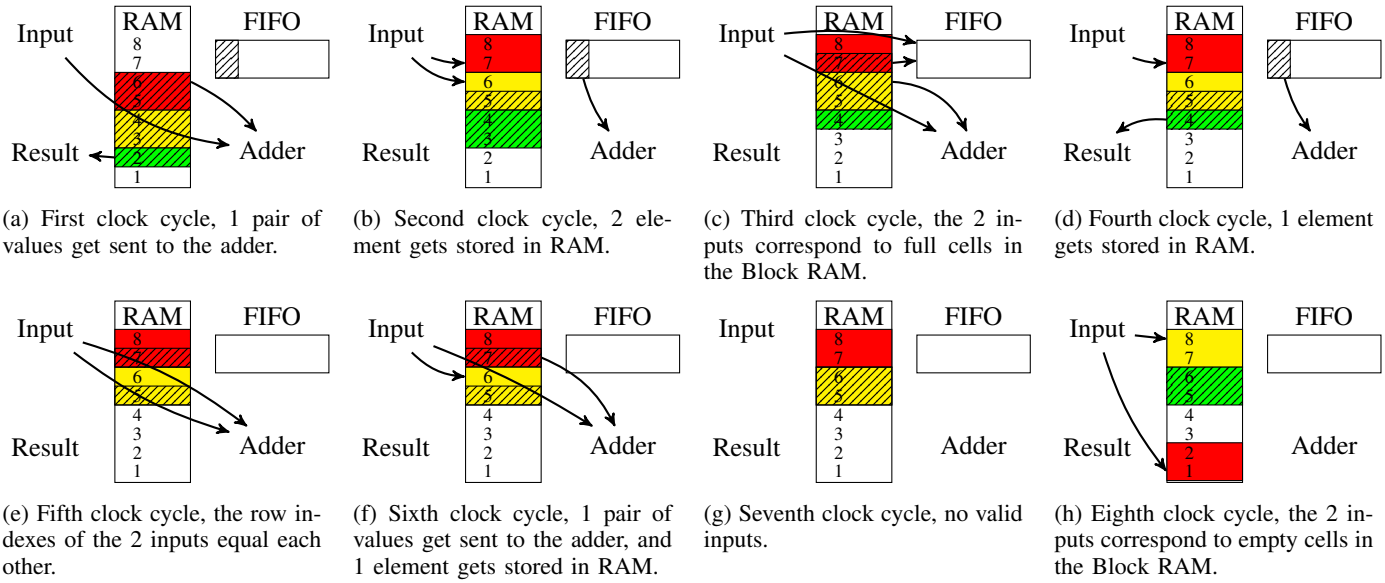
Case 5: (Figure 5f) One value has a row index corresponding to an empty cell, and the other to a full cell. The first value goes in the empty cell and the full cell goes to the output with the second value.

Case 6: (Figure 5c) Both values have row indexes that correspond to full cells in the Block RAM. One input value and corresponding Block RAM cell goes to the output. The output can only handle one output pair at a time, so the other input value and corresponding Block RAM cell goes to the FIFO.

Case 7: (Figure 5e) The inputs Row0 and Row1 equal each other. In this case the the values go through the pipeline and do not use the Block RAM in the center of the block. They simply pass through to the adder with the row index.

For example, consider a simpler case where the active window equals 2. Figure 5 shows 8 clock cycles of operation. At every clock cycle up to 2 valid input values with corresponding row indexes arrive. For simplicity we do not show the values being calculated in the figure.

We should consider the possibility that the windows could advance before all the final values reach the fading window. If this does not happen then the MAC has to stall or else incorrect



(a) First clock cycle, 1 pair of values get sent to the adder.

(b) Second clock cycle, 2 element gets stored in RAM.

(c) Third clock cycle, the 2 inputs correspond to full cells in the Block RAM.

(d) Fourth clock cycle, 1 element gets stored in RAM.

(e) Fifth clock cycle, the row indexes of the 2 inputs equal each other.

(f) Sixth clock cycle, 1 pair of values get sent to the adder, and 1 element gets stored in RAM.

(g) Seventh clock cycle, no valid inputs.

(h) Eighth clock cycle, the 2 inputs correspond to empty cells in the Block RAM.

Figure 5: This shows a simple example of the Intermediator running for four clock cycles. For simplicity, the size of the active window (active intermedate y values) is 2.

values would occur in the result. Let us look at the worst case. Only inputs from the adder correspond with to the elements in the fading window. So, the theoretical worst case occurs with a full adder pipeline and each value corresponds to the same row. Every 16 cycles (the adder pipeline length) the number of elements with the same row in the pipeline cuts in half. Therefore the worst case would take 80 ($(log_2(16)+1)\times 16$) clock cycles to guarantee that the fading window only has final y vector values. The worst case would also advance the window in 16 clock cycles (1 element per row in the matrix for those 16 rows corresponding to the active window). So in theory the MAC could stall, but in practice this never happens.

D. GRMLCM

Vector values get reused multiple times. The SpMV kernel uses each vector value by precisely the number of elements in the corresponding column of the matrix. We want to load the vector values as few times as possible. We can achieve this through caching or changing the matrix traversal. Column major traversal allows perfect reuse of the x vector, but the intermediate y vector would get huge. Our approach compromises by doing column major traversal on the small scale and row major traversal on the large scale. Let us call the traversal Global Row Major Local Column Major (GRMLCM). Consider the following example matrix: a row-major traversal would read the matrix in the following order: $[A_{11}, A_{14}, A_{17}, A_{25}, A_{28}\ldots]$. Conversely, GRMLCM with local size equal to four would read it as: $[A_{11}, A_{41}, A_{32}, A_{33}, A_{14}\ldots]$. For concision, let us call GRMLCM, with local size equal to four, GRMLCM4. This approach has similarities to register- or cache-blocking in [5]. Our engine can handle a value of 16 for the local column major size or GRMLCM16.

Table I: Packet Encoding

Header Encoding	Value	Delta	Packet Size
0 (0b000)	End of Row		1 byte
1 (0b001)	common	5 bits	2 bytes
2 (0b010)	common	13 bits	3 bytes
3 (0b011)	common	29 bits	5 bytes
4 (0b100)	common	45 bits	7 bytes
5 (0b101)	uncommon	5 bits	9 bytes
6 (0b110)	uncommon	21 bits	11 bytes
7 (0b111)	uncommon	45 bits	14 bytes

E. R^3 Compression Format and Decoder Design

Often iterative SpMV computation time dwarf the preprocessing time of the matrix (like changing the traversal to GRMLCM16). Since the matrix loading takes more than half of the memory bandwidth, it makes sense to try to compress the matrix. We created R^3 Compression Format for this purpose. This compression converts each matrix element into a variable sized packet. The size of the packet depends on how compressible the value and index information is.

Matrix storage has a significant impact on performance. The more compressed the matrix, the faster it loads to the chip. The least compressed (simplest) scheme takes 16 bytes per element (Coordinate list, COO format). 4 bytes to store the row, 4 bytes to store the column and 8 bytes to store the double precision value. With row compression this goes down to 12 bytes (Compressed Sparse Row, CSR format), because the row changes infrequently in a row major traversal.

Many more storage schemes exist [7], [16]–[19] that do not port well to reconfigurable computers. Our scheme focuses on compression, and compresses each element to between 2 and 14 bytes.

Table II: Matrix Statistics

Matrix	Field	dimensions	nnz	nnz/row	Unique Values	R^3 Gflops	2× Intel E5-2690	Nvidia Tesla M2090
Test Set I								
dense	Example	2,000x2,000	4,000,000	2,000	1	13.6	14	**23**
consph	FEM/Speres	83,334x83,334	6,010,480	72	1,574,940	8.7	11	**15**
cant	FEM/Cantilever	62,451x62,451	4,007,383	64	107	12.7	12	**17**
rma10	FEM/Harbor	46,835x46,835	2,329,092	49	1	13.6	**24**	11
qcd5_4	QCD	49,152x49,152	1,916,928	39	1	12.8	**30**	20
shipsec1	FEM/ship	140,874x140,874	3,568,176	25	132,862	7.9	10	**11**
mac_econ_fwd500	Economics	206,500x206,500	1,273,389	6.2	118,306	5.9	**23**	6
mc2depi	Epidemiology	525,825x525,825	2,100,225	4.0	3,584	6.2	21	**22**
scircuit	Circuit	170,998x170,998	958,936	5.6	282,665	6.2	**12**	6
Test Set II							**Nagar & Bakos [10]**	**Nvidia Tesla S1070**
dw8192	Electromagnetics	8,192x8,192	41,746	5.1	673	**2.1**	1.7	0.5
t2d_q9	Structural	9,801x9,801	87,025	8.9	1,383	**3.8**	2.5	0.9
epb1	Thermal	14,734x14,734	95,053	6.5	73,230	**3.3**	2.6	0.8
raefsky1	Fluid Dynamics	3,242x3,242	294,276	91	271,382	**5.2**	3.9	2.6
psmigr_2	Economics	3,140x3,140	540,022	172	531,668	**4.9**	3.9	2.8
torso2	Model	115,967x115,967	1,033,473	8.9	806,653	**6.4**	1.2	3.0

1) Value Compression: It was noted in [20] that sparse matrices often have repeated values. Table II shows the unique value count of the matrices we tested. Our implementation records the 256 most common values in the matrix. More precisely, our implementation records the 256 most common values in each 1/64th of the matrix. This happens because each processing element has its own Block RAM of repeated values. So for every element in the matrix the value gets retrieved as one byte (a common value) or eight bytes (an uncommon value). As described later there exists a three-bit header for overhead. One could view this as approximately one bit of overhead for value compression and two bits of overhead for index compression.

2) Index Compression: [20] also noted the clumpy distribution of values in most matrices. So instead of storing the row and column of each element (or just the column as in CSR), we store the distance from the previous element or the delta value. The delta value equals the number of elements between the previous and current element using whatever traversal method the matrix uses, in our case GRMLCM16. In the previous matrix example the deltas are: $[0, 3, 15, 3, 1 \ldots]$. Our method stores the delta in a space between 5 and 45 bits (5, 13, 21,29, or 45 bits).

3) Matrix Data Storage: We store the matrix data as a stream of elements or packets. One coming right after the next. The first 3 bits of each packet represents the header of the packet. The next 5 bits represent the 5 least significant bits of the delta. The next 8 or 64 bits (depending on the header) represents the index to the common value or the uncommon value, respectfully. The rest of the packet represents the rest of the delta value. Table I shows the different types of packets possible. An "End of Row" (similar to a newline character) packet type helped with the design of the decoder.

Using the example matrix earlier, the start encodes as: 0x12610709091913790F191801 (read right to left). The first 2 bytes (0x1801 or 0b0001100000000001) contain the first element. It decodes as: header: 1, value: uncommon index 24, delta: 0. The second value (0x0F19 or 0b0000111100011001) decodes as: header: 1, value: uncommon index 15, delta: 3.

Table III: Processors

Chip	E5-2690	Tesla M2090	Virtex-5 LX330	Virtex-6 HX565T[c]
Type	CPU	GPU	FPGA	FPGA
Lithography	32nm	40nm	65nm	40nm
Clock (Ghz)	2.9-3.8	1.3	0.55	0.60
Gflops (dp)	175	655	9.9[a]	48.6[a]
I/O (GB/s)	51.5	177	50[a]	100[a]
Cache (MB)	20	0.77	1.3[b]	4.1[b]

[a] Author's estimate
[b] Total Block RAM storage
[c] Only used for comparison

IV. Experimental Results

We used 2 sets of matrices as benchmarks. The first set came from [7]. These matrices range from 1 million non-zero elements to 6 million non-zero elements. The second set came from [10]. These matrices range from 40 thousand to 1 million non-zero elements. We acquired the matrices from the Florida Matrix Collection [21] and from Nvidia [7].

We compared against three different platforms/implementations. [7] and [10] published the results using test set I and test set II respectively. Intel published the performance of SpMV using their MKL (Math Kernel Library) [4] on a pair of Intel E5-2690 CPUs, using the same matrices that were listed in [7]. Also, [22] used a more recent GPU (Nvidia Tesla M2090) compared to the one in [7] (Nvidia GTX 285). We use the performance numbers from [22] using the Nvidia Cusp library. Table III lists all of the processors used in the mentioned platforms. We also attempted to compare the processors to each other and to the more current Xilinx Virtex-6 HX565T [23], used in the Convey XM-1 [14]. We calculated the Gflops number on the FPGA by estimating the number of multiply-accumulate blocks that could fit on a FPGA and assuming a clock speed of half the FPGA's rating. The comparison makes some simplifying assumptions. For

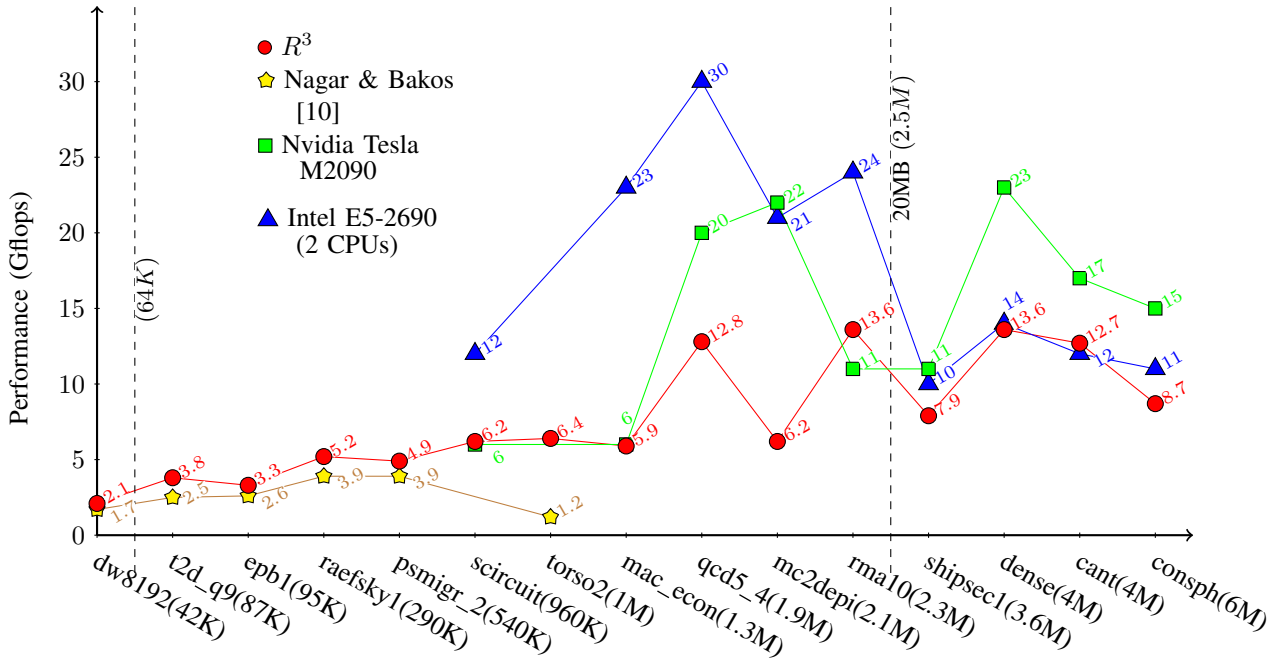


Figure 6: nnz vs Performance on each platform. The small matrices, ones around 64K or less, performed poorly on most platforms, due the overhead. CPUs experience the opposite effect. They take a performance hit once the matrix no longer fits in cache.

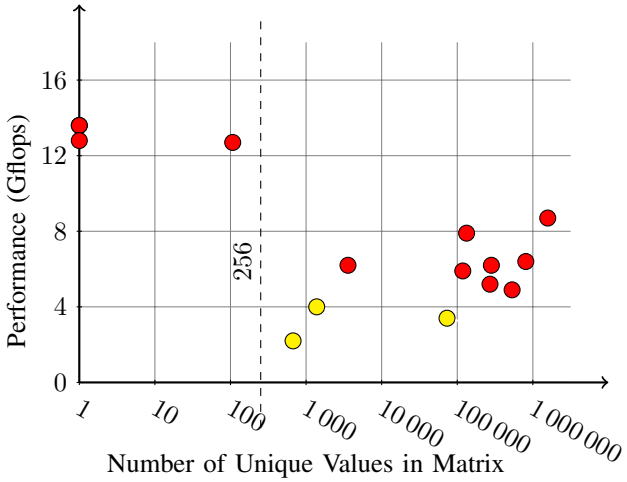


Figure 7: Unique values in a matrix vs the performance of R^3. Matrices with fewer than 256 unique values (only common elements exist) enables R^3 format to compress much better. The ●'s are outliers due to their size (see Figure 6).

example, in the FPGA I/O bandwidth, memory addressing would cut into the actual memory bandwidth.

We measure performance in Gflops: using M as the height of the matrix and nnz as the number of non-zeros in the matrix, the performance equals $(2 * nnz - M)$ / runtime, as we have nnz multiply operations and $nnz - M$ addition operations.

A. Analysis

Table II shows the performance of R^3 on each matrix. Test set I compares our R^3 implementation to an Intel machine (2 x 8-core Intel Xeon E5-2690, 20MB cache 2.9Ghz) using the Math Kernel Library (MKL) [4] and an Nvidia GPU (Tesla M2090) in [22]. Test set II compares R^3 to a previous Convey HC-1/HC-2 personality [10] and the Nvidia Tesla S1070 used in their paper. As the Table shows, although the Intel and Nvidia platforms perform well, the performance varies. The Tesla S1070 in [10] uses CSR to store the matrix instead of using the best performing storage format.

Table II attempts to show how the matrix features effect performance of our implementation. General trends do occur. For obvious reasons the dense matrix performs the best. As shown in Figure 7, matrices with many repeating values perform well due to better compression. All of the matrices with less than 256 unique values perform at more than 12 Gflops where as the matrices with less than 256 unique values performed at less than 9 Gflops. Figure 6 shows larger matrices perform a little better than smaller ones where the overhead of starting the operation can effect a larger percent of the runtime. It takes about 1,000 clock cycles to get to an efficient steady-state operation. With 64 processing elements this creates about 64,000 empty multiply-accumulate operations of overhead. The engine performs better when $nnz >> 64K$ and worse when $nnz < 64K$ as matrix dw8192 illustrates.

Figure 6 also highlights the effect of the large 20MB L3 cache on the Intel E5-2690 CPU. On four out of the nine matrices the Intel platform handedly outperforms R^3 and

the Nvidia platform, but does not "win" on the four largest matrices in the set. This happens because those matrices do not fit in the cache.

V. Future Work

Many solutions to important problems require SpMV with large sparse matrices. When SpMV does not take a significant amount of a program's runtime, simple and naive implementations of SpMV work well. But on large problems where SpMV requires a significant amount of runtime, fast complex implementations warrant attention. R^3 begins to show high-performance reconfigurable computing can be competitive with existing state-of-the-art.

R^3 does not perform to its full potential. The lag of reconfigurable computing technology attributes to this, but there exists ways to improve this design on the current HC-1/HC-2 hardware. We are exploring the possibility of creating a hardware design that is capable of more vector reuse. Also we are looking into better matrix compression schemes.

A. Reconfigurable Computing Niche

General purpose hardware (CPUs and GPUs) are undeniably efficient at SpMV. However, CPUs begin to perform badly when the matrices get larger than 20MB, due to slow memory I/O. GPUs can not efficiently handle giant matrices, due to the fact that no commercial GPU card has more than 8GB of ram. SpMV also does not lend itself well to traditional Supercomputing clusters, due to relatively slow interconnect speeds. In the near future, it is likely that high-performance reconfigurable platforms such as the HC-2 will outpace other platforms.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) under the awards CNS-1116810 and CCF-1149539.

References

[1] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford, Technical Report 1999-66, Nov. 1999.

[2] D. Sato, Y. Xie, J. N. Weiss, Z. Qu, A. Garfinkel, and A. R. Sanderson, "Acceleration of cardiac tissue simulation with graphic processing units," *Medical and Biological Engineering and Computing (MBEC)*, vol. 47, no. 9, pp. 1011–1015, Sep. 2009.

[3] Y. Wang, H. Yan, C. Pan, and S. Xiang, "Image editing based on sparse matrix-vector multiplication," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, May 2011, pp. 1317–1320.

[4] *Intel Math Kernel Library Reference Manual*, 11th ed., Intel, 2012. [Online]. Available: http://software.intel.com/

[5] E. Im, "Optimizing the performance of sparse matrix-vector multiplication," Ph.D. dissertation, University of California, Berkeley, Jun. 2000.

[6] G. Schubert, G. Hager, H. Fehske, and G. Wellein, "Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+OpenMP programming," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium and PhD Forum (IPDPSPW)*, May 2011, pp. 1751–1758.

[7] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," Nvidia, Technical Report NVR-2008-004, Dec. 2008.

[8] M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programable Gate Arrays (FPGA)*, 2005, pp. 75–85.

[9] M. Gerards, "Streaming reduction circuit for sparse matrix vector multiplication in FPGAs," Master's thesis, University of Twente, Aug. 2008.

[10] K. Nagar and J. Bakos, "A sparse matrix personality for the Convey HC-1," in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2011, pp. 1–8.

[11] J. Sun, G. Peterson, and O. Storaasli, "Sparse matrix-vector multiplication design on FPGAs," in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2007, pp. 349–352.

[12] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programable Gate Arrays (FPGA)*, 2005, pp. 63–74.

[13] *Virtex-5 Family Overview*, 5th ed., DS100, Xilinx, Feb. 2009.

[14] *Convey Reference Manual*, 1st ed., Convey, Richardson, TX, May 2012.

[15] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, Mar. 1991.

[16] W. Cao, L. Yao, Z. Li, Y. Wang, and Z. Wang, "Implementing sparse matrix-vector multiplication using CUDA based on a hybrid sparse matrix format," in *Proceedings of the IEEE International Conference on Computer Applications and System Modeling (ICCASM)*, Oct. 2010, pp. 161–165.

[17] V. Karakasis, G. Goumas, and N. Koziris, "A comparative study of blocking storage methods for sparse matrices on multicore architectures," in *Proceedings of the SIAM Conference on Computational Science and Engineering (CSE)*, Aug. 2009, pp. 247–256.

[18] Y. Sazeides and J. E. Smith, "The predictability of data values," in *Proceedings of the ACM/IEEE International Symposium on Microarchitectures (MICRO)*, 1997, pp. 248–258.

[19] M. Burtscher and P. Ratanaworabhan, "Fpc: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers (TC)*, vol. 58, no. 1, pp. 18–31, Jan. 2009.

[20] K. Kourtis, G. Goumas, and N. Koziris, "Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, Sep. 2008, pp. 511–519.

[21] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, 2011. [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices

[22] A. Monakov, "Specialized sparse matrix formats and SpMV kernel tuning for GPUs," in *Proceedings of the GPU Technology Conference (GTC)*, May 2012.

[23] *Virtex-6 Family Overview*, 2nd ed., DS150, Xilinx, Jan. 2012.

[24] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Understanding the performance of sparse matrix-vector multiplication," in *Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Feb. 2008, pp. 283 –292.

[25] "Matrix Market," National Institute of Science and Technology. [Online]. Available: math.nist.gov/MatrixMarket

[26] K. Nagar and J. Bakos, "A high-performance double precision accumulator," in *Proceedings of the International Conference on Field-Programable Technology (FPT)*, Dec. 2009, pp. 500–503.

[27] S. Sun and J. Zambreno, "A floating-point accumulator for FPGA-based high performance computing applications," in *Proceedings of the International Conference on Field-Programable Technology (FPT)*, Dec. 2009, pp. 493–499.

[28] 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—Part I: Regular Papers (TCAS-I)*, vol. 59, no. 1, pp. 113–123, 2012.

[29] S. Vangal, Y. Hoskote, N. Borkar, and A. Alvandpour, "A 6.2-GFlops floating-point multiply-accumulator with conditional normalization," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 41, no. 10, pp. 2314–2323, Oct. 2006.